Pergamon

Information Systems Vol. 19, No. 1, pp. 69–88, 1994
Elsevier Science Ltd. Printed in Great Britain
0306-4379/94 $6.00 + 0.00

0306-4379(94)E0006-B

COMPOSITE-OBJECT VIEWS IN RELATIONAL DBMS: AN IMPLEMENTATION PERSPECTIVE

H. PIRAHESH,[1] B. MITSCHANG,[2] N. SÜDKAMP[3] and B. LINDSAY[1]

[1]IBM Almaden Research Center, San Jose, CA 95120, USA
[2]University of Kaiserslautern, Dept. of Computer Science, P.O.Box 3049, 67653 Kaiserslautern, Germany
[3]IBM Heidelberg Scientific Center, Vangerowstr. 18, 69115 Heidelberg, Germany

(Received 18 June 1993, in final revised form 28 October 1993)

Abstract — We present a novel approach for supporting Composite Objects (CO) as an abstraction over the relational data. This approach brings the advanced CO model to existing relational databases and applications, without requiring an expensive migration to other DBMSs which support CO. The concept of views in relational DBMSs (RDBMS) gives the basis for providing the CO abstraction. This model is strictly an extension to the relational model, and it is fully upward compatible with it. We present an overview of the data model. We put emphasis in this paper on showing how we have made the extensions to the architecture and implementation of an RDBMS (Starburst) to support this model. We show that such a major extension to the data model is in fact quite attractive both in terms of implementation cost and query performance. We introduce a CO cache for navigation through components of a CO. With this technique, the performance of navigation through COs, which has been of a concern in RDBMSs in the past, is in fact quite satisfactory. We present our practical experience in using this facility. We show that our work on CO enables existing RDBMSs to incorporate efficient CO facilities at a low cost and at a high degree of application reusability and database sharability.

1. MOTIVATION

It is widely agreed now that complex applications, such as design applications, multi-media and AI applications, and even advanced business applications can benefit significantly from database interfaces that support *composite (or complex) objects* (shortly, **CO**) [15, 23]. A generally accepted characterization defines a CO consisting of several components (possibly from different types) with relationships in between [2, 5, 11, 26, 36, 19]. Interestingly, object oriented DBMSs (OODBMSs) have adopted a very similar model [1, 43, 14, 27]. This is particularly true for OODBMSs used in practice. Especially OO programming environments have made advances in handling of COs. Such environments facilitate the growth of complex applications. As a result, there is considerable pressure on RDBMSs for better support of COs. To respond to this demand, several systems today are bridging OO environments with relational. An example of such a system is the Persistence DBMS [20], which builds a layer on the top of RDBMSs, providing better support for COs. These systems essentially extract the data from the relational and port data to OO environments.

One straightforward way of extracting data with complex structure is to follow the parent/child relationships: for each parent instance, execute a query to get the children; repeat the same thing for each child to get its children, and so on. However, this style of data extraction leads to numerous queries, and does not lend itself to effective set-oriented processing. Essentially, the process of data extraction is broken into fragmented queries where the number of fragments is in the order of number of instances of parent components in the extracted data. A better approach is to employ much more powerful set-oriented processing, where the extraction can be performed with one query. Such set-oriented processing could lead to significant improvement in performance, even in orders of magnitude. Moreover, COs are also used in path queries [27, 22], where a query may refer to one or a set of nodes through relationships. Such queries demand enhancements to query optimization in DBMSs as mentioned for example in [6, 39].

In summary, relational systems to be viable must be able to understand COs (a language extension issue), and must be able to handle them well (optimization issue). Given the prevalence of SQL as a database interface language both for application and database interoperability, we have proposed an extension to SQL to handle COs. In our approach, called SQL Extended Normal

Form (for short XNF) and introduced in [37], we derive COs from relational data. That paper covers the language part giving details on syntax and semantics. One major achievement is that such a powerful extension was made with full upward compatibility with SQL, hence opening up a path for the current DBMSs in research and industry to move forward toward handling of COs. In this paper, we attack the implementation issues, showing how to evolve a relational DBMS to handle this problem. Again, we show a growth path for RDBMSs. As mentioned before, good optimization is essential. Given the extensive optimization techniques implemented in RDBMSs, we have paid attention to reusing these capabilities. Again, from the practical viewpoint, this is very attractive since the optimization component is not cheap.

XNF's notion of CO is based on the view paradigm defining CO views (also called object views [35, 44] or structured views) composed from an underlying relational database. Hence, COs provide an abstraction mechanism that unifies both the structural view as well as the tabular view of the data. With this approach the applications benefit from the power of RDBMSs to handle the tabular data, and benefit from the XNF extensions as well. This enables data sharing among traditional applications and CO applications and even offers to migrate existing applications in order to exploit CO features. Essentially, this approach is a must for the large number of applications on RDBMSs that are evolving into support for modern applications mentioned earlier. Further, XNF provides an extension of SQL to support processing models for COs offering efficient navigation and manipulation facilities by means of a properly extended application programming interface (API). Currently, the cache manager supports both an extended SQL interface as well as a seamless C++ interface. The latter shows extensibility of our approach to bridge to important existing application environments.

The remainder of the paper is organized as follows. Sect. 2 describes the XNF approach to COs and provides an overview of the XNF language features and semantics. Architectural issues are addressed in Sect. 3 and an overview of Starburst's query processing architecture is given. Then Sect. 4 shows the main ideas underlying the integration of CO query processing into the Starburst relational framework. Details are given on how these concepts were realized as a language extension to the Starburst DBMS. This discussion reveals that the portion of XNF extensions to Starburst is very small, i.e., the reuse of the Starburst infrastructure (query processing components) is very high. A comparison to the relational case shows the benefits of the XNF approach. Sect. 5 looks into CO caching and API realization concepts and also reports on first practical experiences accompanied by some initial performance data and some discussions on related work. Sect. 6 gives an outlook to future work after having summarized the main achievements of XNF and its realization in Starburst. As a major result, we demonstrate a technology that enables RDBMSs to deal with COs constructed from relational data. Further, we show that the CO facilities can be easily integrated into a relational architecture, thereby reusing significant portions of the (now matured) relational query processing infrastructure.

2. XNF LANGUAGE OVERVIEW: BASIC CONCEPTS, SYNTAX, AND SEMANTICS

In XNF a CO consists of a collection of named component tables and relationships defined between these tables. Since XNF's notion of CO is based on the view paradigm defining CO *views*, the component tables and the relationships that are part of a CO are constructed from the tuples stored in the base relations, in other words, the *COs have to be instantiated from relations by evaluating XNF view queries* (c.f. [35]). 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 is called CO *abstraction* and the gap between the (stored) tabular view to relational data and the (derived) structured view achieved via CO abstraction is bridged by XNF's language facilities to be introduced in this Section.

Instead of the relational view concept (and its, by default, normalized result relations), XNF rather applies a more ER-like concept [7, 43, 38, 27] and has the *components of the view kept separate* and the *relationships in between made explicit* in order to reach the desired CO representation[†].

[†]For example, modern systems like object viewers [19, 27] or the set of Bachmann tools [4] rely on an ER-based view to the applications' (composite) objects.

The XNF approach is embodied in the powerful **CO constructor** that constitutes an XNF query. Those queries define CO views (i.e., structured views), which can be seen as an extension to the SQL view concept towards multi-table views that are organized as collections of inter-related rows. In the context of XNF, these CO views are known as **XNF views** that are defined through XNF queries. The basic building blocks for an **XNF query** are:

- **XNF tables** are nothing different than tables in the relational model. In Fig. 1 *xdept, xemp, xproj, xskills* are XNF tables. These normal form (NF) tables have attributes and they are populated by tuples that are derived from the underlying (relational) database. In general table expressions† are used to define the XNF tables. For graphical representation, XNF tables are drawn as nodes in the form of rectangles.
- **XNF relationships** are similar to the relationships known from the ER approach but, analogously to XNF tables, derived from the base data. In Fig. 1 *employment, ownership, empproperty, projproperty* are relationships. A relationship is defined between its partner tables by means of a **predicate**. In contrast to many ER models, we allow n-ary relationships since we can relate more than two partner tables in a relationship. The notion of roles (same as in ER models [7, 27]) is known, since partners can play certain roles w.r.t. a relationship. In addition there might be attributes defined for the relationship. Relationships are populated by so-called **connections** that represent the relationship instances existing between the corresponding partner tuples. Speaking in relational terms, we can say that connections are tuples that might have some relationship attributes and that show the foreign keys of the partner tuples they reference. For graphical representation purposes, relationships are drawn as edges using small black diamonds connected to their partner table nodes via an arrow pointing from the parent node to the child nodes.

An XNF query is identified by the keywords OUT OF and consists of the following parts:

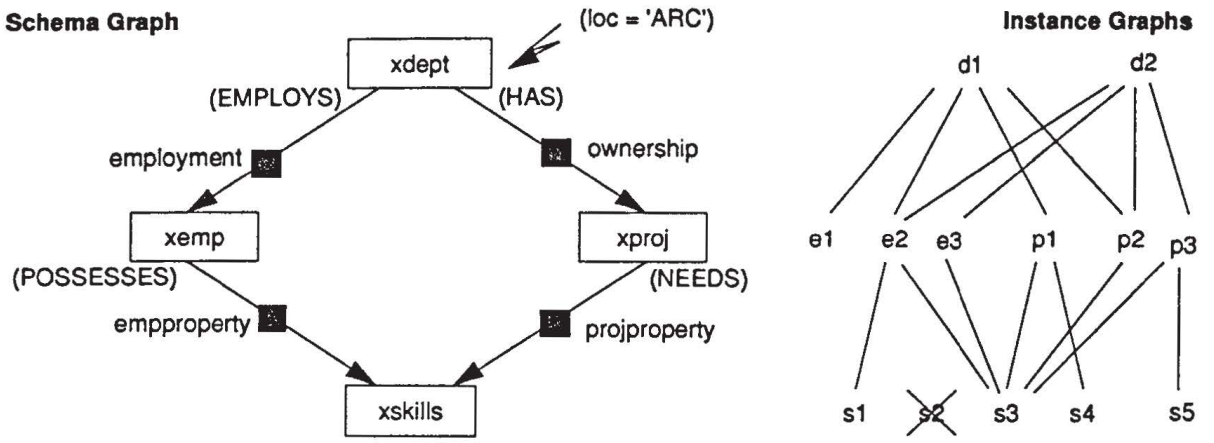
- definitions for the component tables using table expressions, in general identified by the keyword SELECT,
- definitions for the relationships, identified by the keyword RELATE, and
- specifications for the output, identified by the keyword TAKE.

Table and relationship definitions are mainly expressed using existing SQL language constructs. They make out XNF's CO constructor which can be seen as a proper extension to SQL by a compound query statement. With this, an XNF query simply reads like this:
'**OUT OF** ... *the CO* (that is constructed by the CO constructor, i.e., table and relationship definitions)
TAKE ... *the parts projected* (that define the resulting CO)'

A very similar syntax is used in OODBMSs (e.g., see [27]). However, XNF syntax provides additional constructs for definition of views, predicate-based relationships and column or node projection. We will go into further detail later. As an introduction to XNF syntax and semantics, let us discuss the example CO abstraction called *deps_ARC* given in Fig. 1. The upper part of Fig. 1 shows on the left the schema and on the right the instance level showing the COs derived, whereas the lower part of Fig. 1 gives the corresponding CO query that defines this abstraction.

This XNF query retrieves the departments located at 'ARC', and to each one the corresponding employees and employee skills as well as the projects and project skills are connected. As shown by this sample XNF query, the nodes, i.e. the component tables, are derived through standard SQL queries. Syntactic short-cuts (see definition of *xemp, xproj*, and *xskills* component tables) are provided for sake of brevity. In our example the base tables departments *(DEPT)*, employees *(EMP)*, projects *(PROJ)*, and skills *(SKILLS)* of the underlying relational database are used for derivation. The relationship tables that make up the edges of the query's schema graph show a different syntax, but basically also adopt SQL query facilities. The RELATE clause gives the

†Table expressions are similar to views in that they define a derived (or intensional) relation, but their scope is limited to the query where it is defined and its lifetime is the lifetime of the query.



XNF Query

```
CREATE VIEW deps_ARC
 AS
OUT OF xdept    AS  (SELECT * FROM DEPT WHERE loc = 'ARC'),
       xemp     AS   EMP,
       xproj    AS   PROJ,
       xskills  AS   SKILLS,
       employment AS RELATE  xdept VIA EMPLOYS, xemp
                     WHERE   xdept.dno = xemp.edno),
       ownership  AS RELATE  xdept VIA HAS, xproj
                     WHERE   xdept.dno = xproj.pdno),
       empproperty AS (RELATE xemp VIA POSSESSES, xskills
                     USING   EMPSKILLS es
                     WHERE   xemp.eno = es.eseno AND
                             es.essno = xskills.sno),
       projproperty AS (RELATE xproj VIA NEEDS, xskills
                     USING   PROJSKILLS ps
                     WHERE   xproj.pno = ps.pspno AND
                             ps.pssno = xskills.sno)
TAKE *
```

definition of component tables

definition of relationship tables

Fig. 1. Sample CO 'deps_ARC'

parent table first and then the child tables, and the WHERE clause holds the (standard SQL join) predicate that specifies the criteria for relating the partner tuples via connections. In order to read the relationship-defining query expressions in a convenient way, we have given role names (VIA clause) to the parent partners of the relationships. Based upon their predicates, the relationships establish for any given department connections to the employees it *EMPLOYS*, to the projects it *HAS*, and to the skills that either one of its employees *POSSESSES* or one of its projects *NEEDS*, or both. By means of the USING clause, a relationship may *use* data not only from its partner tables but also from other tables. For example, the two relationships *empproperty* and *projproperty* define many-to-many relationships that are derived from the mapping tables *EMPSKILLS* and *PROJSKILLS* of the relational database. Mapping tables are the typical way of modelling many-to-many relationships in RDBMSs. The *EMPSKILLS (PROJSKILLS)* base table holds information about skills possessed (needed) by an employee (project). Hence, these tables are only used for the derivation of the relationships and their connections, but they do not appear in the COs derived, i.e., at the CO abstraction level.

Retrieval of such an XNF CO results in retrieval of all the tuples of the component tables and provision for the relationship information, i.e. connections defined by the XNF relationships. Of course not all the tuples of XNF tables are meaningful for a specific CO. In the above example, obviously, only those tuples, for which there is a connection, are meaningful components w.r.t. the CO. That is, only those components that are 'reachable' within the CO are important for the CO. This concept, named *reachability*, restricts to only the relevant components of a CO. Reachability

says that each component of a CO must be reachable from another component of the same CO through a connection that also has to exist in that CO. For that reason, skill *s2* does not belong to the COs given in Fig. 1. The so-called root components (e.g. all 'ARC'-located department tuples of the XNF table *xdept*) are reachable by definition, since they define the anchors of the COs. So far, an XNF CO specifies a heterogeneous set of records with different record formats. If a component tuple is used multiple times within a view, then it exists, of course, only once in the view, but it participates in multiple connections (possibly from different relationships). Therefore the important notion of **object sharing** (illustrated by the instance graphs in Fig. 1 showing the employees *e2* and *e3*, the project *p2*, and the skill *s3* as shared objects) is a fundamental part of the XNF CO concept. An XNF query may also specify a **recursive CO** being identified by a cycle in the query's schema graph. This cycle basically defines a 'derivation rule' that iterates along the cycle's relationships to collect the tuples until a fixed point is reached and no more tuples qualify.

A convenient way of addressing parts of a CO node is the use of **path expressions**. They exploit the CO structure defined by the component tables and the relationships. A path expression consists of a sequence of component tables (and relationships), thus defining a path on the CO structure. In general, it denotes a subset of the tuples of its target table: all these tuples are to be reachable from some (root) tuples through the path defined.

XNF COs may be combined, projected, and restricted. Combination is done by simply defining a relationship between any node of one CO and any node of another one. Projection is defined by listing all the nodes and relationships to be retained. The star '*' is used as a special syntactic construct for projection of all the components with their attributes and all the relationships defined. Restriction can be done through additional predicates on the node tables and the relationships. There is also a set of CO update operators, enhancing the interface to handle insert, read, update, and delete operations. In addition, the interface supports connect and disconnect operations on relationships.

Update of the nodes is essentially identical to update of views in the relational DBMSs, because XNF nodes are derived tables expressed via table expressions. Note that use of views, as opposed to direct access to the base data, does not compromise updatability. Update of any portion of a base table can always be replaced with update of a view consisting of a proper selection over the base table. Such views are commonly used in practice for authorization. Analogously to SQL, we allow the user to go beyond such simple views, including in them operations such as join, aggregation. Such richer views provide much more powerful abstraction for queries but restrict updatability. Relationships often are defined based on simple foreign keys or connect tables. For instance in Fig. 1, *edno* in *EMP* is a foreign key and *EMPSKILLS* is a connect table. Connect and disconnect operations on such relationships translate to updating the foreign keys or inserting/deleting the associated tuples in the connect tables. Again, similar to XNF nodes, the users can define richer relationships using arbitrarily complex predicates. Such relationships provide much more powerful abstraction for read queries but restrict updatability. However quite a number of views are updatable due to the fact that a CO's component tuples are kept separate (and identifiable). This is in contrast to the relational model, where the join components are subject to concatenation, and thus the component tuples (that have to be updated) are mostly not visible anymore in the concatenated result table (i.e. cannot be updated). Although the XNF language allows to derive non- updatable views, a careful view definition might circumvent some updatablility problems. More information on that can be found in [37].

XNF provides a native API for applications that directly run on the RDBMSs. In addition, as mentioned before, we allow connection to external application environments, e.g., providing a seamless C++ interface (see Sect. 6.2). The native interface extends the cursor construct of SQL. A cursor is an iterator which allows browsing of all elements of a set, i.e. all tuples of a table. The cursor model is also popular in OODBMSs, and some provide extensive support for it [27]. XNF API provides two kinds of cursors that support navigation along the tuples of a node table (independent cursors) as well as navigation from parent to child tuples along relationship edges (dependent cursors).

All retrieval and manipulation operations of the XNF language work at the XNF level, taking into account the given graph structure and the heterogeneous tuple set. Since the result of an

XNF query consists of a set of component tables and relationships, an XNF query (or XNF view) can be used as input for a subsequent XNF query or view definition. This is also true for all other XNF operations. Therefore the model is closed under its language operations. More information on the XNF language, the multi-lingual API, and (update) semantics can be found in [37].

3. QUERY PROCESSING ARCHITECTURE

Our approach is to specify the data that needs to be extracted by an (SQL or XNF) query. That is, we must produce the *complete* result of the query. In contrast to regular SQL query, where the result is a homogeneous set of tuples, the XNF query produces tuples belonging to different nodes and relationships. Thereby, a result tuple may represent a tuple of a component table or a connection between tuples of component tables. The database server then should be able to send the complete CO to a client (on a workstation), where the application is running. To fulfil the performance requirements on the client side, the CO is transformed and converted into a main-memory representation. This means connections are represented by virtual memory pointers which allow very fast and also flexible navigation via cursors on the CO. If the CO is updatable, changes can be made locally (at the client site) and later on transferred back to the database server.

Since both the XNF language as well as the XNF API are built on SQL ideas, we decided to develop the XNF system as an *extension* to an existing RDBMS rather than building a new DBMS. Hence we advocate for an integrated DBMS, which handles both the tabular as well as the CO data. In doing so, we are able to reuse important system features that have taken years to build and are vital for e.g. system robustness, failure tolerance, and system performance. Especially, since the 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 RDBMSs remain applicable. We report on this in more detail in Sect. 4. From a technical viewpoint (and also from an economic one) we chose Starburst DBMS [16] as the starting point. Starburst was particularly attractive due to its extensibility features as we will see shortly.

3.1. Compilation and Execution Facilities

The query processing architecture of Starburst incorporates the *query language processor* CORONA [16] and the *data manager* CORE [29]. CORONA compiles queries (written in extended SQL) into calls to the underlying CORE services to fetch and modify data. Roughly speaking, CORE and CORONA correspond to System R's [3] RSS (Relational Storage System) and RDS (Relational Data System), respectively.

As depicted in Fig. 2, there are five distinct stages of query processing in CORONA; each stage is represented by a corresponding system component. For the moment, we consider only the unshaded parts of Fig. 2; the shaded areas mark XNF extensions and will be discussed in Sect. 4. An incoming SQL query is first broken into tokens and then parsed into an internal query representation called *query graph model* (shortly QGM). Only valid queries are accepted, because *semantic analysis* is also done in this first stage. During *query rewrite*, the QGM representation of the query is transformed (rewritten by transformation rules) into an equivalent one that (hopefully) leads to a better performing execution strategy when processed by the subsequent stage of plan optimization. *Plan optimization* chooses a possible execution strategy based on estimated execution costs, and writes the resulting *query execution plan* (QEP) as the output of the compilation phase. This evaluation plan is then repackaged during the *plan refinement* stage for more efficient execution by the *query evaluation system* (QES). At runtime QES executes the QEP against the database. Thereby, each QES routine interprets one QEP operator, which takes one or more streams of tuples as input and produces one or more streams as output. The adopted execution strategy, called table queue evaluation, is a demand driven, pipelined method that supplies cursors to iterate over each tuple in a result stream.

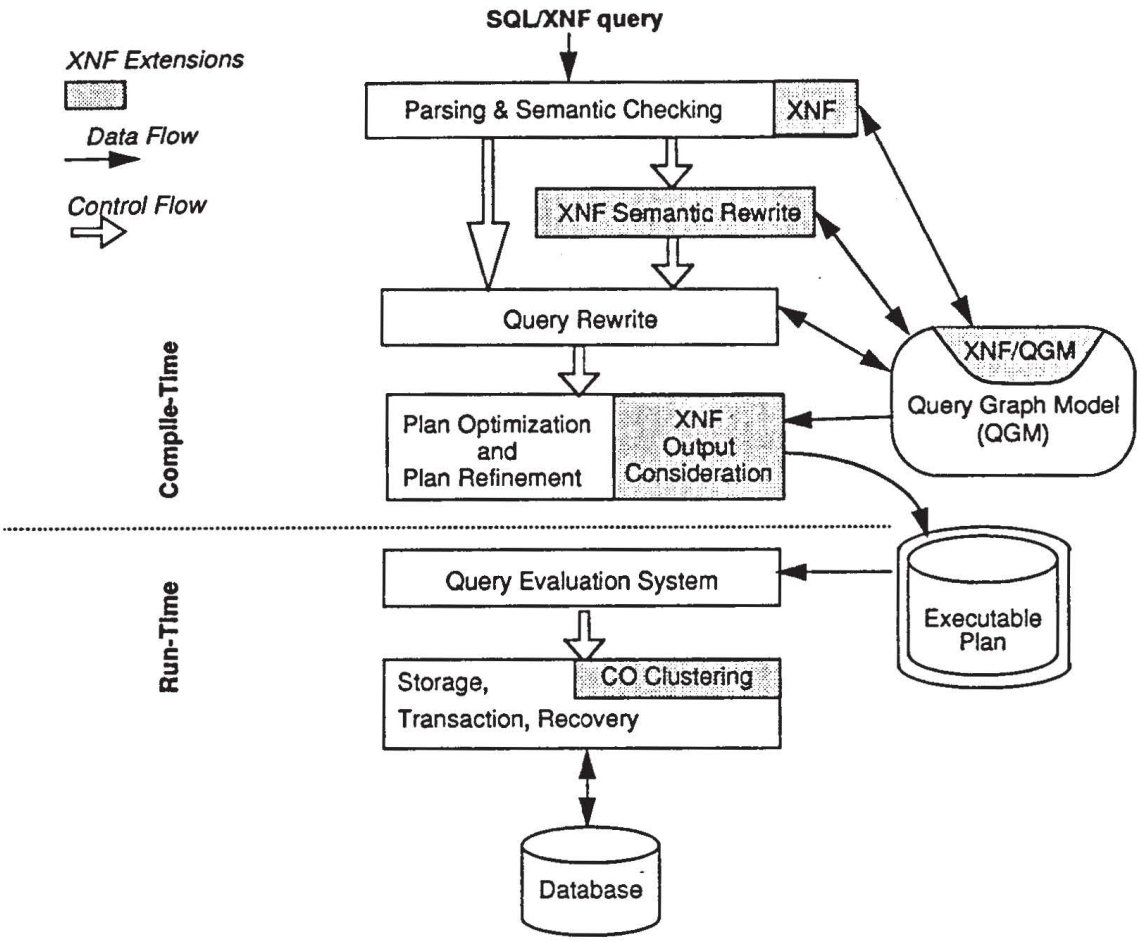


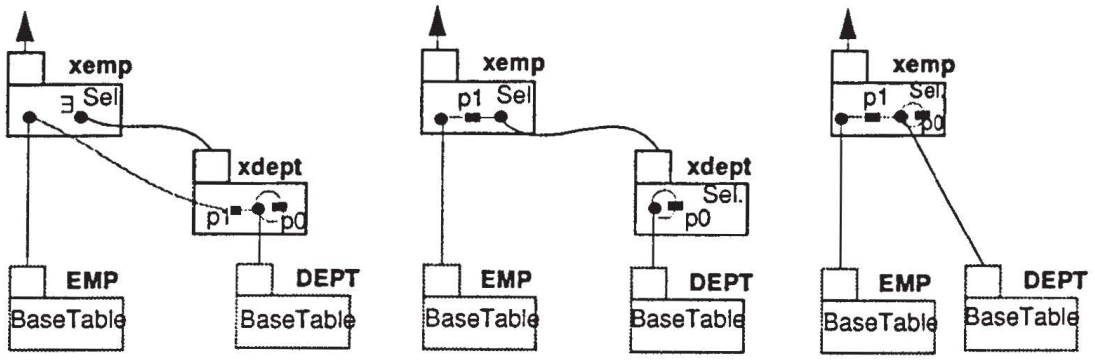
Fig. 2. Stages of Starburst Query Processing

3.2. Query Graph Model and Rewrite Optimization

The query graph model is an internal semantic network that describes the query during all stages of compilation. Since Starburst was designed to be an extensible database system also w.r.t. language extensions, orthogonality and flexibility were among the cornerstones of the QGM design.

From a logical point of view QGM can be understood as a kind of entity-relationship model that maintains attributes of query entities (e.g., base tables, derived tables, column predicates) and the relationships in between (e.g., columns belonging to tables, predicates defined over columns and constants, predicates restricting tables). Thus, QGM can be regarded as the 'schema' for a main memory database that stores information about a query. QGM is based on the notion of *table as an abstract data type.* That is, queries are represented as a series of high level operators (e.g., SELECT, GROUP BY, INSERT, UPDATE, DELETE, UNION, INTERSECTON) on either base tables (i.e., physically stored ones) 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 has to be derived from other tables the body refers to. There are properties associated with an operation such as ordering, duplicate elimination, cost, etc., which are used in the optimization step.

In a graphical notation (cf. Fig. 3) we represent the operator by a box that consists of a big rectangle that is labelled with the operator's name (e.g. 'Sel.' for the SELECT operator, for base tables either 'Base Table' or simply the table's name) and that also covers the operator's body, and small rectangles on the top that make up the operator's head. Fig. 3a shows the initial QGM representation for the SQL query that retrieves only those employees that are employed in a department located in 'ARC':



(a) Initial QGM Graph

(b) Existential Subquery to Join Query Transformation

(c) 'Merge' of Adjacent 'SELECT' Boxes

Fig. 3. Equivalent QGM Graphs

```
SELECT *
FROM   EMP e
WHERE  EXISTS (SELECT 1
               FROM DEPT d
               WHERE d.loc = 'ARC'     /* p0 */
                   AND d.dno=e.edno    /* p1 */
```

This SQL formulation uses an existential subquery. The significance of this example is that XNF path expressions and the notion of reachability naturally translate to such queries, as we will see in Sect. 4.2. OO query languages also use path expressions frequently [22, 27]. Such path expressions also translate to existential subqueries [39]. One straightforward execution strategy used in many DBMSs is to retrieve employees first and for each execute the subquery, and if the existential predicate is satisfied, output the result. Such a strategy may result in poor performance since most of such employees may not be in departments at 'ARC' location. A better strategy could be to find departments at 'ARC' location first and then get their employees. This is achieved by a rewrite optimization by first transforming the subquery to join (applying the 'E to F Quantifier Conversion' rule introduced in [39]) and then combining the two 'SELECT' boxes into one (e.g., applying the 'SELECT Merge' rule introduced in [39]). The result of these two rewrite steps in terms of QGM is shown in Fig. 3b and Fig. 3c, respectively. The final QGM graph shows a join between DEPT and EMP. Now, we can first choose the ARC departments and then find their associated employees. The significance of optimization of such queries has been shown in [6, 39]. The performance study in [39] shows orders of magnitude improvement in performance of queries with existential predicates, proving the necessity of such transformations.

4. COMPOSITE OBJECT PROCESSING

The distinguished stages of XNF's CO query processing are shown in Fig. 2. Those features that are different to the ones used in the traditional Starburst are shaded. Fig. 2 already exposes that the XNF language processor is truly an extension to the SQL processor. XNF queries are translated to a form very close to the standard SQL, allowing reuse of the extensive optimization and evaluation machinery of the RDBMS with little change. This translation is performed at compile time, and is optimized, eliminating any runtime overhead.

The discussion below will clearly disclose that the extensions only affect minor changes to the compilation part: parsing, semantic checking, and rewrite as well as the internal query representation (i.e. QGM) have to be adapted to XNF needs.Thus, we demonstrate that it is possible to integrate major language extensions, like XNF's CO facility, into a relational framework at low

cost. This is particularly true for the Starburst extensible database system and its carefully crafted query processing components.

4.1. XNF Semantic Checking

The crucial extension to the relational case was the CO constructor. Since this extension affected the language grammar, both the language parser and the semantic checking had to be extended correspondingly. In the same way as the standard SQL processor created during this phase the internal query representation, i.e., a normal form QGM graph (for short NF QGM), the XNF processor had to create the so-called XNF QGM graph that has to incorporate the XNF query semantics. In order to do this, a new operator had to be installed for QGM. The purpose of this **XNF operator** is to reflect the semantics of the language's CO constructor. Therefore, the XNF operator had to be able to incorporate $n >= 1$ incoming tables and to produce $m >= 1$ output tables being the resulting node tables and relationship tables of the CO constructed. In addition to this, regular output processing had to be modified to allow generation of a heterogeneous set of tuples in the answer set (generation of tuples belonging to different nodes and relationships). This is done by the so-called 'top' operator, which deals with the interface between the query processor and the application program. Each QGM graph has a single top operator.

In the first stage of XNF query compilation the internal query representation is built by means of the XNF semantic routines. As already mentioned, XNF QGM uses the XNF operator in order to incorporate XNF query semantics. For the XNF query from Fig. 1 we have shown the corresponding XNF QGM graph in Fig. 4. For visibility purposes, we have shaded the XNF box that represents the XNF operator. It clearly marks the context where XNF semantics applies, and it shows further the components (in the body of the XNF operator) that constitute the corresponding CO. In the following we will demonstrate how the XNF semantic routines work in order to build an XNF query graph form a given XNF query.

Since an XNF query consists of three building blocks, there are also three semantic routines that together construct the final XNF query graph. In the following we describe the subsequent phases:

(0) QGM initialization
This semantic routine initializes the QGM for the given query by installation of the XNF operator, which is drawn as a box labelled 'XNF'. If the query is named, then this XNF box gets the query's name; in our example the name of the XNF view *deps_ARC* is taken. Similar to the other boxes, the XNF box also consists of head and body: the head describes the output tables that constitute the XNF CO, and the body shows how these tables are derived from other tables the body refers to. The initialization routine also adds the top operator in the form of a box labelled 'Top'.

(1) Derivation of XNF component tables

- The semantic routines within this phase fill out the body of the XNF box, i.e. the components of the CO. Each table definition in the OUT OF clause invokes a semantic routine that defines parts of the body. There is a routine for XNF tables and another one for XNF relationships:
- An XNF table is defined as a table derived from some base tables. Therefore the corresponding semantic routine creates a 'Select' box that refers to the base tables it is derived from. Since these steps are in the context of SQL, the corresponding SQL semantic routines can be (re-)used to construct the query graph for that table expression. For example, the XNF table *xdept* is represented by a 'Select' box (within the body of the XNF box) that refers to the base table *DEPT* (represented by a 'Base Table' box) and that restricts the departments to those with location 'ARC' (predicate p0).
- An XNF relationship is also a derived table based on a table expression. A relationship's table expression at least has to relate its partner tables through the relationship predicate. For example, the table expression for the XNF relationship *employment* relates the *xdept* partner to the *xemp* partner by the relationship predicate p1 that refers to the *xdept*

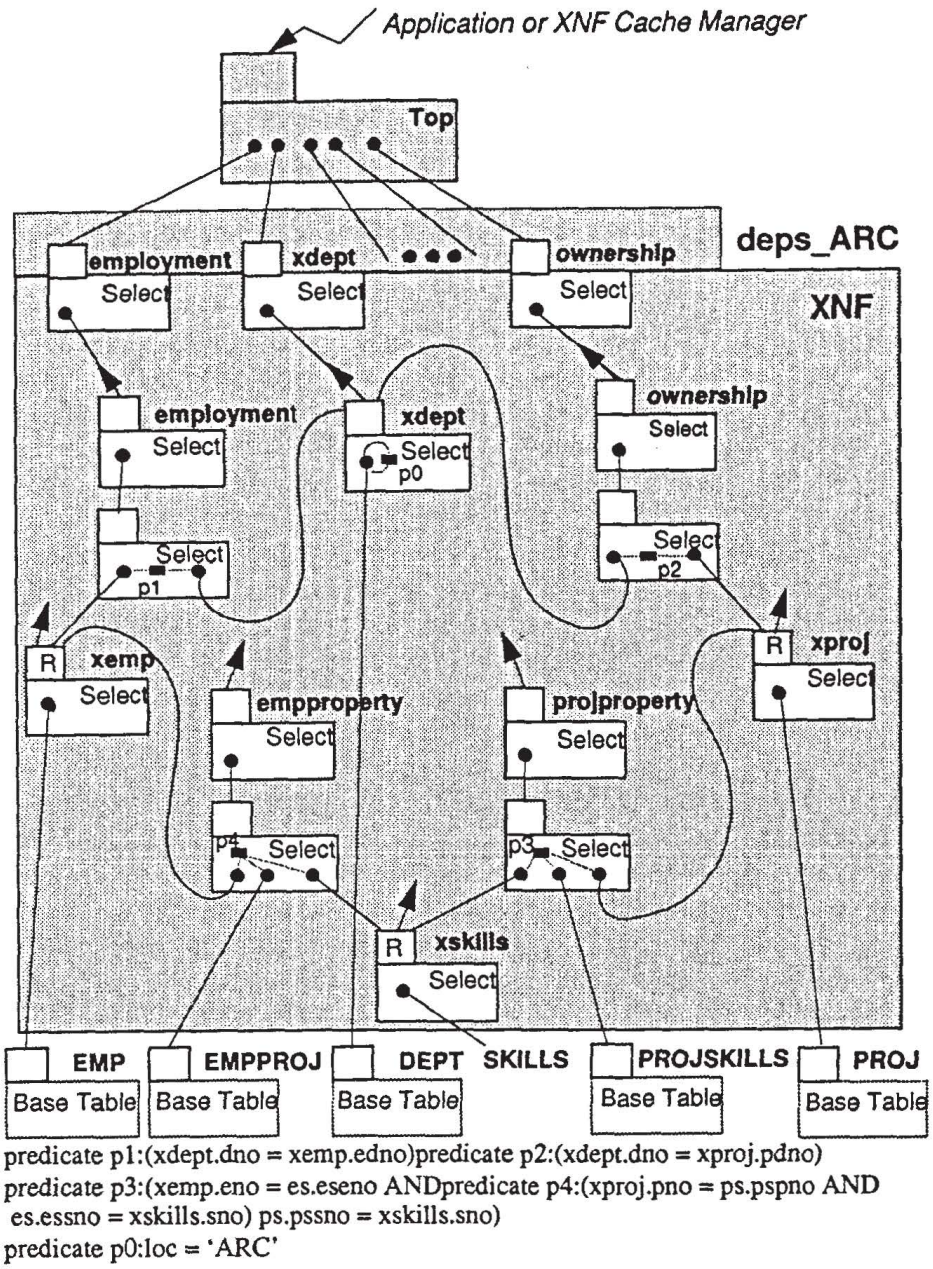


Fig. 4. XNF QGM (for the Query from Fig. 1)

and *xemp* boxes (i.e., tables). Finally, the 'Select' box labelled *employment* refers to the 'Select' box that represents that table expression, thus defining the XNF relationship *employment*. The other XNF relationships are constructed the same way. In our example both the *empproperty* and the *projproperty* relationships refer in addition to their partner tables to other tables (in that example the base tables *EMPSKILLS* and *PROJSKILLS* are referenced) that are used (c.f. USING clause, Sect. 2) in their corresponding table expressions.

(2) Consideration of component restrictions and XNF predicates
In Sect. 2 we have mentioned that it is possible to restrict a CO either via component restrictions or via XNF predicates. So far table and relationship restrictions are not exemplified, but the way this works out should be clear: these restrictions are simply added as predicates to the corresponding boxes that derive the component tables. Some XNF semantics are covered by special XNF predicates. For example, the reachability predicate allows for a fine-grained specification of components that have to be reachable from other compo-

nents. Here, and in the examples from Sect. 2, we assumed that reachability for all non-root components is defined as default. These XNF predicates are extracted and attached to the XNF operator box in order to be properly processed in the next phase (i.e. XNF-semantic rewrite). In Fig. 4 we have marked the boxes defining the components that are to be made reachable with a capital 'R' (placed in the heads of the associated boxes).

(3) Handling projection
Each element in the TAKE clause is subject to projection. For each one, we create an 'output' box (also labelled 'Select') that contains all the output columns and, in case of relationships, the role and the partner information. These output boxes refer to the boxes that represent the XNF tables and relationships. They are connected to the 'Top' box for API accessibility. For sake of simplicity, we have omitted to draw all the 'output' boxes. But those boxes of the XNF body, which are referred to by 'output' boxes, are drawn with an arrow pointing from their head to the head of their surrounding XNF box.

The above explanations revealed that the XNF constructor can be represented more or less by means of NF QGM operators. This already indicates that the XNF extensions fit into the NF query processing framework, and that it vastly exploits the basic building blocks provided by NF QGM. Therefore, interpretation of any XNF QGM graph goes similar to the interpretation of an NF QGM graph: basically, we can view the (body of an) XNF operator as a block that comprises its component tables and that gives a notation (in terms of NF QGM operators) for derivation/instantiation of its components from the base tables.

4.2. XNF Semantic Rewrite

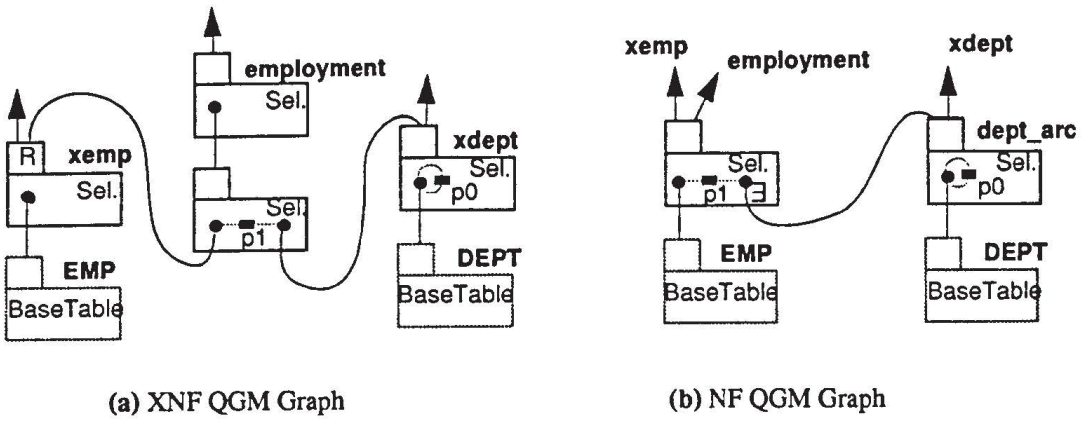
In this step the translation from XNF QGM (and XNF semantics) to NF QGM (and NF semantics) has to be accomplished, thereby transforming an XNF query graph into a semantically equivalent NF query graph. Speaking in other words, this component has to replace the XNF operator and the XNF predicates by corresponding NF operators organized in an NF QGM graph. In this step we exploit that the components (i.e., the building blocks) of COs are tables, whose derivation is already specified via NF query graphs within an XNF operator (see Sect. 4.1). XNF semantic rewrite proceeds in two major steps:
(1) Removal of the XNF operator box
(4) Consideration of XNF predicates, e.g. reachability.

After this 'compilation' down to the level of NF QGM, NF-based query rewrite takes over (see Sect. 4.3).

In the following, we will elaborate a little bit more on reachability rewrite, because it is a major task to be performed in step 2 (consideration of XNF predicates) and also because it clearly shows that XNF QGM semantics is easily expressible in terms of NF QGM semantics. In doing so, we get on one hand a better understanding of XNF semantics, and on the other hand we are able to better quantify the benefits of XNF query processing compared to conventional (SQL) relational query processing. Let us look at the XNF example in Fig. 4, and there, let us further focus on a subgraph comprising the XNF components *xdept, xemp,* and *employment.* Exactly this part is shown in Fig. 5a. In the following discussion, we will describe the consideration of reachability for the *xemp* component table, one time using conventional (SQL) relational processing techniques that only support single component retrieval, and another time applying XNF reachability rewrite within XNF's multi-table framework.

Application of XNF reachability rewrite[†] to the XNF component *xemp* (in order to have the employee tuples reachable from some *xdept* tuples via the relationship *employment*) is exemplified in Fig. 5. There we show to the left (Fig. 5a) the interesting parts of the XNF QGM graph from Fig. 4, and to the right the resulting NF QGM query graph after reachability rewrite (Fig. 5b). We can read the rewritten query graph like this: Access the department table (base table *DEPT*), apply the local predicate p0, and feed the resulting tuple stream (*dept_arc*) to the output **and** as

[†]There are several rewrite rules applicable depending on the type of relationship involved and on the QGM graph structure. However, a discussion of those different transformation algorithms is beyond the scope of this paper.



(a) XNF QGM Graph (b) NF QGM Graph

Fig. 5. XNF Reachability Rewrite

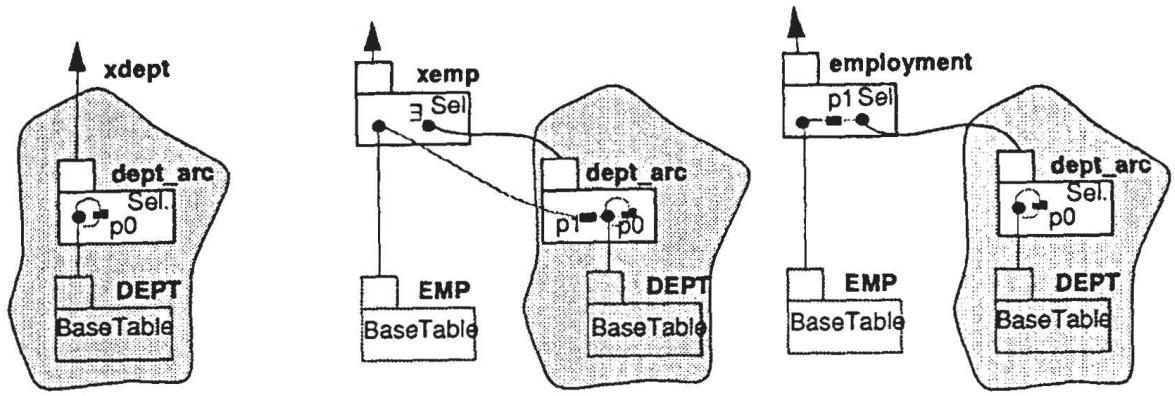
input to the computation of the *xemp* component. For the evaluation of *xemp* tuples, access the employee table and select the tuples where there exists a match to some *xdept* tuples satisfying the predicate p1 of the *employment* relationship. Note that the result is a regular SQL query. Hence all the optimization techniques, such as join order and method selection, index selection etc. apply. Since this query contains an existential subquery, the subquery to join transformation optimization, discussed in Sect. 3.2, does also apply. As a result of this transformation, *xemp* is computed by joining the tables *dept_arc* and *EMP*. This rewrite step has been mentioned already in Sect. 3.2 and visualized in Fig. 3b. The resulting tuple stream gives both the *xemp* output tuples as well as the *employment* output information†. This describes the evaluation taken at run-time and should also be sufficient to indicate the correctness of the rewrite transformation.

XNF reachability rewrite basically applies the same transformations than those relevant to the relational equivalent, i.e., when expressing reachability in terms of conventional (SQL) relational processing techniques. The above scenario (reachability for the xemp component) has already been discussed in Fig. 3 in Sect. 3.2. There we expressed reachability in an elegant way using a correlated subquery that has been rewritten to an optimized NF QGM graph. If we compare the two approaches by simply comparing Fig. 3 and Fig. 5, we clearly see that the last transformation (Fig. 3c) that merges the 'SELECT' boxes is not useful for reachability rewrite, because the intermediate result labelled *xdept* is needed for the final output since it constitutes the XNF component *xdept*.

A closer comparison shows that the query graph of Fig. 5b outputs 3 tables, i.e. *xdept, xemp,* and the relationship *employment*, whereas any query graph depicted in Fig. 3 derives only the *xemp* component. In order to get the remaining two components, two more queries (retrieving *xdept* and *employment*) have to be provided each resulting in a separate query graph. However, if we look at those three query graphs depicted in Fig. 6, we can easily determine that there are same QGM components replicated in the separate graphs. This is the case for common subexpressions marked in Fig. 6 as shaded areas. For easy interpretation, we have, in each case, added an equivalent SQL query expression. Each query creates a derived table labelled in accordance to the XNF component it represents.

Fig. 6 clearly illustrates the common subexpressions that are expressed in separate queries. We clearly see that XNF's CO constructor also provides a capability for multi-query optimization. There is only one QGM graph constructed for such a multi-table XNF query, i.e. common subexpressions are immediately installed in the corresponding query graph as depicted in Fig. 5: the derivation of the *xdept* component is also used for derivation of its child component *xemp*, and both are used for the relationship derivation.

†Since the data for relationship *employment* is already captured by the *xemp* tuples, a separate output of the *employment* connection tuples can be omitted. Fortunately, this kind of output optimization is applicable to many relationships in an XNF query. However, we have to assume that there is a simple function (available in API) that allows the extraction of the relationship information associated with a component.



CREATE **dept_arc**
AS
(SELECT *
FROM DEPT
WHERE loc = 'ARC')

SELECT *
FROM **dept_arc**

(a) Construction for CO Component *xdept*

SELECT *
FROM EMP e
WHERE EXISTS
(SELECT *
FROM **dept_arc** d
WHERE e.dno=d.dno)

(Note, this query graph will be rewritten as shown in Fig.3b and Fig.3c, thus resulting in the same query graph as given in (c))

(b) Construction for CO Component *xemp*

SELECT *
FROM **dept_arc** d, EMP e
WHERE e.dno=d.dno

(Note, this query graph is equivalent to the one shown in (b) due to query rewrite from Fig.4c)

(c) Construction for CO Component *employment*

Fig. 6. Common Subexpressions in Separate Query Graphs

Table 1. Comparison of SQL Derivation and XNF Derivation w.r.t. Common Subexpressions

Component	SQL Derivation	Replicated Query Components	XNF Derivation
xdept	1	0	1
xemp	2	1	1
xproj	2	1	1
employment	3	3	0
ownership	3	3	0
xskills	6	4	4
empproperty	3	2	0
projproperty	3	2	0
Summary	23	16	7

Comparing single component derivation in SQL (Fig. 6) with multi-table derivation as applied by XNF (Fig. 5b) clearly shows the impact of XNF's inherent treatment of common subexpressions. This is depicted in Tab. 1 (refering to the query from Fig. 1) contrasting the amount of processing needed in the XNF approach to the amount of work given by single component derivation as applied by relational systems. It shows that the single component retrieval costs 8 distinct queries (one for each component) together showing 23 separate NF QGM operations (mostly join). In the XNF approach all components are derived as XNF components performing only 6 join operations and 1 selection. The redundant operations (16 (!) operations that are subject for common subexpression treatment) are also listed in Tab. 1. Further Tab. 1 shows that the best we can do in SQL (that is no redundant processing at all) is the same as we get with XNF. From that we can conclude that the XNF rewrite approach is optimal w.r.t. common subexpression treatment. A formal proof of this is beyond the scope of this paper.

4.3. Rewrite and Plan Optimization

Since the previous step already produced a clean NF QGM (that reflects the CO query semantics), the remaining compilation work can be done by the components of the original SQL language processor. That is, the NF QGM graph built by XNF semantic rewrite is transformed by the NF query rewrite component to a semantically equivalent one that, in general, allows more efficient evaluation strategies to be chosen for the QEP when being processed by the plan optimization and query refinement components. We already gave an example of this, converting existential subqueries to joins (cf. Fig. 3). Remember, all these components are shared between the XNF language processor and the SQL language processor. A detailed description of these components can be found in [16, 30, 39].

4.4. Review of Implementation Considerations

From a software engineering point of view, we decided to have two query rewrite components: a new one for the XNF part in addition to the traditional one for the SQL part. Both apply the same transformation techniques, i.e., rule-based rewriting, and both use the same rule representation mechanism as well as the same rule engine (for more information on this see [17, 39]). This decision entailed faster and easier implementation as well as a clear distinction of responsibilities: all rewrite transformations that know about XNF context and semantics were packaged into the XNF semantic rewrite component, and all others were kept within the (NF) rewrite component. As a result, the two rewrite components became less complex and could be tested separately, which considerably simplified the validation of the XNF extensions.

However, in order to cope with the (natural) complexity of XNF QGM, we made some NF simplification rules already available to XNF rewrite. Among those were removal of unused boxes, box merge, and other clean-up operations. The first one cuts a query graph down to only relevant and used boxes, whereas the latter one condenses the graph. For example, when we look again to Fig. 4 we can see that there are lots of boxes each refering to only one single other box (e.g., the pair of boxes that is used as standard representation for XNF relationships): in most cases those pairs can be simply merged into just one box.

5. DATA EXTRACTION, XNF CACHE, AND API

In addition to standard SQL cursor support, we allow the retrieval of *all tuples* contributing to the result of an XNF query and the materialization of the *complete COs*. Traditional SQL systems produce an optimized evaluation plan to be executed at runtime. The actual retrieval is demand driven and performed using a 'one tuple at a time' interface. The application program has to call a fetch operation for each tuple to be transferred to the application. In contrast, the XNF processing model has been designed for a workstation/ server environment, where the database server can deliver complete COs on request. As mentioned before, XNF COs are handled by the database server as a heterogeneous collection of tuples. Each tuple either represents a row of a component table or a connection, i.e. an instance of a relationship. Each tuple has a (system generated) identifier and also a component number, identifying the CO component this tuple belongs to. A connection tuple contains the identifiers of the connected rows.

In Fig. 7 we show the overall structure of our prototype implementation. An application sends a data request, i.e. an XNF query to the DBMS. Query translation and optimization takes place as described in the previous Section (these compile-time activities are marked by the shaded area in Fig. 7). At runtime the generated query plan is executed and the complete result is delivered by the database system, converted into an internal main-memory representation and made accessible to the application program via the cursor interface. For long transactions, XNF allows the cache to be stored on disk and retrieved later, thereby protecting the cache from client machine's failure.

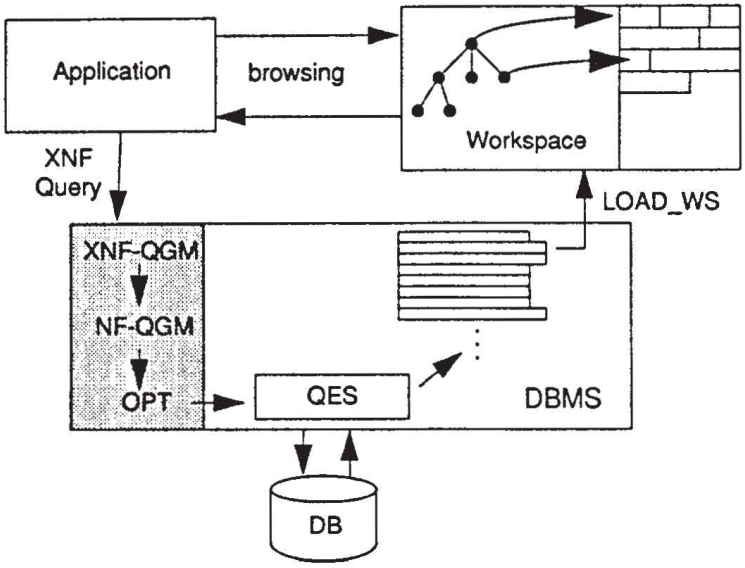


Fig. 7. XNF Cache, Data Extraction, and API

5.1. Data Extraction

The workspace is constructed from the output tuples of the XNF query by converting connections into pointers which allow traversing the structure in any direction. In addition we generate pointers to allow browsing all elements of a component and all elements of a node which are connected to a given component by a specified relationship. These pointers provide primitive access support for the cursors to be defined upon the component tables.

This data extraction scenario gives several opportunities for optimization in addition to all the well known relational optimization technology. Processing an XNF view is equivalent to processing a set of SQL queries. The difference is, that the scope for the optimizer is larger, because all these queries can be optimized together, avoiding unnecessary duplication of work. Here we can use results from research on multiple query optimization [41]. These queries typically use common subqueries to avoid unnecessary redundant computation. For instance, when we generate the tuples of a parent node, we output them, and (in case of reachability) also use them again to find the tuples of the associated children (see Sect. 4). Since parent/child relationships are computed by joins, join optimization is critical for extraction performance. Therefore, the plan optimizer should take into account any parent/child links present in the database ([28], IMS attachment), and clustering of data on disk for I/O and pathlength reduction. The next opportunity for improving the processing performance is due to the fact that the result can be delivered as a heterogeneous collection of tuples. There are no requirements on ordering (as long as there is no ORDER BY clause), so an output tuple can be produced and delivered whenever it becomes available during the execution of the evaluation plan. A third opportunity is given by the observation that there is no need for a 'one tuple at a time' interface. Database server and client workstation can cooperate in such a way that there is only one call (or only few calls) instead of a call for each tuple of the CO, thereby avoiding unnecessary crossing of process boundaries. Together with adequate CO clustering strategies, in addition to supporting index structures, these steps lead to a relatively fast extraction of COs from a relational database. Another decisive technology to reduce query execution by orders of magnitude is to apply parallelism. Set-oriented specification of COs as done in XNF particularly lends itself to exploitation of parallelism technology as proposed for conventional relational query processing in [10, 12, 25, 40].

5.2. Application Interface

Making XNF COs accessible for applications on the workstation can be implemented in several ways. In our prototype we have implemented a subset of XNF API and the XNF cache manager.

We have used this prototype to measure the performance of XNF. One significant result is that the performance of XNF cache is quite comparable with fast OODBMSs reported in Cattell's benchmark [13]. Using the traversal operation from that benchmark, we could access in a pre-loaded XNF cache more than 100,000 tuples per second which matches the requirements for CAD applications.

XNF also allows the cache to be stored in C++ structures, allowing seamless interface between applications and the data in the cache. In our prototype we have a graphical browsing facility for the data in the cache. For our running example, the seamless interface with C++ allows creating classes for *xemp* and *xdept* which include a data member, whose value is a pointer to an *xemp* object. The data extracted from the database server is transformed into the corresponding format expected by the object-oriented environment, e.g. C++ representation of *xdept* and *xemp* objects. In addition to these classes we also need a container class to hold all the instances of e.g. class *xemp*. This container class is used to allow browsing all employees without looking at their departments.

This approach brings about the possibility of further integration of database objects into the application environment. For example we can use a technique, similar to C++ templates, that provides generic XNF cursor services independent of the data type of the nodes or relationships. It basically defines a class *XNFCache* and a class *XCursor*. *XNFCache* has a component for the CO tuples and a component to hold schema information, i.e., a description of the cache structure for the CO. There is a public method, called *evaluate*, which can take an XNF query as input and construct an instance of an *XNFCache* by sending a request to the database server, loading the catalog component, and converting the heterogeneous stream of tuples delivered by the server into the main-memory representation. Access is provided through cursor objects, i.e. instances of class *XCursor*. The constructor of this class takes the name of a component table or a path expression as input and initializes, e.g., an independent or dependent cursor. *XCursor* offers a set of methods for traversing the CO as well as accessing and manipulating it's components.

5.3. Related Work

In the following we want to concentrate on two major aspects w.r.t. object processing, i.e. on shipping of data from the server to the object cache, and on linking of objects for access and manipulation services.

Object-oriented systems ship the whole objects, or even pages to the application address space. Objects are typically linked directly or indirectly via main-memory pointers. This allows fast access to objects and navigation through them. In RDBMSs, objects (tuples) are kept in RDBMS address space. As a result any manipulation of objects, or navigation through them requires crossing address spaces. This has been one reason behind the difference between performance of OODBMSs and RDBMSs.

However, this comes with a trade-off. Shipping whole pages, or whole objects to the application address space may compromise security and integrity of the database. Suppose when an application requests an object, the page containing that object is returned to the application (e.g., ObjectStore [27] ships pages to the clients). A side effect of getting this object is that other objects in that page are also shipped to the application address space. Without further access control at the client, this potentially can compromise security of the data, since the application could access these other objects. Further, application can change these other objects or the control information on the page, compromising the integrity of the data. Such a change could be accidental, due to having a wrong pointer value in the application code. Some systems, such as Versant [31] ship whole objects instead of pages. The method solves the problem of object level side-effects mentioned above. However, without further checking at the client side, it can expose all attributes of an object. For example, there is no way to hide the salary of an employee and expose only the name and address. Object shipping typically is slower than page shipping, since it often increases the traffic (number of messages) between client and server by an order of magnitude. RDBMS go to the extreme of only shipping the objects and within that only the requested attributes, although many such objects could be blocked into a single message. Further, all the manipulation and nagivation specified by DBMS requests must be done within DBMS address space. This can

be a performance boost, because only data needed by the application needs to be sent to the application. This method also provides full integrity and security. Some systems use a mechanism of application code, sometime called stored procedures, which run inside the RDBMS address space to avoid crossing the application to server boundary. If, however, a DBMS application does not exploit such a mechanism, and does not do much data filtering in its requests, it will suffer performance degradation due to DBMS address space crossings.

Linking of objects is often erroneously cited as the major advantage of OODBMSs. The technique of linking objects has been exploited in RDBMS. For example, links were introduced in System R [3] to directly link tuples on disk (i.e., links were disk addresses). This kind of links were exploited in IBM DB2 [18] for efficient navigation through catalog records. Starburst IMS attachment [9] allows links between tuples. If the main-memory storage method [32] is exploited, such links are main-memory pointers that provide comparable performance to that of OODBMSs. However, in RDBMSs, the type of the link, disk or main memory, does not change, regardless of the data being on disk or in main memory. An important advancement of OODBMSs is introduction of pointer swizzling [21, 42], which converts disk pointers to main-memory pointers upon loading of the data. This provides sharing disk data among address spaces with main-memory speed.

The CO facility provided by XNF and introduced in this paper gives flexibility and efficiency w.r.t. both shipping and linking of objects. An XNF query allows to specify a CO view through specification of its components, its relationships, and its qualification predicates in combination with a fine-grained projection facility. Since the XNF API is designed to be multi-lingual, adequate main-memory representations of the extracted COs as well as efficient navigation and manipulation facilities are inherently supported.

6. CONCLUSIONS

The novel approach of supporting COs as an abstraction over relational data is quite attractive. This approach brings the advanced CO model to existing relational databases and applications, without requiring an expensive migration to other DBMSs which support COs. The salient features of the approach are:

- a data model and query language that unifies CO and relational constructs by means of CO views,
- an elegant implementation approach that guarantees efficient data extraction, and
- a multi-lingual API with efficient navigation and manipulation facilities, and a seamless C++ interface to the cached data.

XNF defines an evolution path from RDBMSs to Composite Object DBMSs. It is interesting that we needed to make very little changes to existing RDBMSs in order to make them capable of efficiently handling COs. This is a tremendous advantage, and is very significant in practice. Therefore we called XNF an *enabling technology.* It enables RDBMSs to be extended to deal with CO and CO processing patterns. The technology is inexpensive because it heavily reuses already existing query processing components (with only comparatively small changes), and other system components as for example transaction, recovery, and storage management are totally kept unchanged. Although XNF technology largely builds upon basic relational technology, further extensions (e.g. parallelism and clustering facilities) introduced to the relational part of the system become automatically available to XNF.

XNF technology has been successfully integrated into and is now operational in the Starburst extensible database system developed at IBM Almaden Research Center. For the XNF cache, the implementation of cursor operations has been tuned for fast cursor access and efficient browsing capabilities achieving an access rate of more than 100,000 tuples per second, comparable to the performance of main-memory based OODBMSs. Language compilers (e.g., from object-oriented programming environments, like C++) can easily interface with the generic cursors of the XNF API in order to achieve a final adaptation to the applications' needs. However, note that XNF does not bind itself to only one kind of application language, rather it is open to different application

environments. Such a multi-lingual approach is important, since, in general, applications written in different languages share the data in the database.

Although the extensibility feature provided by Starburst helped a lot in integrating the XNF technology, there is at least conceptually no problem in getting it also into other (non- extensible) DBMS. This is due to the fact that XNF clearly extends SQL, and XNF technology rewrites a CO query into a semantically equivalent NF query. Therefore, only the rewrite component as well as the language extensions have to be incorporated into the query processing component of a DBMS. Extensibility just simplifies that attempt.

Another major conclusion drawn from the discussion so far, should perceive XNF as a high performance approach - as an enabling technology - that provides a path for incorporating relational data into any CO application similar to the Persistence DBMS [20] already mentioned. For example, we can use an XNF DBMS (e.g., the Starburst DBMS presented here) to provide server services to an object-oriented programming system running on the application site. This idea was realized in the prototype system called 'Object/SQL Gateway' [33] that provides object-oriented access to data residing in a relational DBMS. This gateway connects the object-oriented DBMS ObjectStore to the Starburst relational DBMS exploiting XNF technology. It is a first step in providing an integrated access to both types of DBMS using a uniform object-oriented interface. Here, XNF's multi-query optimization helps in considerably reducing the cost of data extraction from relational repository into an object cache. Further, in trying to assess XNF technology, there is already considerable confidence that the query processing concepts for COs presented (especially the multi-query framework as well as its inherent exploitation of common subexpressions) plays an integral part also in query processing for object-oriented languages as well as for deductive database languages [24, 34, 8].

Although we have motivated the performance issue, the goal of this paper is to report on the implementation techniques showing how the system works. A complete benchmarking is out of the scope of this paper, and our conducted in-depth performance study will be subject of a separate publication. In addition, we're working on further improving API capabilities to ease working with COs especially in object-oriented programming environments. Another important issue is improving XNF query processing with special emphasis on CO cluster facilities, as well as on parallelism technology [10, 12, 25, 40].

Acknowledgements — The cooperation of the whole Starburst staff is gratefully acknowledged. Special thanks are due to Peter Pistor, who helped in our joint effort of getting the good stuff into XNF, whilst streamlining the syntax and semantics. Guy Lohman improved the optimizer to handle our complex queries, and George Wilson provided valuable implementation experiences in his work on an earlier prototype. We would like to acknowledge V. Srinivasan and T. Lee of DBTI for their work on XNF cache manager and C++ application interface. We would also like to thank J. Thomas for his comments on an earlier version of this paper.

REFERENCES

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. of the 1st Int. Conf. on Deductive and Object-oriented Databases*, Kyoto-Japan, 40-57 (1989).

[2] A. Albano, G. Ghelli and R. Orsini. A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. In *Proc. 17th VLDB Conf.*, Barcelona, 565- 575 (1991).

[3] M. Astrahan, et al. System R: Relational approach to data base management systems. *ACM TODS*, **1**(1), 97 -137 (1976).

[4] C. Bachman. A Personal Cronicle - Creating Better Information Systems, with some Guiding Principles. *IEEE Trans. on Knowledge and Data Eng. 1*, 17-32 (1989).

[5] D.S. Batory and A.P. Buchmann. Molecular Objects, Abstract Data Types, and Data Models. In *Proc. 10th VLDB Conf.*, Singapore, 172-184 (1984).

[6] J. Blakeley, J. McKenna and G. Graefe. Experiences Building the Open OODB Query Optimizer. In *Proc. of the ACM SIGMOD Conf.*, Washington, 287-296 (1993).

[7] PP. Chen: The Entity Relationship Model. Toward a Unified View of Data. *ACM Trans. on Database Syst.*, **1**(1), 9-36 (1976).

[8] J. Cheiney, and R. Lanzelotte. A Model for Optimizing Deductive and Object-Oriented DB Requests. In *Proc. of Data Engineering Conf.*, Phoenix, February (1992).

[9] M. Carey, E. Shekita, G. Lapis, B. Lindsay and J. McPherson. An Incremental Join Attachment for Starburst. In *Proc. 16th VLDB Conf.*, Brisbane, 662-673 (1991).

[10] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schndeider, A. Bricker, H.-I. Hsiao and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Engineering*, **2**(1), (1990).

[11] P. Dadam and K. Kuespert, 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., 356 - 367 (1986).

[12] G. Graefe. Volcano, an Extensible and Parallel Query Evaluation System, *Research Report University of Colorado at Boulder*, CU-CS-481-90 (1990).

[13] J. Gray (ed.). *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufman Publ. Inc. (1991).

[14] L. Guzenda and A. Wade. ANS OODBTG Workshop position paper, Objectivity, Inc. In *Proc. of the First OODB Standardization Workshop*, (1990).

[15] T. Harder. Non-Standard DBMS for Support of Emerging Applications - Requirement Analysis and Archiectural Concepts. In *Proc. 22nd Hawaii International Conference on System Sciences*, Hawaii, 549-558 (1989).

[16] L. Haas, J.C. Freytag, G. Lohman and H. Pirahesh. Extensible Query Processing in Starburst. *Proc. of the ACM SIGMOD Conf.*, Portland, 377 - 388 (1989).

[17] W. Hasan and H. Pirahesh. Query Rewrite Optimization in Starburst, IBM Almaden Research Center, Research Report RJ 6367 (1988).

[18] IBM Database 2 Administration Guide, Document No. SC26-4374 (1992).

[19] W. Kim. *Introduction to Object-Oriented Databases.* Computer System Series, MIT Press, Cambridge, MA (1991).

[20] A. Keller, R. Jensen and S. Agrawal. Persistence Software: Bridging Object-Oriented Programming and Relational Database. *Proc. of the ACM SIGMOD Conf.*, pp 523-528 (1993).

[21] A. Kemper and D. Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases. In *Proc. of Ninth Int. Conf. on Data Engineering*, Vienna, 155-162 (1993).

[22] M. Kifer, W. Kim and Y. Sagiv. Querying Object-Oriented Databases, In *Proc. of the ACM SIGMOD Conf.*, San Diego, 393-402 (1992).

[23] A. Kemper and M. Wallrath. An Analysis of Geometric Modeling in Database Systems. *ACM Computing Surveys*, **19**(1), 47-91 (1987).

[24] R. Lanzelotte and J. Cheiney. Adapting Relational Optimization Technology to Deductive and Object-oriented Declarative Database Languages. *Workshop on Database Programming Languages*, Greece, (1991).

[25] R. Lorie, J. Daudenarde, G. Hallmark, J. Stamos and H. Young. Adding Intra- Transaction Parallelism to an Existing DBMS: Early Experience, *Data Engineering*, **12**(1), (1989).

[26] R. Lorie and W. Kim, et al. Supporting Complex Objects in a Relational System for Engineering Databases, IBM Research Report, San Jose, CA (1984).

[27] C. Lamb, G. Landis, J. Orenstein and D. Weinreb. The Objectstore Database System. *Communications of the ACM*, **34**(10), 50-63 (1991).

[28] G. Lohman, B. Lindsay, H. Pirahesh and B. Schiefer. Extensions to Starburst: Objects, Types, Functions, and Rules. *CACM*, **34**(10), 94-109 (1991).

[29] B. Lindsay, J. McPherson and H. Pirahesh. A Data Management Extension Architecture. In *Proc. of the ACM SIGMOD Conf.*, San Francisco, 220-226 (1987).

[30] G. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proc. of the ACM SIGMOD Conf.*, Chicago, 18-27 (1988).

[31] M. Loomis. Client-Server Architecture, *Journal on 'Object-Oriented Programming'*, (1992).

[32] T. Lehman, E. Shekita and L.-F. Cabrera. An Evaluation of Starburst's memory- Resident Storage Component, IBM Res. Rep. RJ8919, San Jose, CA, (1992).

[33] T. Lee, V. Srinivasan, J. Cheng and H. Pirahesh. Object/SQL Gateway, *presented at an OOPSLA workshop*, (1993).

[34] R. Lanzelotte, P. Valduriez, M. Ziane and J. Cheiney. Optimization of Nonrecursive Queries in OODB's. *Second Int. Conf. on Deductive and Object-Oriented Databases*, Munich, (1991).

[35] B.S. Lee and G. Wiederhold. Outer Joins and Filters for Instantiating Objects from Relational Databases through Views CIFE Technical Report, Stanford Univ., (1990).

[36] B. Mitschang. Extending the Relational Algebra to Capture Complex Objects. In *Proc. 15th VLDB Conf.*, Amsterdam, 297-305 (1989).

[37] B. Mitschang, H. Pirahesh, P. Pistor, B. Lindsay and N. Südkamp. SQL/XNF - Processing Composite Objects as Abstractions over Relational Data. In *Proc. of Ninth Int. Conf. on Data Engineering*, Vienna, 272-282 (1993).

[38] W. Kim, N. Garza, N. Ballou and D. Woelk. Architecture of the ORION Next- Generation Database System. *IEEE Trans. on Knowledge and Data Engineering*, **2**(1), (1990).

[39] H. Pirahesh, J. Hellerstein and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. of the ACM SIGMOD Conf, San Diego*, pp.39-48 (1992).

[40] H. Pirahesh, C. Mohan, J. Cheng, TS. Liu and P. Selinger. 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).

[41] T.K. Sellis. Multiple Query Processing. *ACM Transactions on Database Systems*, **13**(1), 23-52 (March 1988).

[42] S. White and D. DeWitt. A Performance Study of Alternative Object Faulting and pointer Swizzling Strategies. In *Proc. 18th VLDB Conf.*, Vancouver, 419-431 (1992).

[43] S. Zdonik and D. Maier. *Fundamentals of Object Oriented Databases. Readings in Object-Oriented Database Systems*, ISBN 1-55860-000-0, ISSN 1046-1698, Morgan Kaufmann Publishers, Inc., (1990).

[44] S. Zdonik. Incremental Database Systems. *Proc. of the ACM SIGMOD Conf.*, Washington, pp. 408-412 (1993).