SQL/XNF - Processing Composite Objects as Abstractions over Relational Data

B. Mitschang*, H. Pirahesh, P. Pistor**, B. Lindsay, N. Südkamp**
IBM Almaden Research Center
San Jose, CA 95120, USA
e-mail: {pirahesh, bruce}@almaden.ibm.com

Abstract

Complex applications, such as design applications, multi-media applications, and even advanced business applications can benefit significantly from a database language that supports composite (or complex) objects. Usually such data is inter-related with the data used by traditional applications, such as accounting, ordering, bill of material, and repair and maintenance tracking. Consistency of such data is of utmost importance in applications such as those of aerospace. Hence, sharing of the data among traditional applications and composite object applications is important.

Our approach, called SQL Extended Normal Form (short SQL/XNF or XNF) enhances relational technology by a Composite Object facility, which comprises not only extraction of composite objects from existing databases, but also efficient navigation and manipulation facilities provided by an appropriate application programming interface.

The language itself allows sharing of the database among normal form SQL applications and composite object applications. It provides proper subsetting of the database and subsequent structuring exploiting subobject sharing and recursion, all based on its powerful composite object constructor concept, which is closed under the language operations. XNF is integrated into the relational framework, thus benefiting from the available technology, e.g. relational engine, query optimization. Currently, a major portion of SQL/XNF is operational in Starburst extensible database system at IBM Almaden Research Center.

1. Motivation and Introduction

Existing "second generation" data base systems are focussed on business applications (e.g. booking, storekeeping, cost accounting, project management and decision support). It is widely agreed now that second generation DBMSs are inadequate for a broader class of applications that deal with composite objects (structured data), as opposed to flat relations. Such applications include office automation, medicine, computer aided software engineering (CASE), artificial intelligence (AI), hypertext, and geographical information systems and computer aided design (CAD).

Many of these applications extensively navigate through the data. Some applications, such as CAD, process the data by complex and time consuming algorithms, and demand very high performance. For this, the data needs to be represented by data structures which can be accessed very fast. Caching of data close to the applications is particularly important when applications/tools run on autonomous workstations, with remote access to integrated data repositories. For example, design applications deal with large amount of data, which is mostly organized using such modeling concepts as version, alternative, and configuration. Those applications often work on a well-specified set of data, called *working set*, such as a particular version of a document or a wing of an aircraft for a particular model and version. Such data is typically an aggregation of a set of other documents/designs with specified versions. Usually working sets are extracted from the database and loaded into main memory close to the applications for high performance. After an application completes its work on the working set, the DBMS propagates back the changes to the originating databases. Working sets are typically much smaller than the whole database. For example, in design applications, the sizes of the databases are in the gigabytes to terabytes range, whereas working sets are typically in the range of 1 to 100 megabytes. Thus, loading a working set translates into a data extraction where on average one tuple out of 10000 to 100000 is selected. This again calls for set-oriented query facilities for efficient data extraction, requiring powerful optimization, and high performance execution. The concept of views is extensively used in DBMSs, allowing different applications to view the data differently. Applications dealing with composite objects require this concept to be extended to allow the formation of different views over the same data. Further, these views must be updatable.

We propose an extension of SQL, called SQL Extended Normal Form (or XNF) to satisfy the needs of the applications discussed above. XNF is based on the concept of Composite Objects (CO) as a collection of tables and relationships; COs provide an abstraction mechanism that unifies both the structural view as well as the tabular view of the data. We fully support this extension in the implementation, including query optimization and execution, and concurrency control and recovery.

With this approach the users benefit from the power of relational DBMSs to handle the tabular data, and benefit from the XNF extensions as well. Customers would also like to migrate their existing applications to exploit CO features. This calls for an integrated DB, which handles both the tabular as well as the CO data. Further, the commercial RDBMSs like IBM DB2 [IBM88] and Tandem Non-Stop SQL [BP88] have many industrial-strength features, that have taken years to build, and are vital to the users. These relate to the robustness of the systems, failure tolerance, high performance for SQL accesses and utilities, tools for monitoring performance, application development tools, good integration with the operating systems' features, a variety of storage management options, bulk I/O capabilities, exploitation of multiprocessors and, possibly, intra-transaction parallelism, different degrees of isolation (repeatable read, cursor stability etc.), query optimization, etc. It would

*University of Kaiserslautern, Dept. of Computer Science, 6750 Kaiserslautern, Federal Republic of Germany, e-mail: mitsch@informatik.uni-kl.de
** IBM Heidelberg Scientific Center, 6900 Heidelberg, Federal Republic of Germany, e-mail: {pistor, suedkamp} @dhdibm1.bitnet

be prudent to reuse these features and employ them as a foundation for extensions.

The remainder of the paper is organized as follows. The concept of composite objects is outlined and motivated in chapter 2. Chapter 3 discusses the major XNF query facilities to manipulate COs and to establish the mapping from relations to COs. Chapter 4 explains the implementation of XNF in Starburst DBMS and it shows that these extensions can be accommodated inexpensively and in such a way that the performance of important classes of data base applications can be considerably improved. Chapter 5 discusses related work, and the final chapter gives some conclusions and an outlook on future work.

2. Composite Object Abstraction in XNF

In XNF a Composite Object is defined as a collection of named component tables and relationships defined between these tables. We use the notion of Composite Object (CO) in order to emphasize that a CO is composed of multiple interrelated components. XNF's notion of CO is based on the view paradigm defining *CO views* (also called object views or structured views) composed from an underlying relational database. This means that the component tables and the relationships that are part of a CO definition have to be constructed from the tuples stored in a relational database. Based on the terminlogy of [LW90], the *COs have to be instantiated from relations by evaluating XNF view queries*. Hence different tools and applications may ask for different (not necessarily disjoint) COs over the same common database. This level of CO views achieves what we will call *CO abstraction* and the gap between relational data and CO abstraction is bridged by XNF's mapping facilities to be introduced here and in the next chapter.

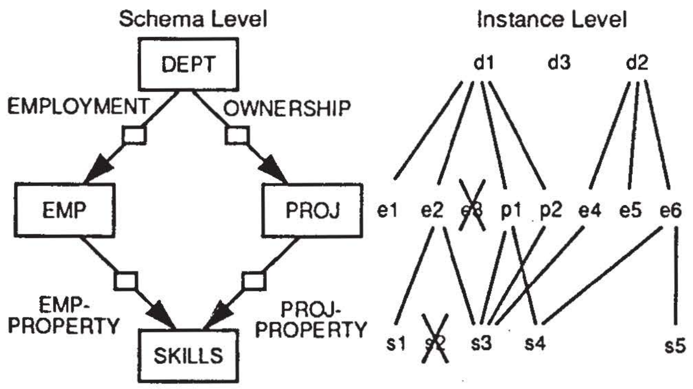


Figure 1: Sample Composite Object 'Company Organizational Unit'. Left Side: Schema graph for the CO 'Company Organizational Unit'. Right Side: Instance level showing component tuples linked via connections

As an introductory example Fig.1shows a sample CO taken from a simplified application scenario. For our purpose a simple business application is sufficient to express the basic concepts of our CO model; it will become obvious how these concepts carry over to non-traditional application areas. The sample company database contains the department (DEPT), employee (EMP), project (PROJ), and skill (SKILLS) tables. The CO depicted in Fig.1 gives for each department the associated employees (via the relationship EMPLOYMENT), the projects it owns (via the relationship OWNERSHIP), and finally the skills that either one of its employees possesses (via the relationship EMPPROPERTY) or one of its projects needs (via the relationship PROJPROPERTY). The left side of Fig.1 shows the schema information associated with this CO. The right side of this figure shows the tuple instances (d1, d2, ..., e1, e2, ...) and the connection instances (drawn as straight lines) relating their partner tuples. This CO represents what we might call the 'company organizational unit' that might constitute the working set for a particular business application.

For a CO to be well-formed, we require that the tables associated by a CO's component relationship must all be component tables of that very CO. This constraint carries over from relationships to their connection instances and from tables to their tuple instances: If one of the tuples is excluded from a composite object, the connections it is involved in have to be excluded, too.

XNF relationships are directed from a distinguished partner table ("parent" of that relationship) to the other partner tables ("child" tables). For example, the relationship EMPLOYMENT connects the partner table DEPT as a parent table to the child table EMP. In a general setting we allow for n-ary relationships, i.e. relationships that relate more than two partner tables. Since properties of binary relationships carry over to n-ary relationships, we will stay with binary ones for the scope of this paper. XNF relationships may be cyclic (e.g. "manages" relationship, with partner table EMP both as parent (in the "manager" role) and as child (in the "reports-to" role)). A component table of a CO is called *root table* if it has no incoming relationships, i.e. this table is not a child table of any relationship. In Fig.1 DEPT is the root table. At the instance level (right part of Fig.1) connections are represented by simple lines rather than arrows in order to indicate that relationships may be traversed in either direction independently of the relationship's direction as indicated in the schema part on the left side of Fig.1.

Direction in relationships is relevant only for determining the instance level of a CO which is defined by the *reachability* concept. Consider two tuples "a" and "b" of a composite object. "b" is said to be *reachable* from "a" if there exists a sequence of connections which, when traversed in parent to child direction, gets one from "a" to "b". The notion of reachability specifies the permissible instances of a CO: Any tuple of a CO must either be part of a root table, or it must be reachable from some tuple in a root table. This so-called reachability constraint restricts the components of a CO to only reachable ones, thus defining a quite natural notion of 'importance or relevance' of a component w.r.t. its CO. Referring to Fig.1 we can see that the tuples e3 and s2 do not fulfil the reachability constraint because they are not reachable through a tuple from a root table. Hence those tuples do not participate in that CO. On the other side, department d3, being a tuple from a root table, is reachable by definition, thus belonging to that CO. As we will see, the reachability concept will also simplify the specification of restrictions in composite object definitions (see section 3.1).

Relationships will provide a number of generic services, like:

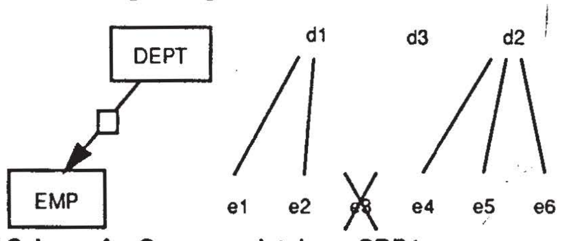
- Testing whether some tuples are related via a specific relationship;
- Denoting the tuples which are related to a given tuple or set of tuples (traversal and 'path expressions' in section 3.5; recall that traversal can be done in the direction of parent to child and vice versa);

Tables and relationships form the nodes and edges of a directed graph (left part of Fig.1)[1]. Based on this *schema* *graph* we define the notions of recursive COs, schema sharing, and instance sharing as follows:

- A CO is called *recursive*, if its schema graph contains cycles. Otherwise it is called *non-recursive*.
- A CO exhibits *schema sharing*, if at least one node has two incoming edges. For example the schema graph depicted in Fig.1 shows the schema-shared node SKILLS.
- Schema sharing is usually accompanied by *instance sharing*. For example in Fig.1 skill s3 is shared by employees e2 and e4 as well as by projects p1 and p2. Note however that schema sharing is not a prerequisite for instance sharing: a single relationship, e.g. EMPPROPERTY, might be sufficient as shown in Fig.1, again, by skill s3, which is shared between employee e2 and e4.

These structural concepts of sharing and recursion nicely match the four types of composing "molecular objects" based on two independent attributes [BB84]: disjoint (no sharing) and non-disjoint; recursive and non-recursive. Note that relationships are important in traversal and querying of COs (see Chapter 3), and in representation of data for efficient application access.

So far, we haven't made any assumptions on how we construct the COs from a relational database. Clearly, our notion of CO should be, as much as possible, independent of the conceivable representations of COs in the underlying database. For example, Fig.2 shows two different database



Partial Schema for Company database CDB1:
DEPT (dno, ..., budget, dmgrno)
EMP (eno, ..., salary, edno, epno)

Partial Schema for Company database CDB2:
DEPT (...)
EMP (...)
DEPTEMP (dedno, deeno, ...)

Figure 2: Two Different Representations for the Company Database (only partially shown)

representations for the information of relationship EMPLOYMENT that associates the departments to their employees (and vice versa). In company database CDB1 an implicit representation has been chosen (which is the usual representation for 1:n, i.e. functional associations) for that end, whereas in company database CDB2 a particular table (that, for example, might hold some attributes describing that association) has been used in order to reach at an explicit representation for the EMPLOYMENT relationship. However, independent from these two different representations, we want to see the two partner tables DEPT and EMP associated by the relationship EMPLOYMENT as is shown in the upper part of Fig.2 (this is a partial view of the CO depicted in Fig.1). In order to derive that information, we have to apply different queries depending on the particular representation, e.g. the relationship EMPLOYMENT might be derived in database CDB1 by joining the tables DEPT and EMP, whereas in CDB2 the particular table DEPTEMP might be sufficient for that end. This derivation is similar to the classical view concept specifying the schema of the target table and how that table is populated. With views, only the derivation needs to take into account the underlying representation of the source data, whereas the result table abstracts from the underlying database. In a similar way, XNF provides view facilities to derive the components of a CO populated by its connection instances and tuple instances. In the next chapter we elaborate on XNF's language approach for derivation and querying of COs.

1. From now on nodes and edges are used as synonyms for component tables and relationships, respectively.

3. The XNF Language

Since COs consist of tables and relationships, a CO constructor will be made up of table constructors and relationship constructors. SQL already provides a variety of facilities for constructing the component tables (e.g. table derivation using the SELECT..FROM..WHERE construct); therefore the only missing facility is a relationship constructor. Relationships will be specified using predicates, identifying tuples which are associated; if needed, relationships may have attributes to further characterize the connections between tuples. For now suppose that we are working with the company database CDB1 partly shown in Fig.2.

3.1 Introductory Query Example

An example of XNF's CO constructor is as follows:

```
OUT OF
  Xdept  AS   (SELECT * FROM DEPT WHERE loc='NY'),
  Xemp   AS   (SELECT * FROM EMP),
  Xproj  AS   (SELECT * FROM PROJ),
  employment  AS  (RELATE  Xdept, Xemp
                   WHERE   Xdept.dno=Xemp.edno),
  ownership   AS  (RELATE  Xdept, Xproj
                   WHERE   Xdept.dno =Xproj.pdno)
TAKE *
```

Starting from tables DEPT, EMP, and PROJ, this example constructs a CO with nodes Xdept, Xemp, and Xproj, and relationships employment and ownership. The query expression populates Xdept from DEPT. Similarly Xemp and Xproj are populated from tables EMP and PROJ, respectively. Unlike Xdept, tuples of Xemp and Xproj reuse the EMP and PROJ tables unchanged. The following short notation can be used for this purpose: Xemp AS EMP.

In contrast to SQL's query constructor, the CO constructor does not perform any concatenation and cartesian product of the participating components. The schema graph for this query is similar to the one shown in Fig.2.

The relationships employment and ownership are defined by the RELATE clause. This clause firstly gives the parent table and then the child table, and the WHERE clause takes the predicate that specifies the criteria for relating two partner tuples via a connection instance. Note that due to reachability no tuple from EMP (PROJ) is to be included into Xemp (Xproj) which cannot be reached from a New York department via the 'employment' ('ownership') relationship (reachability constraint). XNF's CO constructor provides further facilities, as will be shown in the subsequent sections.

3.2 Composite Object Views

Similar to SQL, an XNF query definition can be bound to a view name:

```
CREATE VIEW ALL-DEPS AS
OUT OF Xdept AS DEPT, Xemp AS EMP, Xproj AS PROJ,
  employment AS (RELATE  Xdept, Xemp
                 WHERE   Xdept.dno = Xemp.edno),
  ownership AS   (RELATE Xdept, Xproj
                 WHERE   Xdept.dno = Xproj.pdno)
TAKE *
```

Like tabular views of SQL, XNF views are important for data abstraction (see Chapter 2). Once defined, the user need not care about the way a composite object is ultimately assembled from tables of a relational database.

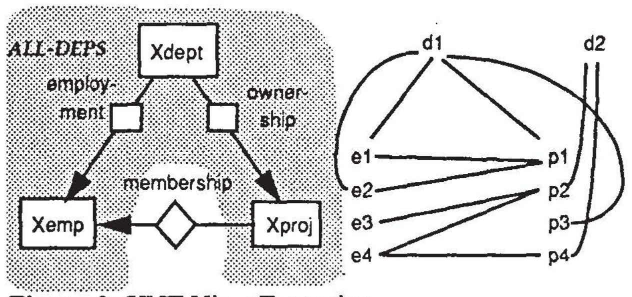


Figure 3: XNF View Extension

One important capability of the view concept that is only implicitly mentioned above is to build views over views, thus reaching at layered levels of abstraction. For example, we can define the view ALL-DEPS-ORG based on the view ALL-DEPS defined above (see Fig. 3 shaded part):

```
CREATE VIEW ALL-DEPS-ORG AS
OUT OF ALL-DEPS,
  membership AS (RELATE Xproj, Xemp
                 WITH ATTRIBUTES ep.percentage
                 USING   EMPPROJ ep
                 WHERE Xproj.pno = ep.eppno AND
                            Xemp.eno = ep.epeno)
TAKE *
```

This view definition takes all the components of the referenced view ALL-DEPS and extends them by an additional relationship 'membership' defined between the parent table Xproj and the child table Xemp. Both partner tables of that relationship are component tables of the XNF view ALL-DEPS. Different from the previous examples, this relationship has an attribute defined in the WITH ATTRIBUTES clause. The information from which the relationship has to be derived is given by the partner tables and by an additional table, the base table EMPPROJ, specified in the USING clause. Therefore, the relationship constructor relates all three tables in order to derive and construct the relationship component. In this example the relationship attribute is simply taken from the base table EMPPROJ, although it is possible to define an attribute using any (column) expression. Once defined, however, this relationship can be used in the same way as the other ones, without any knowledge of how it is actually constructed. Due to the newly added relationship in the schema graph, also new tuples and connections might show up at the instance level. For example in Fig.3 employees e3 and e4 are now considered, because they become reachable via the newly added relationship 'membership'.

So far, we have seen how views can be assembled from simple tables or other XNF views. In addition, we need facilities for removing unwanted parts from existing views. The following sections show how the query facilities can be extended for this purpose.

3.3 Node and Edge Restriction, Structural Projection

Clearly, SQL extensions for CO support also require facilities for querying COs. In the XNF approach, the CO constructor itself is used for that end, since it already provides the appropriate selection facilities. In essence, we follow the same idea as SQL, where the SELECT..FROM..WHERE construct is used in queries as well as in view constructors. For example, assume that we want the ALL-DEPS, but only those employees making less than 2K. This is achieved by

```
OUT OF ALL-DEPS
WHERE Xemp e SUCH THAT e.sal < '2K'
TAKE *
```

As already done in the previous two view definitions, the OUT OF clause is not used here to assemble the pieces from scratch; instead it refers directly to a predefined view. Like the view ALL-DEPS, this query deals with the nodes Xdept, Xemp, and Xproj, and the edges 'employment' and 'ownership'. Different from ALL-DEPS it will not contain an Xemp tuple, where the salary is 2K or more (*node restriction*), and no corresponding 'employment' connection either.

If we want to restrict the employees of the ALL-DEPS view to those who make less than 1 percent of their department's budget, we can best do this by imposing a restriction on the 'employment' relationship (*edge restriction*):

```
OUT OF ALL-DEPS
WHERE employment (d, e)
          SUCH THAT e.sal < d.budget/100
TAKE *
```

Here (d, e) denotes a connection instance of 'employment' symbolized by a pair of associated Xdept and Xemp tuples. This link item is to be discarded from ALL-DEPS, if the corresponding Xemp and Xdept tuples do not meet the specified predicate. Due to reachability, the Xemp tuple itself is also discarded (but not the corresponding Xdept tuple).

In addition to specific tuples or connections, complete edges and nodes might be removed, too, by projection capabilities. If we are not interested in the Xproj node, the previous query is modified as follows:

```
OUT OF ALL-DEPS
WHERE employment (d, e) SUCH THAT e.sal < '2K'
TAKE   Xdept(*), Xemp(*), employment
```

Since the Xproj node is gone, the 'ownership' relationship is discarded implicitly due to the well-formedness of COs.

3.4 Recursive Composite Objects

Let's take the view ALL-DEPS-ORG and extend it by an additional relationship 'projmanagement' that relates employees to the projects they manage (see Fig. 4):

```
CREATE VIEW EXT-ALL-DEPS-ORG AS
OUT OF ALL-DEPS-ORG,
  projmanagement AS (RELATEXemp, Xproj
                           WHERE Xemp.eno = Xproj.pmgrno)
TAKE *
```

Here, the relationships 'membership' and 'projmanagement' define a cycle on the schema graph. Thus EXT-ALL-DEPS-ORG is a structurally recursive CO. The instance level shows the tuples and connections. Connections of type 'projmanagement' are drawn as dotted lines. For example

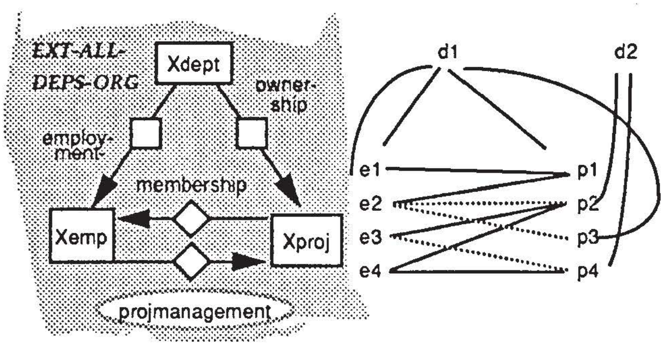


Figure 4: Recursive Composite Object

we can see that employee e2 manages the projects p2 and p3, and employee e3 works on project p2 and manages at the same time project p4, on which employee e4 (who also participates in project p2) works as well.

Reachability is instrumental in querying this type of CO. For instance, we can easily restrict EXT-ALL-DEPS-ORG to projects, which report - either directly or indirectly - to employees of New York departments. In order to do this, we employ a qualification criterion onto the Xdept table as well as a projection that excludes the 'ownership' relationship:

```
OUT OF EXT-ALL-DEPS-ORG
WHERE Xdept SUCH THAT loc = 'NY'
TAKE   Xdept(*), employment, Xemp(*),
       projmanagement, membership(*), Xproj(*)
```

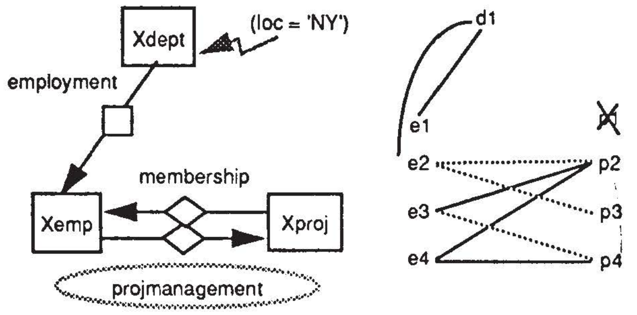


Figure 5: Restriction on Recursive CO

The result of this query applied to the XNF view EXT-ALL-DEPS-ORG shown in Fig.4 is visualized in Fig.5. Due to reachability, the result contains all employees of New York departments (e1 and e2), all projects managed by these employees (p2 and p3), the employees working on these projects (e3 and e4), and so on. Project p1 is not in the result, since it is not reachable anymore.

3.5 Path Expressions

A path expression is a convenient way of addressing parts of a CO node in a navigational style. They exploit the CO structure defined by the component tables and the relationships. Path expressions come in different forms. E.g., if d denotes a specific department in EXT-ALL-DEPS-ORG, then

```
d->employment->Xemp->projmanagement->Xproj
```

or the syntactically reduced path expression

```
d->employment->projmanagement
```

denote the set of all projects which are managed by employees employed by department d. We can specify predicates in path expressions, e.g.

```
d->employment->(Xemp e WHERE e.sal < 2K)->
                         projmanagement->Xproj
```

denotes the projects whose managers make less than 2K and are employed by department d. This example shows a so-called *qualified path expression.*

The following path expression defined for the XNF view EXT-ALL-DEPS-ORG is different from the first one:

```
Xdept ->employment->Xemp->projmanagement -> Xproj
```

It denotes all the projects in that view that are related via the relationships 'employment' and 'ownership' to any department of that view. Clearly, this set is a subset of the set of tuples in the component table Xproj.

In general, a path expression denotes a subset of the set of tuples of its target table. All these tuples are reachable from some root tuples through the path defined and in the case of qualified path expressions, the given predicates must be satisfied by the tuples on the path that leads to the leaf tuples. Hence, we view a path expression to be a table, to which we can apply any table operation, for example restrictions or counting as shown below. The direction in which a relationship has to be traversed is usually inferable from the sequence in which the connected nodes are specified. In specific cases (e.g. cyclic relationships mentioned in chapter 2) role names have to be used to avoid ambiguities.

The following query shows the usage of path expressions for querying XNF views. It uses the path expression defined above in order to address all the projects that are managed by the employees employed by a given department d:

```
OUT OFEXT-ALL-DEPS-ORG
WHERE Xdept d SUCH THAT
        COUNT(d->employment->projmanagement) > 2
        AND d.budget > '1000K'
TAKE *
```

This query restricts the departments (and implicitly via reachability also the employees and projects) of the XNF view EXT-ALL-DEPS-ORG to only those departments where in addition to the budget criterion there must be at least 2 projects related via the relationships 'employment' and 'ownership'. If we ask in another query for the departments that manage through some of its staff employees at least one project, whose budget is greater than the departments budget, then it is easier to use qualified path expressions:

```
OUT OFEXT-ALL-DEPS-ORG
WHERE Xdept d SUCH THAT
      (EXISTS  d->employment->
               (Xemp e WHERE e.descr='staff')->
               projmanagement->
               (Xproj p WHERE p.budget > d.budget))
TAKE *
```

3.6 Closure Property

Closure property gives the advantage of using the same query language on base data as well as on derived data or query results. As depicted in Fig.6 the closure property holds for XNF w.r.t. XNF operations and Normal Form (NF) SQL operations.

The following classification scheme for XNF queries is based on the closure property of XNF and on its compatibility to NF SQL. Fig.6 shows the four types of conceivable XNF queries.

The XNF approach covers the whole spectrum, from SQL's 1-table-result queries (so-called NF queries that incorporate regular tables and produce a single regular result table) to XNF's multi-table-structured result queries. Due to space limitations, we have concentrated on type (1) and (2)

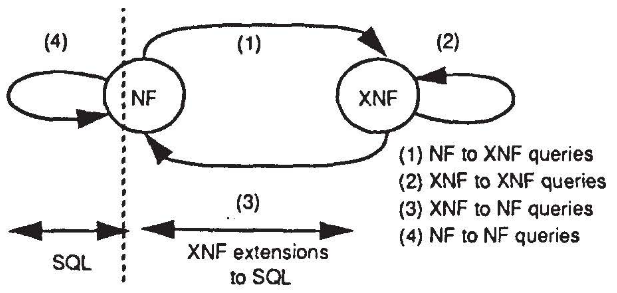


Figure 6: Classification Scheme for XNF Queries

queries (Fig.6) that emphasize on XNF's CO constructor. Type (1) queries define COs through a query that takes n input tables and produces m output tables and, optionally, some relationships that constitute the resulting CO. Type (2) queries build upon XNF views taking their result CO as input and producing another CO as output.

3.7 API Considerations and Manipulation Operations

For further processing, the CO denoted by an XNF query is optionally transferred into a high performance application program cache. While being represented (see section 4.2) by pointer structures, the cache is not exposed to the user at that implementation level. Instead the nodes of a cached CO may be accessed through cursors only. XNF provides two kinds of cursors. If an *independent* cursor is opened on a node, it allows one to browse through all its tuples. *Dependent* cursors are bound to other cursors through a path expression. E.g., if there is an open cursor aDept on the node Xdept (see Fig.3), a cursor anEmpOfDept may be opened on Xemp which depends on aDept through the relationship 'employment'. Different from an independent cursor, opening of this cursor gives only access to those employee tuples which are reachable from the department the cursor aDept currently points to. As we have seen so far, relationships are very useful: firstly, they are required for the specification of path expressions that are then used for cursor definition, and secondly, they are used for building efficient in-memory data structures (see section 4.2) that directly support navigation via cursor operations.

XNF provides operations for changing the cache contents through update, delete, and insert operations on tuples (called udi-operations) as well as connect and disconnect operations on relationships. All udi-operations on XNF component tuples as well as connect and disconnect on relationships are propagated to the corresponding base tuples. The cache is maintained in such a way that cache changes can be propagated in an efficient fashion [KDG87].

In addition to cursor manipulation, XNF supports modification operations. Update, delete, and insert are available at the CO level. For example, a CO deletion statement specifies the removal of all tuples and connections of that target CO. This removal of CO components maps down to removals of the base tuples from which the CO components are derived. For the following CO deletion statement all the department, project, and employee tuples that map to component tuples and relationships of the target CO have to be removed from their base tables.

```
OUT OF  ALL-DEPS
WHERE  Xemp e SUCH THAT e.sal < '2K'
DELETE *
```

In all cases, the CO to be manipulated must be updatable. Since XNF relies on views, we first review the view updatability in relational databases, then we discuss updating of XNF views.

In the relational systems, base tables are updatable. A view over a base table which hides some columns of a base table is equally updatable. We can restrict the tuples of the view by specifying a predicate referring to the columns of the table the view is defined on. This view is also updatable. Hence, use of views does not compromise updability of the database. Such updatable views are commonly used in practice, particularly for authorization. The relational model allows users to specify even more powerful views, hence not restricting the view specification to udpatable views only. For example, views may contain aggregation, joins, etc. In general, such views are read only since an update of a tuple of the view cannot be (reverse) mapped unambiguously to an update of the base data. Updatability of views is extended in systems such as Starburst [HCL90]. We directly benefit from such extensions.

We follow the same design philosophy in specification of XNF views. We need to address update of the relationships in addition to update of the nodes in XNF. In Fig.4, the 'employment' relationship connects Xdept tuples and Xemp tuples. We want to disconnect some relationship instances and connect new ones. Disconnecting an 'employment' relationship instance results in setting the dno of the tuple of Xemp associated with this relationship to the null value. Basically, if a relationship is defined by a foreign key, disconnect results in nullifying the foreign key. Connecting an Xemp tuple to an Xdept tuple results in setting the foreign key of the Xemp tuple. Consider the 'membership relationship', which is M:N. Each relationship instance is built from a tuple in the EMPPROJ base table. The disconnect operation results in deleting the corresponding tuple in the EMPPROJ table and the operation connect results in inserting a tuple in the EMPPROJ table. Following our view update philosophy, we do not restrict the definition of relationships to only the ones that are updatable. For example, users can define a relationship between average level of productivity and years of experience, but cannot update it.

The nodes of XNF are regular views, and as such, updatability rules of views apply to them. For instance, in Fig.4, one can update the salary of employees or the budget of departments. However, update of the dno column of Xemp is done only through the relationship connect/disconnect, as explained above. In general, columns that are used to define relationships are updated by relationship manipulation as explained above. Delete of an Xemp tuple results in disconnecting the associated employment relationship. This is to prevent any dangling relationships. Likewise, delete of an Xdept tuple results in disconnection of all the employment and ownership relationships instances attached to it. In general delete of a tuple can only result in delete of the tuple itself and all the relationships instances directly attached to it.

4. Design And Implementation

We considered two approaches to build this system: (1) build a new DBMS suitable for XNF, (2) adapt an existing DBMS. In either approach, the new system will need composite object data clustering for I/O reduction, fast extraction of data, and composite object query optimization.

Regarding clustering, relational DBMSs typically allow clustering of data along tables, which is inappropriate for composite objects, where we need clustering of component tuples belonging to different tables. However, existing RDBMSs apply already clustering techniques beyond naive table clustering. For example, DB2 [IBM88] clusters tuples of catalog data in the form of composite object clusters to minimize I/O overhead for catalog access. Starburst allows clustering of the parent and children of a relationship ([LLPS91], IMS attachment) to reduce I/O overhead of joins. We concluded that we can benefit considerably from the existing clustering technology in relational DBMSs. So far, this argues in favor of approach 2.

Another feature we need is fast extraction of data. This is particularly so in large engineering and design databases, e.g. in aerospace and automotive industry, where the size of the database could be in terabyte range (as discussed in the introduction). We concluded that XNF must be able to use the technology developed in the relational DBMSs to handle large amount of data and complex query processing. For instance, parallelism can reduce execution of XNF queries by orders of magnitude. Set oriented specification of composite objects in XNF particularly lends itself to exploitation of parallelism technology [DG90, Gr90, LD89, PMC90].

Specification of XNF views mostly reuses the relational query language (SQL in our case). Almost all of the optimization techniques developed in the context of relational DBMSs are applicable for COs as well. We discuss this in more detail shortly.

Operational relational DBMSs are very expensive to build. It quickly became apparent to us that both from technical and economic viewpoints, approach 1, building a new system, did not make sense. Rather, by adapting an existing DBMS, we could build a much more powerful system in a much shorter time. We chose Starburst DBMS [HCL90] as the starting point. Starburst was particularly attractive due to its extensibility features.

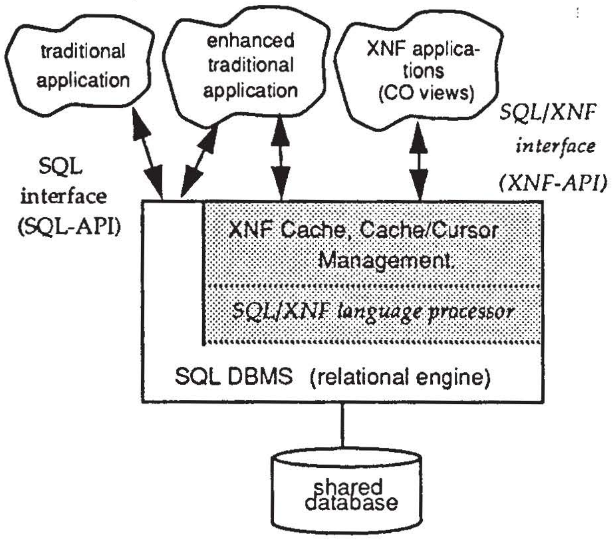

traditional application
enhanced traditional application
XNF applications (CO views)
SQL/XNF interface (XNF-API)
SQL interface (SQL-API)
XNF Cache, Cache/Cursor Management
SQL/XNF language processor
SQL DBMS (relational engine)
shared database

Figure 7: General Architecture of the SQL/XNF Language Processor and Application

In retrospect, our intuition/initial study proved correct. Our XNF extensions to Starburst cost much less than the whole system. We conjecture that this percentage is much lower if we extend a commercial DBMS, capable of handling large databases, such as IBM DB2 [IBM88], Tandem Non-Stop SQL [BP88], or Teradata [Ne86].

4.1 System Architecture

As shown in Fig. 7, SQL/XNF composes a composite object from the relational database. The data is shared between traditional SQL applications and XNF applications. Note that no change is required in the traditional applications to access this shared database. The XNF applications use the (fast) XNF Application Language Interface (API) to access the data. SQL/XNF language processor translates the XNF queries to a form very close to the standard SQL, allowing reuse of the DBMS with little change. This translation is performed at compile time, and is optimized, eliminating any runtime overhead. We should emphasize that XNF is integrated into the RDBMS (not an on-top solution), and extensively reuses the query processing components of RDBMS as explained below.

4.2 XNF Cache and API

Browsing through the XNF structured data is very similar to that of OO languages [BTA90], which is based on collection enumerators and path expressions. An application opens a cursor on a node of an XNF structure, and enumerates the tuples of that node. The application can cross a relationship from one node to another node given a cursor on the first node by opening a dependent cursor specified by a path expression that connects the two nodes. XNF cursors over the cache are very fast. We have made an effort to cut down on the pathlength of cursor operations. Particularly, unlike regular SQL, the access to the cache does not require any inter-process communication. The XNF cache uses virtual memory pointers to link the tuples of an XNF structure. As a result, the browsing is very fast. The structure of the XNF cache and the algorithms used for loading and navigating the cache are not discussed further due to space limitation.

In addition to the cursor interface, language compilers (such as those for object-oriented languages) can interface with XNF cache internally to store and browse through the data, and provide a language specific interface to the applications. For this scenario it is quite obvious that object-oriented programming environments (e.g. C++) are very useful in order to achieve a proper adaptation to the applications' needs. For instance, browsing can be efficiently accomplished by pointer dereferencing through XNF cache and flexible manipulation support is due to the concepts of methods, encapsulation as well as inheritance and overriding.

4.3 Compilation and Execution of XNF Queries

First we give a brief overview of the Starburst internal components. Then we explain the new components and the changes we made to these components to support XNF.

Fig. 8 shows phases of query compilation (see [HCL90] for more detail). Such structuring of phases of query compilation can also be found in other DBMSs, such as [CM88]. During parsing and semantic checking, a Starburst SQL statement is translated to an internal representation, called Query Graph Model (QGM). The query rewrite phase transforms the QGM representation of queries to equivalent ones for better performance. Examples of such transformations are merging of views with queries, predicate pushdown, and magic sets [MFPR90]. The plan optimizer examines alternative plans, such as use of indices and various join methods, for a query represented in QGM and chooses one with a lower cost. Query refinement follows the plan generated by the plan optimizer and produces an executable plan which is run by the query evaluator at runtime.

In Fig. 8, XNF extensions are represented as shaded areas. First we needed to modify the grammar to understand XNF language constructs, and added the appropriate semantic checking routines. XNF queries are mostly composed of existing SQL language constructs. This lends itself to significant reuse of the semantic routines of the existing SQL constructs, avoiding expensive duplication of the code. This considerably simplifies our implementation of the semantic checking. Queries are represented in the QGM internally. Thus, we needed to enhance QGM representation to handle XNF queries. QGM is based on the notion of table abstraction. Queries are represented as a series of high level operators (e.g. SELECT, GROUP BY, UNION, and INTERSECTION) on either base tables or derived tables. An operator consists of a head and a body: the head describes the output table and the body shows how this table is derived from other tables the body refers to (e.g. performing projection on the join result of the input tables according to a given predicate). We added an XNF operator to QGM to capture the semantics of the XNF CO constructor in the language. This XNF operator is able to produce m>=1 output tables, each of them representing either a node or an edge. The model allows arbitrary nesting and mixing of XNF constructors and SQL constructors, maintaining the closure property.

We introduced a new step, XNF semantic rewrite, which translates XNF operators in QGM to regular SQL operators (Fig. 8). The significance of this is that it enables us to reuse the expensive optimization and evaluation machinery of the relational DBMS. We give just a summary of this translation due to space limitation. Basically, we formulate one query for each node or relationship output of an XNF query, observing XNF semantics such as reachability. These queries typically use common subqueries to avoid unnecessary redundant computations. For instance, when we generate the tuples of a parent node, we output them, and also use them again to find the tuples of the associated children. Regular output processing of SQL is modified to allow generation of a heterogeneous set of tuples in the answer set (generation of tuples belonging to different nodes and relationships). Due to the power of the Starburst query rewrite component, we were able to go for straightforward transformations from XNF to SQL QGM operators. Any optimization of the resulting QGM can be deferred to the query rewrite step, which takes care of merging query blocks or other simplifications and clean-up operations.

Processing of XNF does not require any change to query rewrite and no significant change is required in the plan optimization. In the plan optimizer handling of joins is heavily used since parent child relationships are computed by joins. The plan optimizer should take into account any parent/child links present in the database ([LLPS91], IMS attachment), and clustering of data on disk for I/O and pathlength reduction in optimization of joins.

The query refinement is modified to deliver heterogeneous sets of tuples to the cache manager efficiently. When a parent tuple is accessed, it is sent directly to the output, and it is used to compute the children. The answer tuples are sent out to the cache manager once they are computed, avoiding further buffering since the cache manager buffers them anyway.

Again, note that we only needed to make comparatively small extensions to the RDBMS compiler. All the other

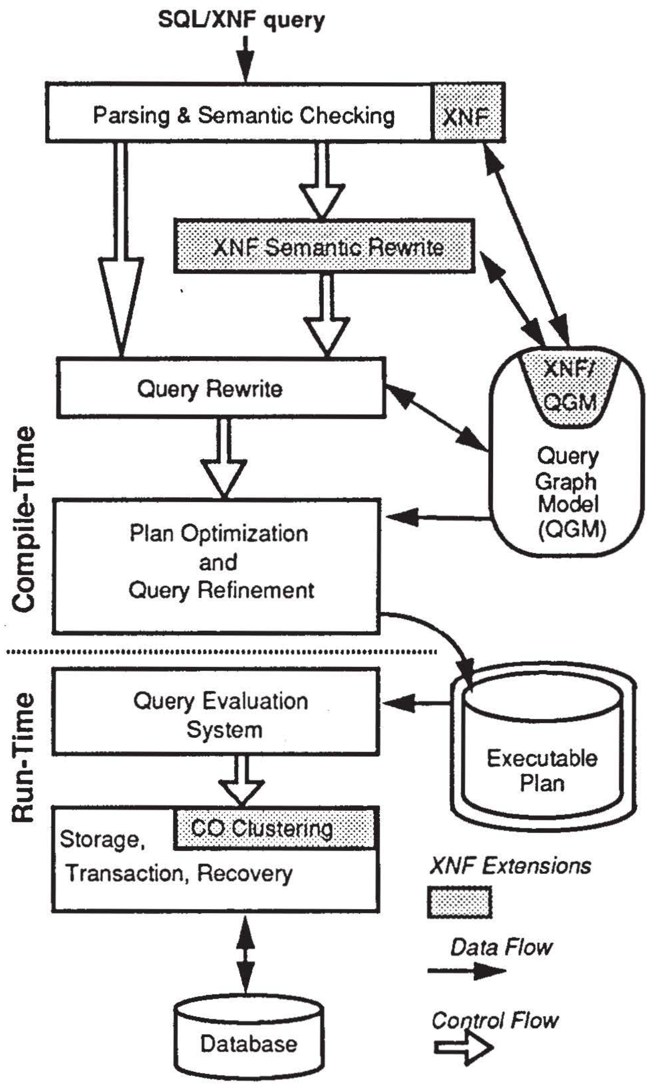


Figure 8: Stages of XNF Query Processing

components, notably the query evaluation system, transaction, recovery and storage management, are completely shared between XNF and regular DBMS users. Yet the result is a much more powerful query capability, combined with high performance, satisfying the needs of the demanding applications, e.g., in design and engineering areas.

Loading of the cache benefits from the power of XNF queries. The scope of query optimization encompasses all the queries that define the nodes and relationships of an XNF structure. The optimizer is able to take advantage of common subexpression across these queries. Further, the answer to all these queries are combined. This allows the DBMS to more efficiently block the heterogeneous answer tuples for sending between the DBMS and application processes (which may reside in different processors across a network).

The browsing of the data in the cache using cursors is very fast due to main memory pointers between tuples in the cache. The performance improvement over regular SQL DBMS interface is in orders of magnitude, and is comparable to the performance improvement of OODBMS over relational DBMSs reported in Cattell's benchmark [Gr91]. However, XNF provides this high performance while preserving the ability to share data with other relational applications. Currently, we are in the process of completing performance measurements of XNF. These results will be reported in a separate publication. However, early results show that SQL/XNF meets the performance requirements of such applications as CAD/CAM where the performance of regular SQL has been shown to be inadequate [Gr91], whilst OODBMSs are considered to be more suitable.

5. Related Work

We focus on the main aspects of XNF: the power of the relationships, CO abstraction, and overall architecture and system design.

The relationship concept is particularly emphasized in the literature [AGO91, ASL89, ZM90] and in practice [RB91], and is heavily supported in practical OO systems (e.g., [So92, De91, LLOW91, On91, Ve91, Ob90]). In these systems, relationships are typically implemented with pointer sets on each side of the relationship, each set pointing to the objects on the other side of the relationship. The system keeps these pointer sets consistent. Typically, relationships are defined as part of the base objects on each side of the relationships. In contrast, XNF allows definition of viewed relationships, and incremental addition of relationships[1]. Further, XNF allows the viewed relationships to be defined as part of its DML. Powerful optimization capabilities of the DBMS equally apply to queries involving relationships. In addition, relationships in XNF can have attributes. For instance, in Fig.3, the membership relationship may have an attribute on percentage of the time an employee works on the associated project.

Let's go through some examples to illustrate these points. Using the structure of Fig.3, we want to define departments and employees, and an 'involve' relationship, which gives the employees who work at least half time on projects of a department, and 'employment' relationship which gives the employees employed by a department. This structure provides an abstract view of ALL_DEPS-ORG structure of Fig.3, hiding the Xproj component. Here, the 'involve' relationship is a concatenation of the 'ownership' and 'membership' relationships with a restriction on attribute percentage. Such a view can be defined declaratively in XNF. In contrast, in the OO systems cited above, one must write accessor functions (say in C++) to implement the 'involve' relationship. Often, this is not an acceptable solution, particularly for the end users that want to create such views on an existing operational system, where reprogramming the system is not an option. Declarative specification of the relationships in XNF also allows reuse of SQL optimizers. Such optimization is essential since it may lead to orders of magnitude improvement in performance, particulary in handling of path expressions [PHH92]. Further, it is not clear how the project information becomes hidden in the view since there is no construct equivalent to XNF's component/relationship projection. Both Xdept and Xemp have fields that refer to Xproj, and such fields must become hidden.

Further on, a view concept plays a key role in schema evolution. Suppose there is an application that works with the schema of Fig.3. Now a new application is installed that shares the same database but needs a slightly different view being a new relationship between Xemp and his/her medical records. OO systems usually require modifying Xemp to add this new relationship, and recompiling the applications that refer to Xemp (since the data structure of Xemp has changed!). Further, in some systems, the data in the database may have to be changed to add new pointer sets for the new relationship. But in most cases this is not practical. Particularly, a casual end user may create and use this relationship for a short period of time. This must not cause thousands of existing programs that are concurrently using the database to be recompiled! Often users have read only access to the data. Hence, data in the database cannot be updated to add pointer sets for the new relationship.

There are several efforts to extend the relational data model with a type system [ANS91, LLPS91]. Further, handling of typed views in the context of OO languages has been addressed in [HZ90,SLT91, SS91]. The results are analogous to traditional database views, except that the object views hide or expose methods as well as data. Careful use of the query language is the key to define updatable views. XNF is integrated with [LLPS91], which provides support for types and handling of (dependent) cursors. Since XNF uses queries as its constructors, extensions to the data model/query languages for handling of types are directly applicable to XNF. For example, nodes of XNF can have types: depttype, emptype, etc. Essentially, queries may have object generating semantics [SS91]. Therefore, tuples in an XNF result become typed objects. Further effort is underway in extension of the relational model to define types associated with relationships (references) [ANS91, LLPS9, ADL91]. The notion of path expressions has also been introduced in [BTA90, LLOW91, On91, Ve91, KKS92]. One major difference is that XNF path expressions are very close to SQL subqueries, and preserve semantics of SQL, including null values and duplicates. As discussed before, XNF path expressions return tables, and can be used in place of any table reference in queries. As a result, richness of XNF path expressions is achieved with little additional constructs and semantics.

In contrast to our integrated approach, [BW89, LW90] suggest an on-top approach. This approach integrates an object-oriented program with databases through instantiation of objects from relational databases by evaluation of view queries. The system model applied has three elements: the object type model that defines the structure of the objects, the relational data model for storage of base data, and the view model that contains the relational query and that defines a mapping between objects and relations. That view model is restricted to only acyclic select-project-join queries. Basically this approach is comparable to XNF but major differences are obvious. Firstly, XNF realizes with its CO constructor a more powerful view concept (multi-table views), which, secondly, provides an abstraction level that considerably reduces the final mapping (if needed at all) to the application's favorable data structure. Thirdly, viewed from the other side, we can use XNF as another kind of view model within the system model of [LW90]. Hence, XNF can profit from the framework defined (i.e., the object type model and the corresponding compiler). With this, the approach of [LW90] gets extended due to an enhanced view model, and the implementation gets simplified due to XNF's integration of CO processing into relational DBMSs.

There are various other approaches to modelling and management of COs as extensions to the relational model. In [LK84] COs are defined by special columns (assigning an identifier to a tuple, containing the parent identifier, and

1. Base (materialized) relationships are part of XNF but not reported here due to space limitation.

referencing another tuple). Joins among parents and children are supported by system-maintained access paths (called maps) on a per-CO basis. Although this approach integrates CO processing into the relational framework, its usages are limited because of the restrictions of the data model to essentially hierarchical COs that are statically defined in the database schema. On the other hand, the Molecule Atom Data (MAD) model [Mi89] supports network-like as well as recursive COs. MAD specifies its COs (called molecules) on a reference basis in the CO/molecule query and not in the schema. With this, more flexibility is achieved, because COs are now similar to views defined over the underlying database by means of a CO query. Compared to the XNF approach, the MAD approach is less flexible, because the molecule building references must exist in the database, and therefore also in the schema; remember that the relationships in XNF can be defined on an ad-hoc basis through a predicate in the query. Query processing in MAD [HMS92] is also based on a set of operators that is different to the known relational operators due to the molecule semantics applied. For that reason, the MAD implementation does not fit smoothly into the relational query processing framework (see [HMS92]), thus restricting sharing of relational technology (and system code).

The nested relation approach [SS86], often referred to as NF2, provides more flexibility compared to Lorie's approach[LK84]. NF2 is implemented in several prototypes and extended in several ways [DK86, LK88, PA86, SPSW90, CD88]. NF2 is targeted towards hierarchical COs by generally placing child components with the parent component. In general, access to sub-components goes through the parent. Sharing of components between parents is done by listing of foreign keys (or logical references), which implies that access is done on a join basis as in relational systems. Flexibility is achieved through specific operations that can flatten out or restructure the nesting given in the database schema. Because of these model specific operations, the implementation reflects an extended relational engine.

6. Conclusions and Outlook

In this paper we have introduced the XNF approach that supports processing of Composite Objects as abstractions over relational data, thereby bridging the gap between relational stores and the (structured) data view of advanced applications. The XNF approach comprises the following major concepts:

- a data model that unifies CO and relational concepts,
- a common query language for handling both CO and simple relational data,
- an implementation approach that guarantees efficient data extraction, and
- an API with facilities for efficient navigation and manipulation.

XNF's Composite Objects are heterogeneous sets of interrelated objects. Since COs can be specified as views over XNF databases as well as traditional relational databases, data from a variety of sources can be presented to the applications at an appropriate abstraction level and in a format than can be processed with high performance. However, the language itself keeps the benefits of relational languages, i.e. declarative queries, set-orientation, powerful selection capabilities, and closure w.r.t. its operations.

Rather than implementing a new DBMS from scratch, an evolutionary approach was chosen, where we decided to base the implementation on existing relational technology and to integrate CO processing into the relational framework. XNF queries are translated to relational queries, optimized, and then executed by a relational engine. Our major goals were support for CO data clustering, fast data extraction, and CO query optimization. For that end, we needed only comparatively small changes to the RDBMS compiler; all the other components (query evaluation system, transaction, recovery, and storage management, among others) are kept unchanged.

The API marks an important contribution to efficiency. It supports generic cursors with facilities to cross a relationship from one component to another. The implementation of cursor operations has been tuned for fast cursor access and efficient browsing capabilities. Language compilers (e.g., from object-oriented programming environments, like C++) can easily interface with the XNF API in order to achieve a final adaptation to the applications' needs.

Currently, a major portion of SQL/XNF is operational in Starburst extensible database system at IBM Almaden Research Center. First measurements have shown a performance improvement in orders of magnitude over regular SQL DBMS that compares to the performance improvement of OODBMSs over relational DBMSs as reported in the Cattell Benchmark.

Further, XNF does not bind itself to only one kind of application language, rather it is open to different application environments. This is important since, in general, applications written in different languages share the data in the database. Further, we can view XNF as a high performance approach that provides a path for incorporating relational data into any CO application. For example we can use an XNF DBMS (i.e., an extended RDBMS) to provide server services to an OO programming system running on the application site (eventually in the application's address space on a dedicated workstation).

In the near future, we will collect more comprehensive performance data as the bases for analyzing and further enhancing performance. In addition, we're working on further improving API capabilities to ease working with COs. Finally, we will continue to improve XNF query processing exploiting CO cluster facilities, common subexpressions as well as concepts for parallel query processing.

Acknowledgments

We wish to thank our colleagues of the Starburst project providing the Starburst implementation environment. Further we would like to thank L. Haas, P. Selinger, S. Deßloch, R. Erbe, and K. Küspert for reading an earlier draft. R. Lorie contributed many comments and criticism that lead to a much improved language, G. Lohman improved the optimizer to handle our complex queries, and G. Wilson provided valuable implementation experiences on his work on an earlier prototype. Discussions with J. Ullman and I. Mumick lead to better understanding of semantics of XNF.

Bibliography

AB91 Abiteboul, S., Bonner, A.: Objects and Views, in: Proc. of the ACM SIGMOD Conf., Denver, 1991, pp. 238-247.

ADL91 Agrawal, R., DeMichiel, L., Lindsay, B.: Static Type Checking and Run-time Dispatch of Multi-Methods, in: Research Report, IBM Almaden Research Center, 1991.

AGO91 Albano, A., Ghelli, G., Orsini, R.: A Relationship Mechanism for a Strongly Typed Object-Oriented Database

Programming Language, in: Proc. 17th VLDB Conf., Barcelona, 1991, pp. 565-575.

ANS91 Melton, J. (ed.): Data Base Language SQL3, ISO/ANSI working draft, X3-92-001 DBL-KAW003b, Dec. 1991.

ASL89 Alashqur, A.M., Su, S.Y.W., Lam, H.: OQL: A Query Language for Manipulating Object-oriented Databases, in: Proc. 5th Int. Conf. on VLDB, Amsterdam, 1989, pp. 433-442.

BB84 Batory, D.S., Buchmann, A.P.: Molecular Objects, Abstract Data Types, and Data Models, in: Proc. 10th VLDB Conf., Singapore, 1984, pp. 172-184.

BP88 Borr, A., Putzolu F.: High Performance SQL Through Low-Level System Integration, in: Proc. of the ACM SIGMOD Conf., Chicago, 1988, pp. 342-349.

BTA90 Blakeley, J., Thompson C., Alashqur, A.: Strawman Reference Model for Object Query Language, in: Proc. of First OODB Standardization Workshop, X3/SPARC/DBSSG/OODBTG, 1990.

BW89 Barsalou, T., Wiederhold, G.: Knowledge-Based Mapping of Relations into Objects, in: Computer Aided Design, 1989.

CD88 Carey, M., DeWitt, D., Vandenberg, S.: A Data Model and Query Language for Exodus, in: Proc. of the ACM SIGMOD Conf., Chicago, 1988, pp. 413-423.

CM88 Chang, P. Myre, W.: OS/2 EE Database Manager overview and technical highlights, in: IBM Systems Journal, Vol. 27, No. 2, 1988, pp.105-118.

De91 Deux, O. et al: The O2 System, in: Communications of the ACM, Vol. 34, No. 10, 1991, pp. 35-48.

DK86 Dadam, P., Küspert, K., et al. : A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies, in: Proc. of the ACM SIGMOD Conf., Washington D.C., 1986, pp. 356-367.

DG90 DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H.-I., Rasmussen, R.: The Gamma Database Machine Project, in: Knowledge and Data Engineering, Vol. 2, No. 1, March 1990.

Gr90 Graefe, G.: Volcano, an Extensible and Parallel Query Evaluation System, Research Report, University of Colorado at Boulder, CU-CS-481-90, 1990.

Gr91 Gray, J. (ed.): The Benchmark Handbook for Database and Transaction Processing Systems, Morgan Kaufman Publ. Inc. 1991.

HCL90 Haas, L., Chang, W., Lohman, G., et al.: Starburst Mid-Flight: As the Dust Clears, in: Special Issue on Database Prototype Systems, IEEE Transactions on Knowledge and Data Engineering, Vol.2, No.1, 1990, pp. 143-160.

HMS92 Härder, T., Mitschang, B., Schöning, H.: Query Processing for Complex Objects, in: Data and Knowledge Engineering 7, 1992, pp. 181-200.

HZ90 Heiler, S., Zdonik, S.: Object Views: Extending the Vision, in: 6th Int. Conf. on Data Engineering, Los Angeles, 1990, pp.86-93.

IBM88 IBM Database 2 System and Database Administration Guide, IBM Publication Document No. SC26-4374, Dec. 1988.

KDG87 Küspert, K., Dadam, P., Günauer, J.: Cooperative Object Buffer Management in the Advanced Information Management Prototype, in: Proc. 13th VLDB Conf., Brighton, 1987, pp. 483-492.

KKS92 Kifer, M., Kim, W., Sagiv, Y.: Querying Object-Oriented Databases, in: Proc. of the ACM SIGMOD Conf., San Diego, 1992, pp. 393-402.

LD89 Lorie, R., Daudenarde, J., Hallmark, G., Stamos, J. Young, H.: Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience, Data Engineering, Vol. 12, No. 1, March 1989.

LK84 Lorie, R., Kim, W., et al. : Supporting Complex Objects in a Relational System for Engineering Databases, IBM Research Report, San Jose, CA, 1984.

LK88 Linnemann, V., Küspert, K.: Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions, in: Proc. of the 14th VLDB Conference, Los Angeles, California 1988.

LLOW91 Lamb, C., Landis, G., Orenstein, J., Weinreb, D.: The Objectstore Database System, in: Communications of the ACM, Vol. 34, No. 10, 1991, pp. 50-63.

LLPS91 Lohman, G., Lindsay, B., Pirahesh, H., Schiefer, B.: Extensions to Starburst: Objects, Types, Functions, and Rules, in: Communications of the ACM, Vol. 34, No. 10, 1991, pp. 78 - 94.

LW90 Lee, B.S., Wiederhold, G.: Outer Joins and Filters for Instantiating Objects from Relational Databases through Views, CIFE Technical Report, Stanford University, May 1990.

MFPR90 Mumick, I., Finkelstein, S., Pirahesh, H., Ramakrishnan, R.: Magic is Relevant, in: Proc. of the ACM SIGMOD Conf., Atlantic City, 1990, pp. 247-258.

Mi89 Mitschang, B.: Extending the Relational Algebra to Capture Complex Objects, in: Proc. 15th Int. VLDB Conf., Amsterdam, 1989, pp. 297-308.

Ne86 Neches, P.: The Anatomy of a Data Base Computer - Revisited, in: Proc. of COMPCON Conf., 1986.

Ob90 Objectivity, Inc.: Objectivity Database System Overview, Menlo Park, CA, 1990.

On91 Ontologic Inc: ONTOS Reference Manual, Burlington, Mass 1991.

PA86 Pistor, P., Andersen, F.: Designing a Generalized NF^2 Data Model with an SQL-type Language Interface, in: Proc. 12th Int. Conf on VLDB, 1986.

PHH92 Pirahesh, H., Hellerstein, J., Hasan, W.: Extensible/Rule Based Query Rewrite Optimization in Starburst, in: Proc. of the ACM SIGMOD Conf., San Diego., 1992, pp. 39-48.

PMC90 Pirahesh, H., Mohan, C., Cheng, J., Liu, T., Selinger, P.: Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches, in: Proc. of the Int. Symposium on Databases in Parallel and Distributed Systems, Dublin, 1990.

RB91 Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

So92 Soloviev, V.: An Overview of Three Commercial Object-Oriented Database Management Systems: ONTOS, ObjectStore, and O2, in: ACM SIGMOD Record, Vol. 21, No. 1, 1992, pp. 93-104.

SLT91 Scholl, M., Laasch, C., Tresch, M.: Updatable Views in Object-Oriented Databases, in: DOOD 2nd Int. Conf., Springer Verlag, LNCS No. 566, 1991, pp.189-207.

SPSW90 Schek, H.-J., Paul, H.-B., Scholl, M.H., Weikum, G.: The DASDBS Project: Objectives, Experiences, and Future Prospects, in: IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, 1990, pp. 25-43.

SS86 Schek, H.J., Scholl, M.H.: The Relational Model with Relation-Valued Attributes, in: Information Systems, Vol. 2, No. 2, 1986, pp. 137-147.

SS91 Scholl, M., Schek, H.: Supporting Views in Object-Oriented Databases, in: IEEE Database Engineering Quarterly Bulletin, June 1991.

Ve91 Versant Object Techn. Inc.: VERSANT Technical Overview, Menlo Park, CA, 1991.

ZM90 Zdonik, S., Maier, D.: Readings in Object-Oriented Database Systems, Morgan Kaufman Publ., 1990.