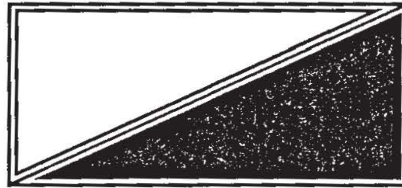
# 2



# Integration of Composite Objects into Relational Query Processing: The SQL/XNF Approach

Bernhard Mitschang
Hamid Pirahesh

## 2.1 Introduction

Complex database applications, such as design applications, multi-media and AI applications, and even enhanced business applications can benefit significantly from a database language that supports composite objects. The data used by such applications are often shared with more traditional applications, such as cost accounting, project management, etc. Hence, sharing of the data among traditional applications and complex object applications is important.

Our approach, called SQL Extended Normal Form (short SQL/XNF) provides a general framework that supports novel processing models based on composite objects. Especially, it enhances relational technology by a composite object facility, which comprises not only extraction of composite objects from a shared database, but also adequate browsing and manipulation facilities provided by an appropriate application programming interface. Further on, the language allows sharing of the database among normal form SQL applications and complex object applications. SQL/XNF provides sub-object sharing and recursion, all based on its powerful composite object constructor concept, which is closed under the language operations. SQL/XNF DDL and DML are a superset of SQL, and are downward compatible with SQL.

In this paper we concentrate on query processing issues for composite objects. We discuss the main ideas underlying the integration of composite object processing into a relational framework and contrast this to common relational query processing. We also report on the realization of these concepts in our implementation of SQL/XNF as an extension to the Starburst DBMS.

## 2.2 Motivation

Complex applications, such as design applications, multi-media and AI applications, and even enhanced business applications ask for a database processing model that is different to the conventional ones. The idea is to have a processing model that naturally supports in each state of application processing the actual processing context or 'working set.' Such a **context** defines the amount of data needed to perform a specific application task. E.g., VLSI design is done one step at a time using different tools: a synthesis tool is used to transform logic description of a chip into chip structure data, which is then used by the planning tool to design the floorplan that, in turn, is fed into

the chip assembly tool in order to get out the mask layout that is needed for chip manufacturing. The process model adopted by each tool is characterized by first reading its input data, then working in this context, and finally writing its output data (that in most cases is then input data of the subsequent tool that takes over). Similar scenarios exist in mechanical CAD, geographic applications (i.e., geographic information systems), and even in software engineering: each tool selects its input data (e.g., the surface geometry of an airplane, the street maps for the downtown area of Berlin, the software module ' index manager' of DB2), i.e., context data of say 10MBytes, out of a shared database in the TBytes range covering all the applications' CAD data.

Zooming into the data that defines a context for a tool, we recognize that a context consists of components that are related to each other according to the applications' semantics. E.g., the (input) context for the chip planning tool describes the structure of the 'cell under design' consisting of components defining the sub-cells together with their area estimations, and components giving the net data as well as the pin data that define the connections between the sub-cells relating sub-cell components through pins on the same net [Wiederhold 1986]. Similarly, geometric models [Mortenson 1985] like CSG (constructive solid geometry), BREP (boundary representation), or spline-based geometry representations are used in CAD applications, and topology-based models [Guenther 1991] prevail geography applications in order to represent the context data. This analysis characterizes a context as a so-called **composite (or complex) object** (shortly, **CO**) consisting of several components (possibly from different types) with relationships in between. In the following, we prefer the notion of composite object in order to emphasize that a CO is composed of multiple interrelated components; still it can show a complex inner structure that is transparent at the CO level. Of course different tools and applications may have different COs that may overlap. Therefore COs are mostly 'views' (**object views** or **structured views**) composed from a shared database.

The provision of a context requires from the application program the specification of the context data (and data structures) and from the underlying database management system (DBMS) efficient delivery of the component and relationship data. This means for the DBMS to provide, on the one hand, an adequate interface that allows the application program to define its contexts in form of declaratively (and not procedurally) specified composite objects. On the other hand, the DBMS has to extract the CO, i.e., its components and relationships, out of the shared database through efficient subsetting,

i.e., qualification, and structuring. Once a context is extracted (and loaded), the application program works with the context data, mostly navigating by means of the relationships defined. When the context is properly defined there is no object-faulting any more. This contrasts the processing model most object-oriented systems apply, that is, after having specified some object as 'starting points' the applications navigate over the object network using the relationships (pointers or references) installed. There, object-faulting is not precluded as is in the context-based processing model described above.

Since relational technology is generally accepted and widely used in traditional as well as in engineering applications, a lot of data is already stored in relational databases and accessed through those applications. Even more data is being transferred from flat files and navigational DBMSs such as IMS and DBTG-type systems to relational systems. The major goal and driving force in doing this is sharing of data between multiple application types, i.e., among traditional applications and CO applications.

From a practical point of view there is a need to bridge the gap between a relational store (i.e., data stored in relational DBMSs) and the CO abstraction level mentioned before. The XNF approach provides solutions to this problem offering a CO interface to relational data and handling CO at the application interface. In order to do this, the XNF approach must comprise

- a sound language and data model that unifies composite objects and relational concepts,
- an application programming interface that adequately supports navigation and manipulation, and
- an efficient implementation approach.

In this paper we concentrate on query processing issues for composite objects. We discuss the main ideas underlying integration of composite object processing into a relational framework and contrast this to common relational query processing. We also report on the realization of these concepts in our implementation of SQL/XNF as an extension to Starburst DBMS [Haas 1990]. The rest of the paper is organized as follows: Section 2.3 describes the XNF approach to composite objects. Then, in Section 2.4, we repeat the main steps and issues in relational query processing, and Section 2.5 talks about processing of CO queries. Issues in query representation as well as rewrite optimization are the main focus. Section 2.6 draws a line to related work and

gives an outlook to future work after having summarized the main achievements of XNF and its realization in Starburst. For the course of the paper it is assumed that the reader is familiar with SQL syntax and semantics [ISO-ANSI 1989].

## 2.3 SQL/XNF Approach to Complex Objects

In bridging the gap between relational data and CO abstraction, XNF has to handle COs that are stored in relational DBMSs. This means, the components and the relationships that are part of a CO definition must incorporate the relational data or, speaking the other way around, components and relationships have to be derived from the relational data, i.e., from the tuples stored in flat tables. In the words of [Lee 1990], the COs have to be instantiated form relations by evaluating view queries.

The major achievements of the XNF approach to be described in this chapter are:

- sharing of components both within and between objects, thus permitting an (component) object to play multiple roles in relationships to other or the same objects,
- recursive COs, where an object may have subobjects that, in turn, may have subobjects, arbitrarily deep, enabling, e.g., bill-of-material processing or management of organization hierarchies,
- CO views, allowing different COs to be defined over the same shared data.
- retention of benefits from the relational model, like declarative queries over COs, closure of the model w.r.t. its query language, and provision of a consistent extension of the relational data model,
- CO abstraction, that is, data contained in existing relational DBMSs can be presented to applications at an appropriate level of abstraction.

### 2.3.1 Basic Concepts, Syntax, and Semantics

Instead of the relational view concept (with its by default normalized result relation), we would rather apply a more ER-likeconcept [Chen 1976]

and have the components of the view kept separate and the relationships in between made explicit in order to reach the desired CO representation. Even modern systems like IBM's Repository Manager or the set of Bachmann's tools [Bachman 1989] rely on an ER-based view to the applications' (composite) data. This way of thinking is at the heart of the XNF approach and embodied in XNF's powerful **CO constructor** that constitutes an XNF query. These queries define CO views (i.e., structured views), which can be seen as an extension to the SQL view concepts 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. Unless otherwise noted, both terms are treated interchangeable in the following discussions. The basic building blocks for an **XNF query** are:

- **XNF tables** are nothing different than tables in the relational model. These normal form (NF) tables have attributes and are populated by corresponding tuples that are derived from the underlying (relational) database. In general table expressions are used to define these tables. For graphical representation, 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. 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 [Chen 1976]) 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 relations existing between the corresponding partner tuples. Speaking in relational terms, we can say that connections are tuples that show the foreign keys of the partner tuples they reference and the relationship attributes if defined. Drawing this analogy is very useful, because we can treat relationships very similar to tables. For this reason, the term table or component table refers to both, unless otherwise noted. For graphical representation purposes, relationships are drawn as small black diamonds connected to the nodes that represent their partners.

The CO constructor is a proper extension to SQL by a compound query statement that allows the specification of a collection of tables, populated

with the records one needs to see, and of the relationships among the resulting records. An XNF query is identified by the keywords OUT OF and consists of the following parts:

- definitions for the component tables, identified by the keyword SELECT,
- definitions for the relationships, identified by the keyword RELATE, and
- specifications for the output, identified by the keyword TAKE.

Component and relationship definitions make out XNF's CO constructor. With this, an XNF query simply reads like this:
'OUT OF ... the CO (that is constructed by the CO constructor)
TAKE ... the parts projected (that define the resulting CO)'
As an introduction to XNF syntax and semantics, let us discuss the example CO 'deps-k55' given by Figure 2.1. The upper part of Figure 2.1 shows on the left the schema and on the right the instance level for CO 'deps-k55', whereas the lower part of Figure 2.1 gives the corresponding query that defines this compelx object.

As shown by this sample XNF query, the nodes, i.e. the component tables, are derived through standard SQL queries. Syntactic shortcuts (see definition of xemp, xproj, and xskills component table) are provided for sake of brevity. In our example the base tables departments (DEPT), employees (EMP), projects (PROJ), and skills (SKILLS) are used for derivation. The relationship tables that make up the edges show a different syntax, but basically also apply SQL queries for their definition. In order to read the query in a convenient way, we have given role names (VIA clause) to the parent partners of the relationships. Based upon the relationship predicates (given in the WHERE clauses), the relationships (identified by the key word RELATE) defined 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. The EMPSKILLS (PROJSKILLS) table holds information about skills possessed (needed) by an employee (project). Mapping tables are the typical way of modeling many-to-many relationships in relational DBMSs. Therefore,

these tables are only important for the processing of the relationship and for the establishment of the derived connections, but they are not needed in the result of the XNF query, hence they do not appear at the CO abstraction level.

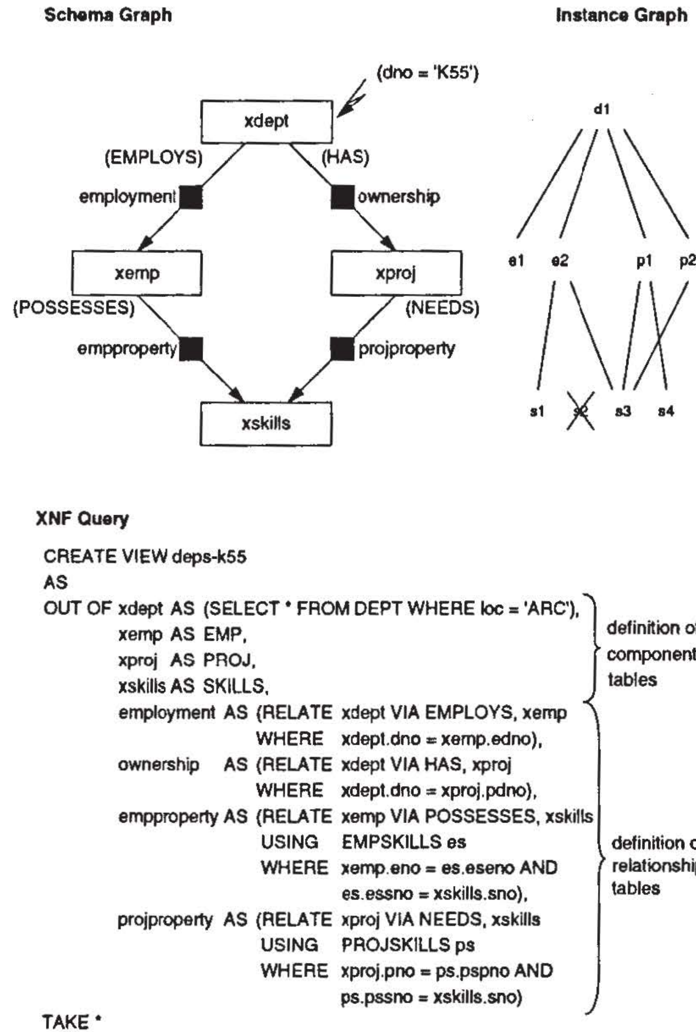



FIGURE 2.1
Sample CO 'deps-k55'

Retrieval of such an XNF CO results in retrieval of the tuples defined by the XNF tables and provision for the relationship information 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 an existing 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 relationship instance that also has to exist in that CO. The so-called roots (or root components, e.g. department tuples (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 multiply used 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** is a fundamental part of the XNF CO concept. Sharing of components can occur either because of (n:m) relationships (then it is also called instance sharing) or because of overlapping relationship definitions (then called schema sharing). Both types of sharing are naturally incorporated into XNF and can coexist. Schema sharing can be made visible by the so-called **schema graph**, which is built from the XNF tables used as nodes and the XNF relationships being the edges (see Figure 2.1). The graph visualizes the structural (as well as schema or type level) aspects of the corresponding CO. It is a very simple but expressive presentation form that is very useful when working with queries involving XNF views. Due to the reachability feature, there is a notion of parent and child partners w.r.t. relationships. This defines a direction to the relationships, which in turn makes the whole schema graph a directed graph. The arrow that is used to draw the edges gives the direction. Nodes having no incoming edge (i.e., these nodes are no child partners in any relationship) are termed root nodes. If the schema graph shows cycles, the XNF query specifies a **recursive CO**, otherwise it defines a non-recursive CO. Those nodes having more than one incoming edge are shared between their partners. This makes up schema sharing and it is referred to as **non-disjoint (shared) COs**. At the instance level we can view the nodes' tuples and the relationships' connections as being organized in a so-called **instance graph** that is built in analogy to the corresponding schema graph (see Figure 2.1).

XNF COs may be combined, projected, and restricted. Combination is simply done by definition of a relationship between any node of one CO and a node of another. Projection is defined by listing all the nodes and combining 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 predi-

cates on the node tables and the relationships. 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 relationship tables, an XNF query (or XNF view) can be used as input for a subsequent XNF query or view definition. Because all operations stay in the framework defined by XNF queries (or XNF views), the model is closed under its language operations.

### 2.3.2 API for XNF

Once an XNF query is processed and the CO consisting of components and relationships has been extracted, the application programs want to work with the CO through an adequate application programming interface (API) that supports manipulation and navigation along the given relationships. At the moment two kinds of APIs are envisioned:

- The **structure loader** loads the data in the desired format into an application-provided data space. This supports applications that, for some reasons, have specific format requirements. Manipulation and browsing of the CO is done by mechanisms provided in the application programming environment, e.g., in a C++ environment browsing can be efficiently accomplished by pointer dereferencing.

- Alternatively, a **cursor-based API** allows for individual component-oriented access. Here, each root node has a cursor associated that can be OPENed and FETCHed in order to produce successive instances of the root node. In addition, direct as well as indirect child nodes will have also cursors attached that are automatically (re-)OPENed with each FETCH from its parent cursors. Using these (hierarchical) cursors, the entire CO can be traversed.

Of course there are other kinds of API conceivable and currently under investigation, but for the purpose of this paper the above given abstract view to a CO API is sufficient.

### 2.3.3 Implementation Strategy and Overview

As shown in Figure 2.2 SQL/XNF can be seen as a language processor that creates COs and their constituting components and relationships by derivation/instantiation from relational data. Likewise to sharing of the (NF) database between traditional SQL applications and XNF applications, these

two types of applications do also share their base 'relational engine' (that is the DBMS software, e.g., an SQL DBMS). This kind of architecture allows on the one hand the traditional SQL application to run unchanged in its known environment. On the other hand, the XNF application can work at the CO level through the XNF interface that is realized by the XNF language processor. XNF queries are translated to (optimized) NF queries that are executable by the underlying relational engine. This way, XNF's CO processing is integrated into the relational framework (not an on-top solution) , thus benefiting from the wealth of available relational technology (e.g., representation structures for queries, query rewrite and optimization, storage and access structures etc.).

## 2.4 Relational Query Processing

Before talking about query processing for (XNF) COs, we want to give a better understanding of relational query processing. As our sample system, we use the Starburst Extensible Database System that is best described in [Haas 1990]. Most interesting are the relational language processor being described next and the internal representation structure for queries that will be presented thereafter.

### 2.4.1 Starburst's Language Processor CORONA

Processing of the data manipulation language is done by first, compilation of the query, and secondly, execution. Starburst consists of two components that match these two stages: the query language processor CORONA [Haas 1989], and the data man-ager CORE [Lindsay 1986]. CORONA compiles queries (written in an extended SQL version) into calls to the underlying CORE services to fetch and modify data. Roughly speaking, CORE and CORONA correspond to System R's [Astrahan 1976] RSS (Relational Storage System) and RDS (Relational Data System).

As depicted in Figure 2.3, there are five distinguishable stages of query processing in CORONA; each stage is represented by a corresponding system component. 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 equiva-

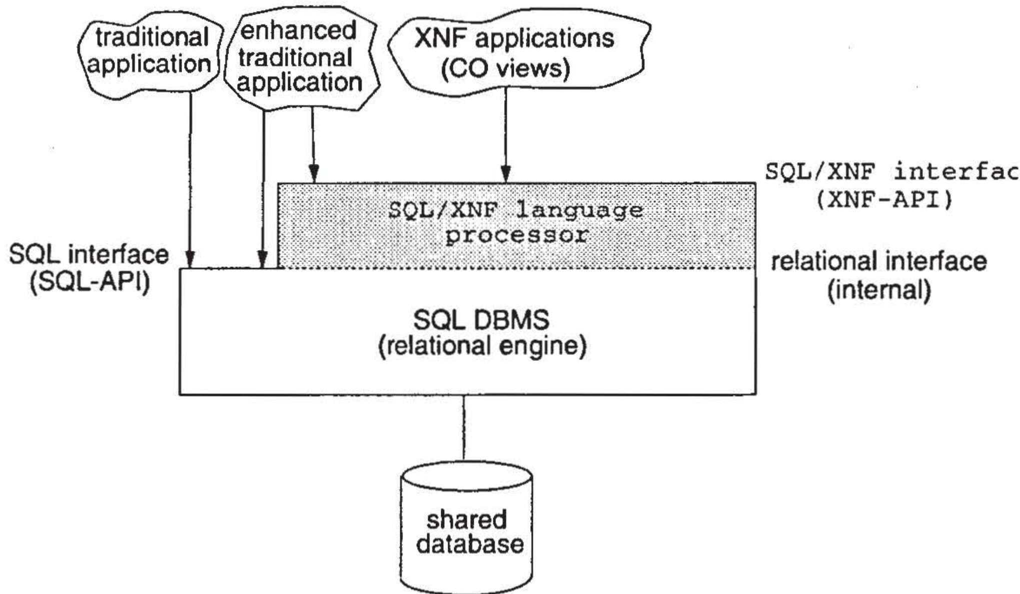



FIGURE 2.2
General Architecture of the SQL/XNF Language Processor

lent 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 by 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.

### 2.4.2 Starburst's Query Graph Model

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, the design of QGM had to be able to cope with these kind of extensions. Therefore 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-

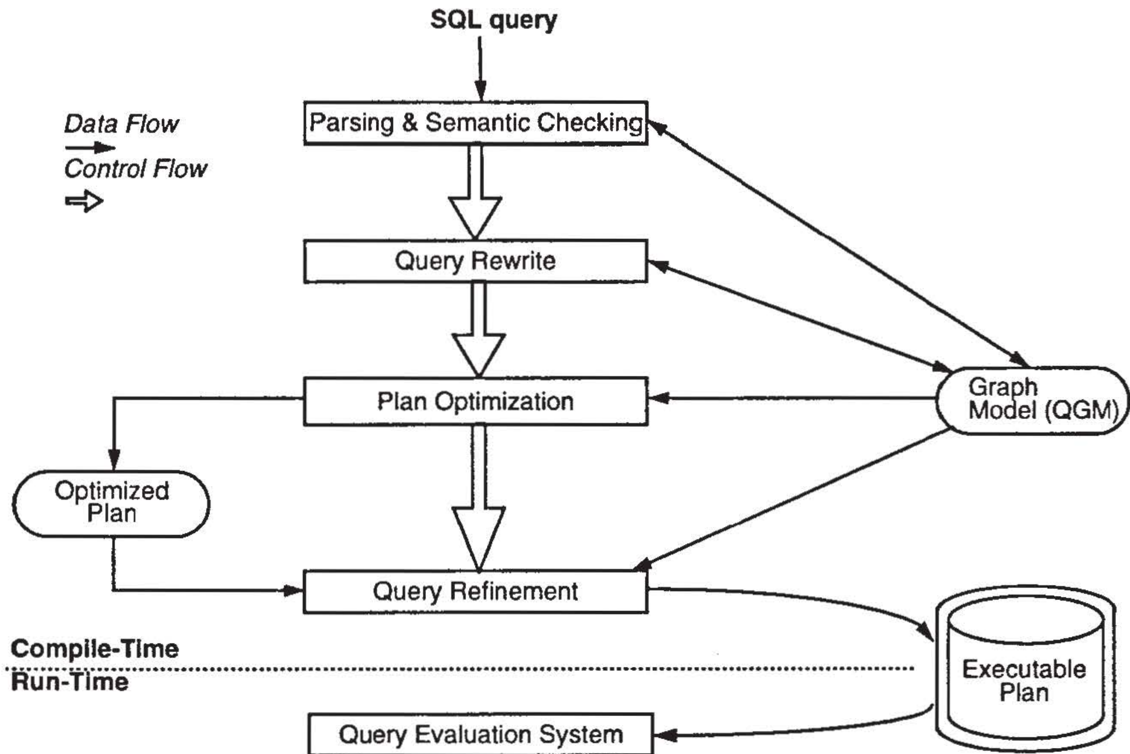



FIGURE 2.3
Stages of Query Processing

relationship model that maintains attributes of query entities (e.g., base tables, derived tables, columns 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. For a complete description of QGM and of the query transformations it permits for rewriting, we refer to [Hasan 1988]. Here we want to introduce some basic concepts through a detailed discussion of the Starburst SQL query and its corresponding QGM structure given in Figure 2.4.

QGM is based on the notion of table abstraction. That is, queries are represented as a series of high level operations (e.g., SELECT, GROUP BY, INSERT, UPDATE, DELETE, UNION, INTERSECTON) on either base tables (i.e., physically stored ones) or derived tables. An operation 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. In our graphical notation we represent the operation by a box that consists of a big rectangle that is labeled with the operation's name and that also covers the operation's body, and small rectangles on the top that make up the operation's head. Scanning a table produces a stream of tuples that has properties such as order, duplicates or no duplicates, cost etc., which are used by the optimization step.

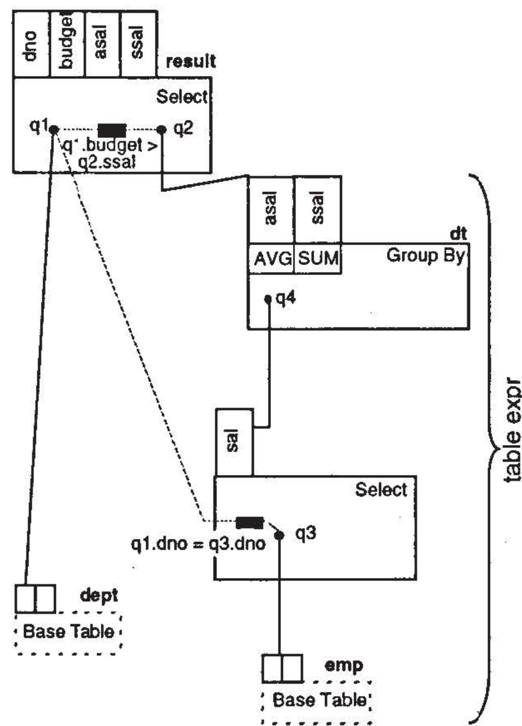



FIGURE 2.4
Sample Query and Corresponding QGM Structure

The query from Figure 2.4 retrieves for each department (range variable q1 over base table dept) the department number (dno) and department budget (budget) as well as the computed average salary (asal) and the

sum of all salaries (ssal) of the employees working for this department, if the department budget is greater than the derived sum of salaries. This query uses the language concept of derived tables: the table dt is such a derived table that is computed by a table expression given by the inner SELECT...FROM...WHERE construct that, in turn, uses the correlation concept for associating employees to their valid departments. As shown by the range variable q2 being defined over table dt, derived tables and base tables are treated the same. These logical parts of the query are easily detectable in the corresponding QGM graph. The upper QGM box is labeled 'Select' and realizes a SELECT operation, which might perform selection, projection, and join. This box represents the outer SELECT...FROM...WHERE construct. The head of this box shows the result table (here the columns dno, budget, asal, and ssal) of the query. The body consists of two vertices (called set formers) q1 and q2 that realize the two table references of the FROM clause. These set formers range over their associated tables, i.e., q1 ranges over the base table dept and q2 over the derived table dt. This is drawn as edges (called range edges). The other edge that connects q1 and q2 (called the qualifier edge and drawn as a dotted edge) gives the join predicate from the WHERE clause. The rest of this QGM graph is devoted to derive the table dt. The lowest box also labeled 'Select' ranges over the base table emp (indicated by the internally introduced set former q3) and retrieves the employees that work for the department specified by the qualifier edge that governs the correlation to q1, i.e., to the departments. The box labeled 'Group By' uses the set former q4 that ranges over the previously discussed table. The operation associated with this kind of box is grouping of tuples from the input table (referred to by set former q4) and application of aggregate functions (AVG and SUM) to each group.

At an abstract level we can interpret the QGM graph being generated like this: For each tuple within the tuple stream that comes out of the 'Base Table' box, the result box has to do two tasks. First the derived table dt has to be generated w.r.t. the actual department. In order to do this, the 'Group By' box has to be evaluated, which needs the output stream from the lowest 'Select' box. This box, i.e., operator, is responsible for selection of those employees that relate to the actual department given by the correlation predicate. Secondly after table dt has been derived, the join and the qualification are performed in order to derive the result table for this query graph.

## 2.5 Composite Object Processing

For a moment let us reconsider the things said before, and then let us put the pieces together: From Section 2.4 we have learned how to derive the result table from an (SQL) query, and the message from Section 2.3 was that COs are instantiated from the underlying relational database by derivation of the component node tables and the component relationship tables. So, CO processing in a relational framework simply means derivation of the CO's component tables; and in order to do this, we will use the techniques introduced in the previous section.

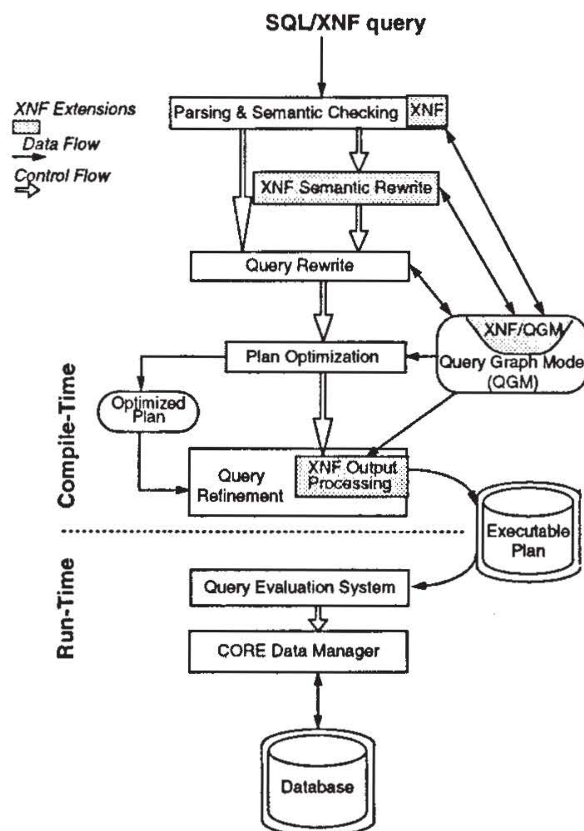



FIGURE 2.5
Stages of XNF Query Processing

### 2.5.1 Overview of XNF Language Processing

The XNF language processor (cf. Section 2.3.3) is developed as an extension to Starburst's CORONA. The distinguished stages of XNF query pro-

cessing are shown in Figure 2.5. Those features that are different to the ones used in the traditional Starburst CORONA (and shown in Figure 2.3) are shaded and all the common ones are shown unchanged. Figure 2.5 already exposes that the XNF language processor is truly an extension to CORONA's standard SQL processor. Basically we can distinguish three consecutive steps in XNF compilation. In order to emphasize more on the integration of CO processing into the relational framework, and for better understanding of the specific extensions we explain the corresponding components and their in-going and out-coming data structures. This discussion will clearly disclose that the extensions only affect 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.

1. XNF semantic routine processing
   The crucial extension to the language 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 old processor created during this phase the internal query representation, i.e., a normal form QGM graph, the XNF processor has to create the 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, this XNF operator had to be able to incorporate $n \geq 1$ incoming tables and to produce $m \geq 1$ output tables being the result node and edge tables of the CO constructed. For the rest of the paper we will call the QGM for the SQL queries NF QGM, and the one that contains the XNF operator the XNF QGM, and if the difference between both does not really matter, then we simply call it QGM. In addition to this, the top operator had to be adapted, too. The purpose of this operator is to deal with query parameters (like host variables and query constants) and to provide a basis for the API cursors; all QGM graphs have a single top operator. A description on how an XNF QGM graph for a sample query looks like will be given in the next section.

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. Speaking in other words, this component has to get rid of the XNF operator and replace

it by NF operators. In this step we exploit that the components, i.e., the building blocks, of COs are derived tables.

3. Query rewrite and plan optimization
Since the previous step already produced a clean NF QGM (that reflects the CO query semantics), the resulting compilation work can be done by the components from the SQL language processor. That is, the now NF QGM graph is taken and transformed by the 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 plan optimization and query refinement component. All these components are shared between the XNF language processor and the SQL language processor.

From a software engineering point of view, we decided to have two query rewrite components: one for the XNF part and the already existing one for the traditional SQL part. Both components 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 [Hasan 1988]). This decision entailed faster and easier implementation as well as a clear distinction of responsibilities: all rewrite transformations that must know about XNF context and semantics were packaged into the XNF semantic rewrite component, and all others were put into the (NF) rewrite component. In result we got less complex tasks to be performed by these components.

### 2.5.2 Query Representation

In the first stage of XNF query compilation the internal query representation is built by means of the XNF semantic routines. As already mentioned, this XNF QGM uses the XNF operator in order to incorporate XNF query semantics. For the XNF query from Figure 2.1 we have shown the corresponding XNF QGM graph in Figure 2.6. Again, those parts that are exclusively XNF QGM are shaded, while the NF QGM parts are kept un-shaded. In the following we will explain how such a query graph is built from a given query.

Since an XNF query consists of three building blocks, there are also three semantic routines associated that construct the final XNF QGM graph in three subsequent phases:

**(0)** QGM initialization
When it is recognized by the parser that there is an XNF query (i.e.,

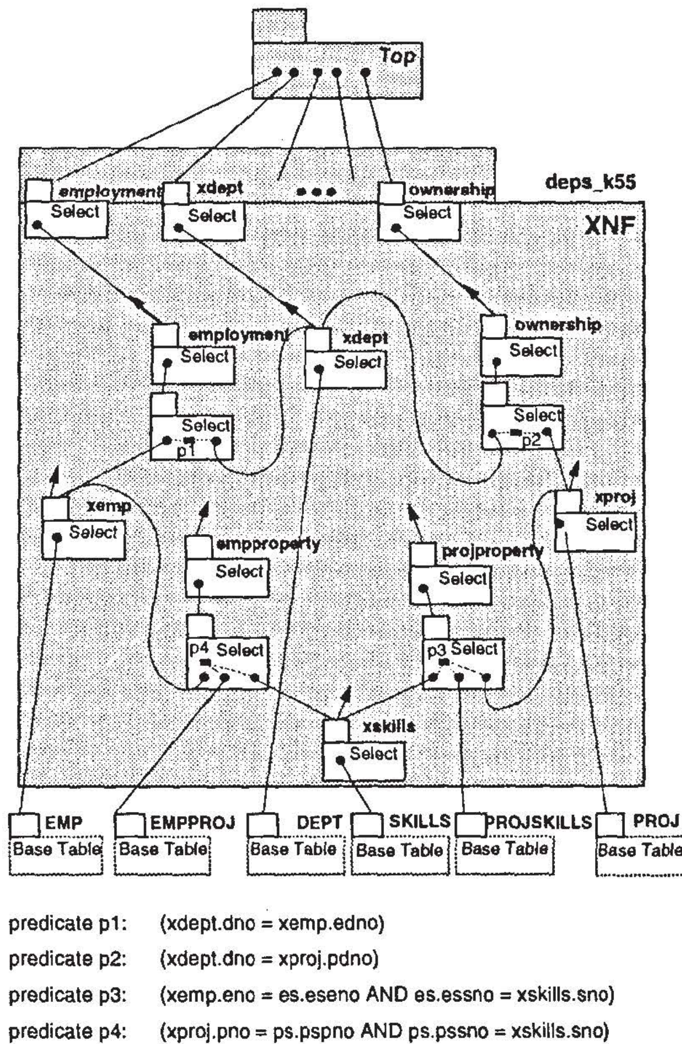



FIGURE 2.6
XNF QGM for the Example Query

when reading the key word 'OUT OF'), then the initialization routine is invoked. This semantic routine initializes QGM by installation of the XNF operator, which is drawn as a box labeled '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-k55. 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. Furthermore, the initialization routine adds the top operator in form of a box labeled 'Top.'

(1) Derivation structures for XNF component tables
The semantic routines within this phase fill out the body of the XNF box. Each table definition in the OUT OF clause invokes a semantic routine that defines parts of the final QGM. There is a routine for XNF tables and another one for XNF relationships:

- An XNF table is defined as a derived table over 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 semantic routine represents this, using NF QGM constructs. 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, in this case, also has a qualifier restricting the departments to those with location 'ARC.'

- An XNF relationship is also a derived table that is always based on a table expression. First, we represent the table expression by a 'Select' box, and then we derive from that the relationship table through a 'Select' box that refers to that 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, whose qualifier edges refer to the xdept and xemp boxes (i.e., tables). The 'Select' box labeled employment refers to the 'Select' box that represents that table expression, thus defining the derivation of 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 another table that is used in the corresponding table expression.

(2) Consideration of node and edge restrictions
In Section 2.3 we have only mentioned that it is possible to restrict XNF tables and relationships. This is not exemplified, but the way this works out should be clear: these restrictions are simply added as a predicate (i.e., as a qualifier edge) to the 'Select' box that represents the subject for restriction.

(3) Handling projection

Each element in the TAKE clause is subject to projection. For each one, we create an 'output' box (labeled 'Select') that contains all the output columns and, in case of relationships, the role and the partner information. These output boxes are connected to the 'Top' box, and they refer to the boxes that represent the XNF tables and relationships. 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 XNF box.

The above explanations revealed that the XNF constructor can be represented more or less by means of NF-QGM operators. This already shows that the XNF extensions fit into the rest of NF QGM, 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 (cf. Section 2.4.2): basically, we can view the (body of an) XNF operator as a block that comprises its components, which constitute the CO giving a notation for derivation/instantiation the components from the base data.

For an XNF QGM graph no QEP can be created and with this there is also no query evaluation by QES, because neither plan optimization nor query refinement can deal with the XNF operator. In order to get into this track we have to transform an XNF QGM graph into a semantically equivalent NF QGM graph. This is done by the XNF semantic rewrite, as already mentioned before. This component first gets rid of the XNF operator, and secondly it has to install reachability, because when we interpret the body of an XNF operator with NF QGM semantics, we then recognize that reachability is not manifested. Both transformations are done via corresponding rules that are executed by the rule engine, which is shared with NF-based query rewrite. After this 'compilation' down to the level of NF QGM, NF-based query rewrite takes over. In our design we clearly separated the rewrites that need to be done while XNF semantics is still given from those transformations that work on plain NF QGM. In order to cope with the (natural) complexity of XNF QGM, we also used some simplification rules that are also known to NF-based rewriting: removal of unused boxes, and box merge. 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 Figure 2.6 we can see that there are lots of boxes that refer only to one single other box (e.g., the pair of boxes that is used for representation of XNF relationships): in most cases

these pairs can be merged into one resulting box.

## 2.6 Conclusion, Outlook, and Related Work

In this paper we revealed that, from a practical point of view, next generation database systems are under duress for sharing data among traditional applications and CO applications. This means that there is a need to bridge the gap between a relational store (i.e., data stored in relational DBMSs) and the CO abstraction level. The XNF approach provides solutions to this problem

- by offering a CO interface through a sound language and data model that unify CO and relational concepts, and
- through provision of an API handling COs properly at the application interface.

Since the efficiency of the system is crucial, and because relational data and relational applications are omnipresent, we decided to base the implementation on the wealth of existing relational technology. That is, we integrated CO processing into the relational framework: XNF queries are translated to relational queries, optimized, and then executed by a relational engine. In order to do this, we basically had to introduce one single operator, the XNF operator, and its translation to the level of relational queries. The benefits of this approach are manifold and the most important ones are

- exploitation of proved relational technology as well as acceptance of newly developed one, like parallelization in query processing [DeWitt 1990, Graefe 1990, Lorie 1989, Pirahesh 1990],
- DBMS software sharing (e.g., compiler, rule engine, QGM, relational engine, i.e., query runtime system), and
- data sharing among relational abstractions and CO abstractions.

In contrast to our integration approach stands the on-top approach followed by Wiederhold and described in [Barsalou 1989, Lee 1990]. There an object-oriented program is interfaced 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 defines a mapping between objects and relations. That view model is restricted only to an acyclic select-project-join query. Basically this approach is comparable to XNF but major differences are obvious. First, XNF has 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 processing format. With this, XNF does not bind itself to only object-oriented application interfaces as is done in [Lee 1990]. In contrast, XNF is open to different application environments; this is especially important since there are different object-oriented models that need this kind of CO support. Thirdly, DBMS software could be considerably shared (and not replicated at different processing levels) due to the integration of XNF processing and relational processing. Fourthly, viewed from the other side, we can use XNF as another (and what we think, better) kind of view model within the system model of [Lee 1990], thus profiting from the framework defined (i.e., the object type model, the corresponding compiler etc.).

There are various other approaches to modeling and management of COs as extensions to the relational model. Lorie's [Lorie 1984] COs are defined by special columns (assigning an identifier to a row, containing the parent identifier, and referencing another row). 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 more or less hierarchical COs that are statically defined in the database schema. As liberation from these restrictions and towards more degrees of flexibility, we can view the MAD model [Mitschang 1989] that supports network-like as well as recursive COs. This Molecule Atom Data model 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 in the query by a predicate. Again, and in contrast to XNF, any membership in a MAD relationship must be explicitly specified by referencing the two partner tuples. Query processing in MAD [Haerder 1992] is also based on a set of operators, which are different from the known relational

ones due to the molecule semantics applied. Another approach that provides more flexibility as compared to Lorie's is the NF2 approach [Schek 1986]. By now it is implemented in several prototypes and extended in several ways [Dadam 1986, Linnemann 1988, Pistor 1986, Schek 1990]. This nested relation approach is targeted towards hierarchical COs by generally placing 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 nestings given in the database schema. Because of these model specific operations, the implementation reflects an extended relational engine. As in the other approaches, and in contrast to XNF, membership in a relationship is explicitly set.

What we call the navigational approach is the way many object-oriented systems deal with COs. In most cases they directly represent the relationships between the components through pointers mostly defined and managed by methods. Of course this approach to CO is not as flexible as XNF's, since on the one side only those relationships that are predefined in the static schema can be navigated on, and on the other side membership in a relationship must be explicitly set. Other approaches to COs [Stonebraker 1991, Hudson 1989] define their COs through object attributes that are evaluated in order to specify the object's sub-components. If query languages are considered (e.g., RELOOP [Cluet 1989]), then these languages show similarities to languages like MQL (MQL is the molecule query language of the MAD model [Haerder 1992]) and XNF. Therefore, there is considerable confidence that query processing concepts for COs play an integral part in OO query processing as well as in query processing for deductive database languages [Lanzelotte 1991a, 1991b, Cheiney 1992].

## Acknowledgements

The cooperation of the whole Starburst staff is greatly acknowledged. Special thanks are due to Bruce Lindsay, Peter Pistor, and Norbert Suedkamp, who all helped in our joint effort of getting the good stuff into XNF, while streamlining the syntax. G. Lohman improved the optimizer to handle our complex queries, and G. Wilson provided valuable implementation experiences on his work on an earlier prototype.

## Bibliography

[Astrahan 1976] Astrahan, M., et al. "System R: Relational Approach to Data Base Management Systems." *ACM TODS*, **1**, **1**, pp. 97–137.

[Bachman 1989] Bachman, C. "A Personal Chronicle—Creating Better Information Systems, with Some Guiding Principles." *IEEE Transactions on Knowledge and Data Engineering* **1**, pp. 17–32.

[Barsalou 1989] Barsalou, T., Wiederhold, G. "Knowledge-Based Mapping of Relations into Objects." *Computer Aided Design.*

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

[Chen 1976] Chen, P.P. "The Entity Relationship Model: Toward a Unified View of Data." *ACM TODS*, **1**, **1**, pp. 9–36.

[Cluet 1989] Cluet, S., Delobel, C., Lecluse, C., Richard, P. "RELOOP: An Algebra-Based Query Language for an Object-Oriented Database System." In *First International Conference on Deductive and Object-Oriented Databases*, Elsevier, Kyoto, Japan.

[Dadam 1986] Dadam, P., Kuespert, 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., Mayu 1986, pp. 356–367.

[DeWitt 1990] DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H.-I., Rasmussen, R. "The Gamma Database Machine Project." *Knowledge and Data Engineering*, **2**, **1**.

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

[Guenther 1991] Guenther, O., Schek, H.-J. (eds.). "Advances in Spatial Databases." *Proc. 2nd Symposium*, SSD.

[Haas 1989] Haas, L., Freytag, J.C., Lohman, G., Pirahesh., H. "Extensible Query Processing in Starburst." In *Proc. of the ACM SIGMOD Conf.*, Portland, pp. 377–388.

[Haas 1990] Haas, L., Chang, W., Lohman, G. et al. "Starburst Mid-Flight: As the Dust Clears." *Special Issue on Database Prototype Systems, IEEE Transactions on Knowledge and Data Engineering,* **2**, **1**, pp. 143–160.

[Haerder 1992] Haerder, T., Mitschang, B., Schoening, H. "Query Processing for Complex Objects." *Data and Knowledge Engineering,* **7**, pp. 181–200.

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

[Hudson 1989] Hudson, S.E., King, R., "CACTIS. A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System" *ACM TODS*, **14**, **3**, pp. 291–321.

[ISO-ANSI 1989] ISO-ANSI "Working DraftDatabase Language SQL2 and SQL3".

[Keller 1991] Keller, T., Graefe, G., Maier, D. "Efficient Assembly of Complex Objects." In *Proc. of the ACM SIGMOD Conf.*, Denver, pp. 148–157.

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

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

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

[Lindsay 1986] Lindsay, B., McPherson, J., Pirahesh, H. "A Data Management Extension Architecture." In *Proc. of the ACM SIGMOD Conf.*, San Francisco, pp. 220–226.

[Linnemann 1988] Linnemann, V., Kuspert, 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, CA.

[Lohman 1991] Lohman, G., Lindsay, B., Pirahesh, H., Schiefer, B. "Extensions to Starburst: Objects, Types, Functions, and Rules." *Communications of the ACM*, **34**, **10**, pp. 94–109.

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

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

[Mitschang 1989] Mitschang, B. "Extending the Relational Algebra to Capture Complex Objects." In *Proc. of 15th Int. VLDB Conf., Amsterdam*, pp. 297–306.

[Mortenson 1985] Mortenson, M.E. "Geometric Modeling." *John Wiley and Sons*.

[Pirahesh 1990] Pirahesh, H., Mohan, C., Cheng, J., Liu, TS, 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.

[Pistor 1986] Pistor, P., Andersen, F. "Designing a Generalized NF2 Data Model with an SQL-type Language Interface." In *Proc. of 12th Int. Conf on VLDB*, Kyoto.

[Schek 1986] Schek, H.J., Scholl, M.H. "The Relational Model with Relation-Valued Attributes." *Information Systems*, **2**, **2**, pp. 137–147.

[Schek 1990] 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*, **2**, **1**, pp. 25–43.

[Stonebraker 1991] Stonebraker, M., Kemnitz, G. "The POSTGRES Next-Generation Database Management System." In *Special Issue on Database Prototype Systems, IEEE Transactions on Knowledge and Data Engineering*, **2**, **1**, pp. 78–93.

[Wiederhold 1986] Wiederhold, G., El Masri, R. "The Structural Model for Database Design." *Entity-relationship Approach to System Analysis and Design*, North-Holland, pp. 237–257.

[Zdonik 1990] Zdonik, S. Maier, D. ed. *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers.