# Supporting Parallel Navigation in Object-Relational DBMSs

Michael Gesmann, Theo Härder

Department of Computer Science
University of Kaiserslautern
67653 Kaiserslautern, Germany
e-mail: {gesmann, haerder}@informatik.uni-kl.de

### Abstract

The use of parallelism in relational database management systems (DBMSs) provides a very effective mechanism to achieve high performance. Since object-relational DBMSs (ORDBMSs) as a successor technology continuously gain momentum in database research and development, it seems highly desirable for these systems to exploit parallel data access and manipulation to a comparable extent. However, ORDBMSs have to cope with substantial differences in the data reference pattern (highly meshed structures) and in the query processing techniques making the use of parallelism more difficult.

In this paper, we primarily consider the use of references and multi-valued attributes as key concepts of ORDBMSs and focus on issues of navigational execution models for query processing. We introduce various algorithms for parallel navigation and discuss their implementation. Furthermore, we assess the corresponding execution algorithms and discuss their use in query execution plans.

**Keywords:** Object-Relational Concepts - Parallelism - Navigation - Query Processing

## 1 Motivation

Relational DBMSs embody a very successful technology and have captured an ever-increasing market share during the last decade. Today they are dominating the DBMS marketplace and are used "everywhere" to store most of the world's business data. This overwhelming success is due to the key properties of relational DBMSs: They provide data independence, multiple views of data, and a non-procedural, set-oriented language for data access that is suitable for automatic query optimization. Furthermore, relational DBMSs allow the physical DB representation to change without impacting existing applications; these change options include storage and index structures, cluster properties, as well as partitioning or replication. Historically, however, relational systems are limited in their data modeling capabilities and expressiveness, since they traditionally support only simple predefined data types and operators.

These properties have made relational DBMSs, so far, the first commercial systems that successfully provide application-independent parallelism of query execution, that is, the optimizer can transform declarative queries into parallel query execution plans (QEPs) without any support of or reaction on the application. Their ideal suitability for parallel execution is tightly connected to the simplicity of the relational model and, in turn, to the simplicity of tables and the value-based representation of relationships (primary key / foreign key pairs). Simple and uniform operations are applied to sets of tuples (tables or relations), and the outcome of each operation is a new table. Since the (base) tables can be independently partitioned (and/or replicated) in a horizontal or vertical way, operators can be assigned and composed into highly parallel QEPs (dataflow graphs). By splitting tables and assigning uniform and independent operators to the resulting partitions, a relational system may easily achieve *partitioned parallelism.* In addition, the tuple-by-tuple output of one operator can often be streamed into the input of a subsequent operator, such that the two operators can work in a coupled mode giving *pipelined parallelism* [DG92].

On the other hand, the simplicity of the relational model and its restrictions have limited its usefulness for complex applications such as engineering design. For this reason, object-oriented DBMSs emerged in market niches and provided richer data modeling concepts and the facilities of object-oriented languages, including encapsulation, inheritance, and polymorphism. They go far beyond what relational systems can offer, in that they can store complex user-defined objects that do not neatly fit into tables and can capture the behavior as well as the state of these objects. However, they typically only provide object-at-a-time navigation, since they are often tightly coupled to a specific procedural language such as C++. As a consequence, they tend to lack important "relational goodies", in particular, data independence and automatic query optimization as well as parallel query execution.

Relational and object-oriented database systems have important advantages which vary according to the type of applications. Nowadays, more and more applications appear to have strong requirements such as data independence, non-procedural and set-oriented data access, as well as complex user-defined semantics of object orientation. In principle, there seems to be no technical reason which prohibits the combined exploitation of the advantages inherent to both technologies. However, how to design and realize such a combined model is still an open and also critical question. So far, much effort has been put into the extension of the relational model allowing columns to have complex types, including encapsulation and non-atomic values such as in nested taObjektorientierung,
Bindungszeitpunkte, Anwendungsprogrammierschnitt
stelle, Generische Methodenbles [Ch96, SB99]. While enhancing the modeling power, such "relational extensions" attempt to preserve the relational foundations in the first place, especially the declarative access to data. In contrast, effort has also been put into the extension of various object-oriented models by opening them up to make them appear somewhat "relational". Using the inheritance hierarchy as a starting point, the object-oriented remodeling aims at the inclusion of tables and table hierarchies based on object identifiers and references in the first place [Ki96]. These efforts to combine "the best of both worlds" into a single data model and to realize their advantages in a unified system led to the so-called *object-relational* approach which continues to gain momentum without precise definition of what the object-relational model (ORDM) is. In fact, this object-relational approach currently does not converge to a single, coherent data model and database system [IBM99, Informix99, Oracle98]; rather, it diverges to a spectrum of data models limited by the pure relational and object-oriented models at either extreme. A unifying element may be the standardization effort of SQL99 [SQL99] which is sometimes termed extended relational or even object-relational.

As indicated above, relational systems have successfully evolved to flexible and powerful query engines which effectively use the potential parallelism of the relational technology [DG92, MPTW94]. Performance requirements of large-volume and complex-object data management will also dictate the exploitation of parallel execution in object-relational DBMSs. However, the concepts and underlying

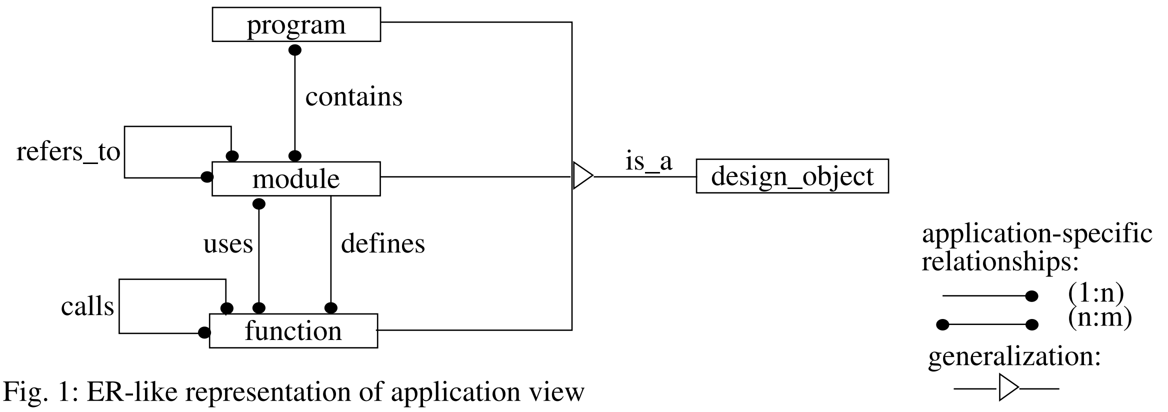



Fig. 1: ER-like representation of application view

mechanisms for query processing in object-relational DBMSs have hardly been explored so far [De96, OHUS96]. Therefore, we will primarily evaluate the potential of ORDMs and their complex object support in the context of parallel query execution. The concepts of query processing and optimization described in this paper have been explored and implemented in the PRIMA project [Ge95, Ge97a, Ge97b, HMMS87, Mi89] which was directed towards complex-object database management. We believe that these results can likewise contribute to object-relational query processing.

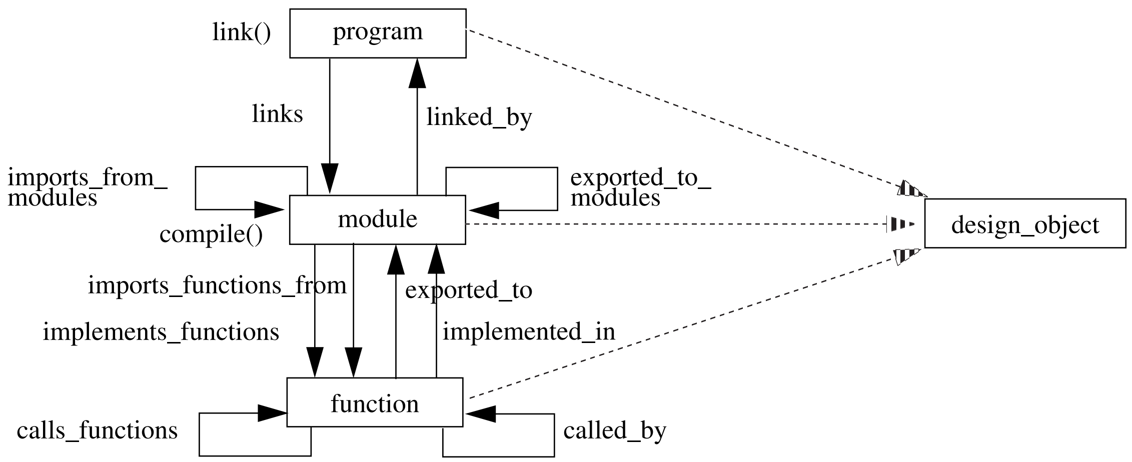
To this end, we illustrate in Sect. 2 the object-relational data model with a few descriptive queries. Navigational concepts of query processing are considered in Sect. 3. Various parallel algorithms for navigation are introduced in Sect. 4. Thereafter, Sect. 5 discusses the use of these algorithms in QEPs and Sect. 6 sketches the problems of DBMS optimizer integration by simplified examples. Finally, we conclude the article with Sect. 7.

## 2 Concepts of ORDBMSs

### 2.1 Running Example

In order to facilitate our discussion we are referring to an application scenario which may occur in some software engineering environments. For this purpose, we sketch a design object hierarchy in realistic detail containing *programs*, *modules*, and *functions*, together with application-specific relationships and the abstraction relationship “generalization“. The example is represented with ER-like graphic notations (Fig. 1). This simplified view will serve to illustrate the object-relational concepts and will be used as a running example throughout the paper.

All relationships in Fig. 1 are assumed to be bidirectional (as it is the case in the real world). When mapping these relationships to a relational DB schema, this “two-way“ property would be (somehow) preserved due to the value-based references of primary/foreign key pairs. However, the mapping itself becomes more complex, because all (n:m)-relationships have to be simulated through two (1:n)-relationships. This deviation is often considered inadequate or unnatural and may embody severe processing deficiencies. In contrast, ORDMs allow the use of reference-based relationships, thus providing a mechanism for direct representation even of (n:m)-relationships. A relationship concept preserving the property of bidirectional links as well as their referential integrity (e. g. the “inverse“ concept of [ODMG97])




a) Graphical representation

```
CREATE TYPE design_o_t AS (
  name STRING,
  last_modification DATE,
  description CLOB) NOT FINAL;

CREATE TYPE program_t
UNDER design_o_t AS (
  size INTEGER,
  state DATE,
  links REF (module_t) ARRAY [],
  . . . ) NOT FINAL;

CREATE FUNCTION link (p program_t)
RETURNS module_t <code>;

CREATE FUNCTION compile (m
module_t) RETURNS BLOB <code>;
```

```
CREATE TYPE module_t UNDER design_o_t AS (
  linked_by REF (program_t) ARRAY [20],
  imports_functions_from REF (function_t) ARRAY [],
  implements_functions REF (function_t) ARRAY [],
  imports_from_modules REF (module_t) ARRAY [],
  exported_to_modules REF (module_t) ARRAY [],
  . . .) NOT FINAL;

CREATE TYPE function_t UNDER design_o_t AS (
  code CLOB,
  result_type type_description_t,
  test_set test_t ARRAY [],
  params param_t ARRAY [],
  calls_functions REF (function_t) ARRAY [],
  called_by REF (function_t) ARRAY [],
  implemented_in REF (module_t),
  exported_to REF (module_t) ARRAY []) NOT FINAL;
```

```
CREATE TABLE design_object OF design_o_t;
CREATE TABLE program OF program_t UNDER design_object (REF IS SYSTEM GENERATED);
CREATE TABLE module OF module_t UNDER design_object (REF IS SYSTEM GENERATED);
CREATE TABLE function OF function_t UNDER design_object (REF IS SYSTEM GENERATED);
```

b) Schema description (SQL99)

Fig. 2: Kernel part of DB schema

would permit natural and elegant DB modeling as well as effective DB operations for application programs. SQL99, however, only supports unidirectional references (without integrity control) requiring application-controlled inverse-references which make DB modeling more inadequate and cumbersome. Hence, when mapping the ER-like representation of Fig. 1 to object-relational data structures we obtain the kernel part of a DB schema (see the graphical representation and excerpt of SQL99 schema description in Fig. 2). For the definition of the DB schema we assume a number of basic and predefined types available such as DATE, CLOB, test_t, etc. All reference attributes of the DB schema in Fig. 2b are graphically represented in Fig. 2a; the dotted arrows show is-a relationships (UNDER clause). Object methods are indicated by some function signatures (link(), compile()).

In the following, we briefly illustrate how these rich modeling constructs and the corresponding query facilities of ORDMs can be used to improve schema definition and query specification. In the sequel of the paper, the DB schema in Fig. 2 together with SQL99 queries is used as a running example to show the flexibility and power of navigational query evaluation.

### 2.2 Complex Object Support

Built-in types and type constructors enable the modeling of composite types. Instances of these types are often called composite objects. Additionally, nesting of such objects and usage of references between objects lead to complex objects, constructed from elementary (composite) objects or other complex objects. Complex objects, as explained in [HMMS87], are dynamically defined by a descriptive DB language and preserve their aggregate structures. At runtime, they are derived by the DBMS server as the result set of a query and shipped to the client side as sets of complex objects thereby keeping their reference structure (structured output). For example, Fig. 4 contains a sample complex object with p2 as root.[1] Complex object support facilitates the internal evaluation of complex relationships and the aggregation of compound tuples (called composite objects in the sequel). In this respect, the most important concepts are reference types, multi-valued attributes, and object methods.

Reference types, e. g., *REF (module_t),* represent inter-object connections in a direct way and make queries through reference paths more concise than through *JOINs* based on primary-key/foreign-key pairs. Another advantage of such implicit joins is that they usually offer more optimization potential than *JOINs* explicitly specified by the user. Moreover, queries with *referencing and dereferencing* (->) may be issued over a type and may automatically affect many different tables. An explicit reference used for a qualification condition on a separate table (*"functions implemented in module m3"*) can be specified as follows:

```
SELECT *
FROM function f
WHERE f.implemented_in -> name = 'm3'
```

Multi-valued attributes constructed as set, list, or array[2] make data modeling and handling much easier. For example, we can define *test_set* as an array using the constructed type *test_t ARRAY []*, and the (n:m)-relationship "uses" in Fig. 1 between *module* and *function* can be specified through the attribute *imports_functions_from* with the constructed type *REF(function_t) ARRAY []* and the attribute *exported_to* with the constructed type *REF(module_t) ARRAY [].* These typed specifications provide a much clearer understanding than storing the array elements in a separate table with a foreign key referencing the parent table. A query (*"Find all function names and the names of those modules which import functions with test1 as one of their tests applied"*) involving multi-valued attributes may look like this:

```
SELECT m.name, m.imports_functions_from -> name
FROM module m, UNNEST(m.imports_functions_from) AS f (id), UNNEST(f.id->test_set) AS t (name)[3]
WHERE t.name = 'test1'
```

---

[1] The preserved structure of the query output enables the client to process these complex objects in a navigational way. Currently, SQL99 is server-centric and does not offer specific client-side data management support. The output of an SQL99 query is always a multiset (table) of unnamed type (where the tuples are homogeneous, but may have composite values).

[2] In current SQL99, only ARRAY is supported as collection type.

[3] In SQL99, arrays can be unnested to tables and then be queried using the usual table accessing mechanism. In this example, the array *test_set* is unnested to table *t*.

This query is more concise than using explicit *JOINs* when embedded multi-valued attributes must be simulated by accessing separate tables.

Table extensions and structured types are used to extend a conventional table by allowing reference types, UDTs, and alphanumeric types as column domains. Users can attach procedures to a table and have the procedures operate on the column values. Such methods provide powerful means to model the behavior of objects and to express various facets of object semantics and relationships between objects, e. g., they can be used to control the integrity of references and inverse-references. Moreover, structured types make it possible for tables to benefit from object properties. Together with reference types, they can be employed to model complex objects. These structures lead to the regular use of path expressions.

### 2.3 Generalization/Specialization

Inheritance, which enables natural variations among types/tables, is supported by organizing types/tables into hierarchies. As illustrated in Fig. 2, *program, module,* and *function* can be modeled as subtables of *design_object*, and queries can be issued against a set of tables in the hierarchy:

```
SELECT *
FROM design_object
WHERE last_modification = '1999/5/5'
```

Such a query searching the *design_object* table and all its subtables ("*for objects that have been modified on May 5, 1999")* would be much more complicated if issued against a relational schema with separate tables for every kind of design objects, which would have to be combined with explicit multi-way *UNION* operations. Furthermore, if a *JOIN* has to be performed among two table hierarchies, the complexity increases strongly.

### 2.4 Extensible Type System

UDTs enable the set of built-in types of the DBMS to be extended with new data types and type-specific behavior. Correspondingly, a query can be enriched with method invocations in search conditions. Thus, it is possible to shift a considerable amount of processing into the method code and to avoid complete specification of computations in SQL. Since the extensible type system is not in the focus of this paper, we won't expand this brief remark.

### 2.5 Consequences for Query Processing

Relational query processing relies on simple, flat, homogeneous data sets and simple, value-based operations on these data; well-understood algorithms are available to evaluate declarative relational queries [Gr93]. In contrast, object-relational concepts imply different query processing characteristics, e. g. frequent reference-based relationship evaluation and more CPU-intensive calculation to execute user-defined methods, UDTs or BLOBs [De96, Pe96]. In particular, processing of **(implicit) joins and unions** requires heavy use of internal navigation, which is the focus of our paper. Actually, the use of navigational concepts is not a brand new idea, since in CODASYL databases application programmers also had to navigate through databases. But there, it was application programmers' responsibility to determine the optimal navigation. In contrast, we explore **system-controlled navigation** when descriptive queries are processed. We will describe some alternatives of how to exploit navigation for query evaluation and to introduce a special navigation operator as well as its use in query processing algorithms.

## 3 Navigational Query Processing Concepts

In order to explain navigational concepts, it is helpful to refer not only to the DB schema, but also to the detailed network of the DB instances and their relationships. For this purpose, however, we have to introduce an appropriate abstraction, since the full details of all relationships (see Fig. 1 and Fig. 2) cannot be captured in a comprehensive way. Therefore, we restrict our instance view to *program* - *contains* - *module* - *uses* - *function*. Since both relationship types *contains* and *uses* are (n:m) and reference-based (REF() ARRAY[]) in both directions, we draw relationship instances in Fig. 3 by bidirectional links (edges) to preserve an intuitive and clear DB instance view. Furthermore, we neither show the *isa*-hierarchy for *design_object* nor the details of the row instances/tuples.

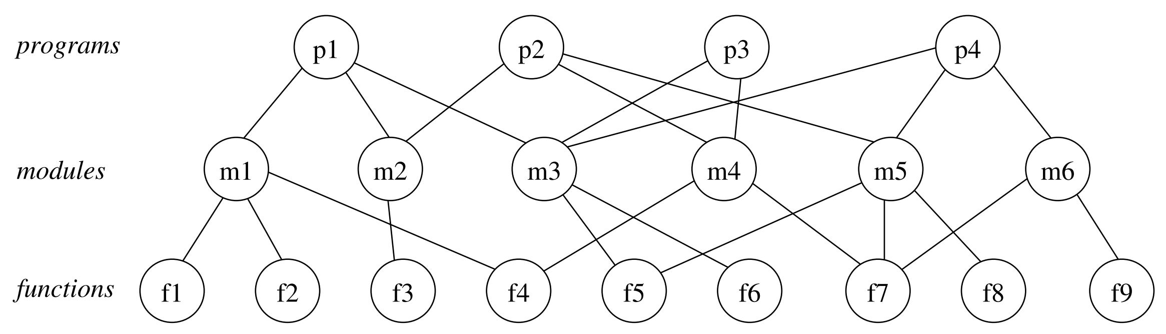



Fig. 3: DB instances and their bidirectional relationships

Let us define some terms in order to simplify the subsequent discussion:

**Query Evaluation Scope (QES)**: is the set of all DB instances and relationships to be considered in order to derive the query result (as specified in the SELECT statement).

**Result Tuple Navigation Scope (RTNS)**: is a subset of the QES to be considered in order to construct a single result tuple. A unique **root tuple** is the starting point of navigation. If bidirectional links are supported and suitable access paths are available, other entry points to an RTNS may be chosen. In case of explicit joins, several RTNSs may be relevant for the construction of a single result tuple.

**Result Tuple Qualification Scope (RTQS)**: is the subset of tuples in the RTNS required to check the qualification condition.

**Result Tuple Projection Scope (RTPS)**: is the subset of tuples in the RTNS contributing data to a single result tuple. The set of such composite result tuples (for a declarative query) is homogeneous[4].

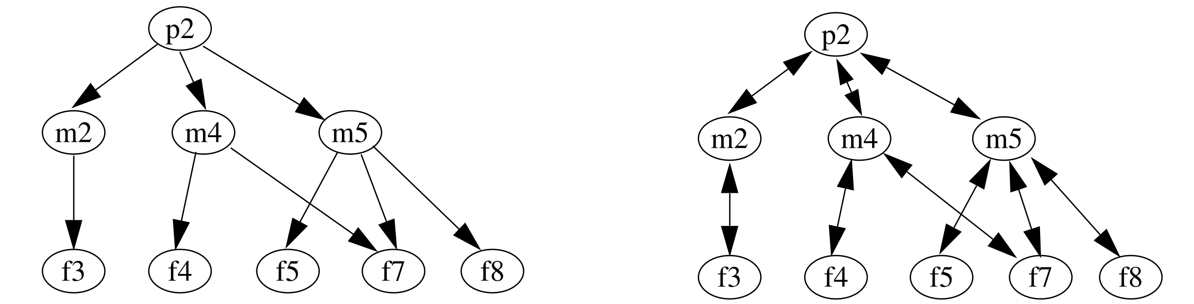
A **Referenced tuple** is accessed through navigation from some root tuple or other referenced tuple.

Note that a tuple may consist of complex structured, multi-valued attributes as well as references to

[4] Note in the relational model, the distinction between the different scopes is meaningless because the QES is always a table (virtually) constructed by the cross-product of all participating tables of the query. Qualification and projection are applied to this table. Navigation in the object-relational model makes the distinction between RTNS, RTQS, and RTPS meaningful in order to discuss optimization issues.

tuples. A tuple may belong to multiple RTNSs. Moreover, a tuple can appear as a root tuple as well as a referenced tuple. Its actual use during query processing determines whether it acts as the role of a root or a referenced tuple.

To illustrate these definitions we refer to Fig. 4a showing a sample RTNS with root tuple p2. Assume all *programs* are selected which contain at least one *module* (existential quantification) that uses *function* f3. Assume further, the result tuple should contain data from the *program* and all related *module* tuples. Hence, the minimal RTQS required for the qualification of this RTNS consists of p2 -> m2 -> f3. However, depending on the actual navigation order our RTQS can grow up to the entire RTNS if f3 is the last object access by the navigation. The RTPS contains {p2 -> m2, p2 -> m4, p2 -> m5}. If the RTQS is evaluated to "true", the corresponding RTNS is qualified. Then, the related RTPS is used to compose the result tuple.




a) Directed scope for p2 (e. g. for SQL99)    b) Bidirectional scope for p2 (e. g. for ODMG)

Fig. 4: RTNS for *program -> module -> function* derived from Fig. 3

### 3.1 Naive Processing

Let us assume, a given query on "complex objects" (meshed DB instances, e. g. in Fig. 3) extracts some data (query result) which has to satisfy some condition defined as a Boolean expression. The most intuitive way to exploit navigation is for each RTNS to start from the root tuple to request their referenced tuples recursively. This can be done in a depth-first as well as a breadth-first way. However, as shown in a couple of previous studies [e. g., KGM91, DNSV96], breadth-first strategies are favorable because they induce less I/O. The reason is that depth-first strategies may induce multiple subsequent requests to a single page while processing a single reference (recursively). In contrast, breadth-first processing usually selects all referenced tuples in a single page immediately.

After having read all tuples belonging to the RTNS, the given expression can be checked. If the expression is evaluated to "true" the RTPS is used to construct the result tuple (for the result set). Otherwise the RTNS will be discarded. Then navigation proceeds with the next RTNS.

Notice that this procedure always constructs the complete RTNS before expression evaluation. This may be very expensive if the processing of referenced tuples has high computation or I/O costs. In order to prevent unnecessary execution steps, query processing should be better adjusted to the query's result as well as to the evaluation of the given Boolean expressions. The idea is to check qualification conditions in the RTNS in such a way that a "false" evaluation minimizes access overhead to the corresponding RTNS [Sch93, Pe96]. Such a procedure is comparable to the early evaluation of restrictions in relational algebra graphs at the schema level. While algebraic optimization tries to keep intermediate results as small as possible, we want to reduce execution costs when processing single RTNSs.

### 3.2 Evaluation of Conditions on Referenced Tuples

The naive processing scheme is only optimal if all referenced tuples of the RTNS are needed to evaluate the condition, i. e. RTNS = RTQS. However, in most cases, qualification or disqualification of results can be decided before having fetched the entire RTNS. This means that we do not have to consider all referenced tuples in the RTNS before condition evaluation. Especially in the case of very large sets of referenced tuples or very expensive condition evaluation, concurrent processing of referenced data may reduce execution costs substantially.

Optimal query execution means to identify qualifying RTNSs and to compose result objects as fast as possible, i. e., to spend minimal effort to check the given condition and to skip such RTNSs which do not qualify. On the other hand, parallel processing within an RTNS should speed up result composition. Since these two optimization issues may have conflicting goals, a compromise between them has to be achieved. The general assumption is, of course, that resources are limited. Otherwise, parallel processing of all referenced tuples would always be optimal.

As already indicated, navigational query execution consists of two phases. A qualification phase on the RTQS determines qualified scopes whereas a subsequent completion phase adds still missing parts of the RTPS to the result objects. The following discussion will show how the overlap of related RTQS and RTPS affects the evaluation of an RTQS condition.[5] Moreover, we show how quantifiers in conditions determine the navigation through references.

In the following, we confine our discussion to the evaluation of simple existentially and universally quantified expressions. Nested quantifiers or more sophisticated quantifiers like EXISTS_AT_LEAST, EXISTS_EXACTLY, EXISTS_AT_MOST, etc. are beyond the scope of this paper.

#### 3.2.1 RTPS containment of referenced tuples

**Referenced tuples within RTPS**

Often an RTQS and an RTPS overlap, i. e., some referenced tuples belong to both scopes of a result tuple. When an RTNS does not qualify for a given Boolean expression, processing of the corresponding RTQS should aim at early disqualification. Therefore, the query execution engine should only process those parts in an RTNS that are needed for condition evaluation. Hence, referenced tuples should be read one at a time to avoid resource waste (e. g., for ALL quantifiers). In contrast, when an RTNS qualifies, all referenced tuples in the RTPS have to be retrieved. Then, parallel requests do not waste resources.

5 As compared to the symmetric notation for SQL queries used in section 2.2 (UNNEST in the FROM clause), the use of subqueries in the WHERE clause directly illustrates the qualification scope. By following references (e. g., from *function* f in the subsequent examples) the RTQS of the current tuple (EXISTS/ALL condition) can be efficiently evaluated using navigation.

**Referenced tuples outside RTPS**

If referenced tuples that determine qualification of an RTNS are not contained in the RTPS, query processing should pursue early disqualification and qualification, i. e., keeping the RTQS as small as possible. In both cases, tuples that are in the RTNS, but not in the RTPS and not needed for condition evaluation will be irrelevant in later processing steps. Therefore, access to such tuples should be avoided.

### 3.2.2 Quantified expressions

**Existentially quantified conditions**

Let us start with existentially quantified conditions on referenced tuples. A path expression implicitly embodies an EXISTS quantifier as exemplified by the query: *"Find the function names and the names of the modules to which the function is exported with at least one module modified later than 1999/5/5"*.

```
SELECT f.name, f.exported_to ->name
FROM function f
WHERE EXISTS(SELECT * FROM UNNEST(f.exported_to) AS m
                        WHERE m.last_modification > '1999/5/5')[6]
```

Here, all *module* tuples may be required to decide qualification or disqualification of a result tuple. Therefore, the naive processing scheme may be optimal for this example. Moreover, we know that a single *module* tuple may lead to the qualification of the entire result tuple. In the case of a more sophisticated condition which requires more CPU-intensive calculations, every referenced tuple can be considered separately. Nevertheless, since all *modules* belong to the query result, they have to be accessed. That means, the execution engine can request all referenced tuples in parallel without inducing any overhead.

The situation changes when referenced *module* tuples are not part of the query result, e. g.,

```
SELECT f.name
FROM function f
WHERE EXISTS(SELECT * FROM UNNEST(f.exported_to) AS m
                        WHERE m.last_modification > '1999/5/5')
```

Again, we may need to access all referenced *modules* to decide disqualification of a result tuple. On the other hand, a single *module* may lead to qualification of the entire RTNS. Here, it is not necessary to further consider *module* tuples which contribute neither to the condition evaluation nor to the query result. Hence, parallel access to all referenced tuples may be disadvantageous when the qualification can be determined early. In this case, the benefit of parallel tuple requests depends on the execution costs for a single referenced tuple, on the probability that single *modules* determine qualification (selectivity), as well as on the overhead induced by sequential processing of referenced objects.

**Universally quantified conditions**

In the case of universally quantified conditions on referenced tuples, e. g.

```
SELECT f.*, f.exported_to ->name
FROM function f
WHERE '1999/5/5' < ALL (SELECT m.last_modification
```

[6] Navigational evaluation does not require the explicit execution of UNNEST to build subtables. References enable direct access to the set of referenced tuples and, therefore, allow for immediate qualification or disqualification.

*FROM UNNEST (f.exported_to) AS m)*

a single *module* tuple may lead to disqualification. On the other hand, we need to fetch all referenced tuples to guarantee qualification of a result tuple. Therefore, processing such conditions does not depend on the RTPS. In both cases, the fraction of referenced tuples to be simultaneously constructed depends on execution costs for a single referenced tuple and on the probability that single referenced tuples cause disqualification.

### 3.2.3 Summary

Table 1 summarizes the analysis of this section. Here, "parallel" means that all referenced tuples can be requested and processed simultaneously without any waste of resources, and "variable" means that the number of referenced tuples to be simultaneously considered varies with the expression's selectivity as well as the costs of the condition evaluation. As a consequence, the query optimizer often has to determine the optimal number of referenced objects to be processed at a time.

Table 1: Processing of referenced tuples in the RTQS

| | EXISTS | ALL |
|---|---|---|
| within RTPS | parallel | variable |
| outside RTPS | variable | variable |

## 3.3 Top-Down Evaluation

In our discussion so far, we have only considered the evaluation of simple quantified conditions via single references. In general, of course, conditions are more complex. They may comprise multiple referenced types and may be combined by Boolean operators. A simple example is query Q1:

*SELECT m.**
*FROM module m*
*WHERE EXISTS (SELECT * FROM UNNEST (m.imports_functions_from) AS f WHERE f.name = 'f1')*
*AND '1999/5/5' < ALL (SELECT p.date FROM UNNEST (m.linked_by) AS p)*

Again, depending on the processing costs of some partial expressions within the condition (object composition as well as expression evaluation) and the Boolean connection of partial expressions, we seek an evaluation order for partial expressions which minimizes the overall execution costs and allows for fast result construction. Such an execution plan, for example, requests the referenced *program* tuples in the first phase and, after having rejected already disqualified root tuples, fetches the referenced *function* tuples in the second phase. Only in the case that two partial expressions are OR-connected and all referenced tuples in the RTQS belong to the RTPS as well, these tuples should be requested in parallel. In other cases, the choice of an execution order for the partial expressions and referenced tuples depends on the SELECT clause as well as the execution costs. With a disjunctive connection of partial expressions, query evaluation should aim at an early qualification of the entire result in order to prevent unnecessary construction of some parts of the RTNS. For a conjunctive connection of partial expressions, query execution should guarantee inexpensive processing to determine disqualification and, at the same time, allow for a fast construction of qualified results. Table 2 summarizes the discussion of this section

Once RTQS processing is finished and qualification is decided, the result tuple has to be completed using the remaining references in the RTPS not processed so far. Of course, in this completion phase (similar to the naive processing scheme) all projected parts can be requested in parallel.

### 3.4 Bottom-Up Restriction

The algorithms described so far are adjusted to the structure defined in a query, i. e. recursively from some root tuple to its referenced tuples in the QES (top-down). However, this approach is far from optimal if access path structures are available for referenced tuples and if the system maintains symmetric inverse-references (see Fig. 4b, [ODMG97]).

Table 2: Relationship between RTPS and Boolean connection of partial conditions

| | OR | AND |
|---|---|---|
| within RTPS/ within RTPS | parallel | variable (early disqualification) |
| outside RTPS outside RTPS | variable (early qualification, interrupt processing of references as soon as qualification is decided) | variable (early disqualification) |
| within RTPS/ outside RTPS<br><br>outside RTPS/ within RTPS | variable (early qualification, interrupt processing of tuples not to be projected as soon as possible) | variable (early disqualification) |

Let us reconsider our example query Q1 from Sect. 3.3. Provided there is an access path on *function.name*, those *function* tuples (referenced tuples in RTQS) that satisfy the first part of the given condition can be selected very efficiently. Subsequently navigation (along reference *exported_to*, which is diametric to *imports_functions_from*) leads to a set of *module* tuples. In contrast to the previously described top-down processing, which considers all root tuples existing in the database, the bottom-up processing continues with a restricted set of root tuples. Due to the navigation directed from component to root tuples, this processing step is called bottom-up restriction.

Of course, bidirectional navigation along (recursive) references requires system‘s control of inverse references. By considering the quantifier type (EXISTS, ALL), cardinalities of references, and NULL values, bottom-up restriction can be extended to more complicated RTQSs and expressions [Ge95]. Due to space limitations we won‘t go into details.

### 3.5 Summary

This section has outlined various uses of navigation operations in DBMSs supporting references among tuples. Starting with a naive top-down evaluation, we have shown how quantification and Boolean connection of partial expressions determine the number of referenced tuples to be considered at a time. Only in very few cases (OR-connected, existentially quantified expressions with referenced tuples in RTPS) all referenced tuples should be considered simultaneously. In all other cases, sequential processing of subsets of referenced tuples is more advantageous. Therefore, we have outlined the top-down evaluation which can process partial expressions in some order defined by a query optimizer. Preceding bottom-up restriction can reduce the number of RTNSs to be processed, if the system is aware of the symmetry of references. The combined usage of these mechanisms (top-down evaluation and bottom-up restriction, different processing sequences for references and partial predicates, parallel requests of referenced objects) offers a wide range of flexible navigation strategies. Prerequisite for navigational query processing is that existing object references and query evaluation order match. While system-maintained symmetric references always fulfil this requirement, query processing "against" unidirectional references must use value-based join algorithms.

## 4 Parallel Navigation Algorithms

So far, we have shown how single RTNS processing can take advantage of navigation. In this section, we discuss the processing of set-oriented requests of referenced objects. Similar to conventional RDBMSs, queries have to be applied to large data extents and may also return large sets of result tuples. In addition, result objects in ORDBMSs may consist of many component objects which, in turn, contribute to a complex internal object structure. Furthermore, the use of application functionality in query predicates (user-defined functions) leads to increasing demands for CPU resources. In order to guarantee acceptable response times, parallel query processing becomes more and more important. This includes parallel navigation through RTNSs via multi-valued references.

To illustrate our approaches to parallel navigation, we start with the simple query "*Find all functions with name < f4 where all modules to which they are exported have names > m2*".

*SELECT f.**
*FROM function f*
*WHERE f.name < 'f4' AND 'm2' < ALL(SELECT m.name FROM UNNEST(f.exported_to) AS m)*

In order to evaluate this query, the DBMS query engine has to consider referencing *function* tuples together with their referenced *module* tuples (via reference *exported_to*). In the same way that relational join operators assemble, i. e. concatenate, result tuples from their input tuples, (intermediate composite) objects have to be materialized for further processing steps such as selection, projection, etc. Explicit materialization is necessary because, in contrast to pure relational query processing, referenced objects are usually composite (i. e. derived). Since such a derivation may be based on selection predicates, some of the referenced objects may not qualify for the materialization.

Obviously, the relational join operator is not designed to materialize such composite objects, because it is confined to the comparison of atomic attribute values. For this reason, we introduce a new binary operator "connect" which connects a set of referencing objects called roots with their referenced objects called components. In the dynamic situation, roots and components may be objects stored in a table or intermediate composite objects, i. e., derived from other objects. Therefore, both connect operands are sketched as operator trees (op-tree) in Fig. 5. When combining related objects, the connect operator is able to apply a predicate thereby filtering the corresponding roots and components. The objects which qualify for the result are assembled via REF relationships and are materialized as composite objects. The connect result is delivered as a so-called composite-object queue (COQ, in analogy to table queues in

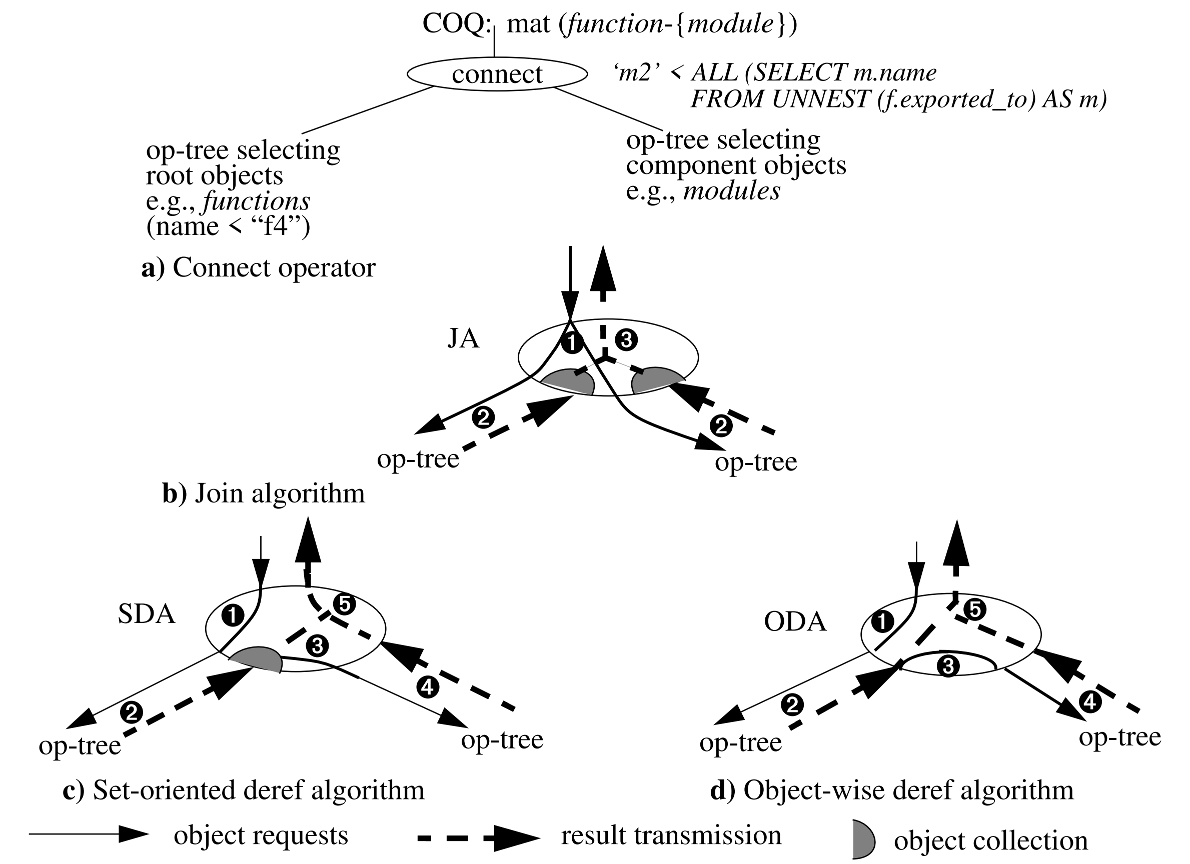



Fig. 5: Connect operator and related algorithms

[MPTW94]) and may be used as input data structure for other relational operators. Especially, such a COQ may act as a root or as a component for a connect at a higher level in the operator tree (see Fig. 6). In addition to block-wise mode, access to COQs is also possible in stream mode in order to support pipelining.

Note that the connect operator is not a symmetric operator and, therefore, is not equivalent to the relational join operator. A directed link from a root to a component contains significant information to be exploited in quantified expressions (see section 3.2)! To effectively evaluate EXISTS and ALL conditions by navigation, all operators in the operator subtree delivering component objects should support selective object requests, that is, the operator invoked should accept a set of OIDs and should be able to selectively construct only the identified component objects. In case that such an operator does not directly refer to a stored table but relies on other operators these OIDs have to be recursively submitted to the related subtrees.

When processing a connect operator, a set of root objects has to be accessed in a first step thereby evaluating a selection predicate, e. g., *function* objects satisfying (name < "f4") in Fig. 5a. Depending on the availability of access paths and other computing resources as well as on the anticipated predicate selectivity, the delivery of qualified roots could be either performed sequentially or in parallel. When a qualified root contains a set of references to components, parallel navigation can be applied to fetch the related components, e. g. *modules*. As illustrated in Fig. 5a, the connect operator assembles qualified root objects (of type *function*) with their (via *exported_to*) referenced component objects (of type *module*) satisfying the ALL predicate and materializes the corresponding result objects in a COQ. The notion "mat (*function*- {*module*})" is used to indicate that the operator materializes objects each consisting of a *function* and its related *module* objects.

In the following subsections, we will discuss algorithms for the connect operator. While navigation allows for new algorithms to directly follow references and materialize result tuples, join-like connect algorithms performing value-based evaluation are also applicable. Both types of algorithms (navigation vs. join), however, embody totally different execution and performance characteristics.

### 4.1 Join Algorithm (JA)

Considering references as regular attribute values makes it possible to apply principles of relational query processing. For example, by providing a few extensions to the proposal described in [SC90, LDM93], it is possible to implement the connect operator as a value-based join operator[7]. In this case, the connect operator simultaneously invokes the evaluation of both operands. The query engine selects the set of all root objects and the set of all component objects satisfying the given conditions (❶ and ❷ in Fig. 5b). Depending on the resources available, both collections, the root and the component objects, could be fetched in parallel. After having received both object collections as input sets, the connect operator evaluates the references thereby attaching component objects (*modules*) to their referencing root objects (*functions*) and returns the set of result tuples in a COQ (❸ in Fig. 5b).

Join-like algorithms outperform navigation in cases where selective requests of referenced objects do not gain any performance advantage, e. g., when naive processing of large object sets is optimal. The reason for this performance difference is that join algorithms are able to optimize the I/O behavior during query execution. Moreover, the query execution can exploit inter- as well as intra-operator parallelism in the "relational" way.

### 4.2 Deref Algorithms

In order to make all navigation strategies applicable (see Sect. 3), a suitable deref algorithm is needed to directly access referenced objects. Such algorithms for the connect operator always rely on a 5-phase execution model:

❶ request root objects (e. g. *function* objects satisfying the given condition);
❷ retrieve all qualified root objects (exploiting parallel access if possible and beneficial);
❸ receive root objects, select their references to the component objects needed (e.g. *module* objects), and invoke the selection of component objects;
❹ use a set of references and retrieve all qualified component objects (exploiting parallel access if possible and beneficial);
❺ receive component objects, construct result objects, and evaluate (partial) conditions (e. g. *'m2' < ALL (SELECT m.name FROM UNNEST (f.exported_to) AS m)).*

[7] The actual implementation of the join algorithm (e. g. nested loop, sort-merge, or hash join) is irrelevant for our discussion.

For this execution model, we introduce the following two generally applicable algorithms which differ in the number and in the time of component object requests in phase ❸.

#### 4.2.1 Set-oriented Deref Algorithm (SDA)

In this approach, the five phases of the execution model remain strictly separated. Even if internal parallelism is exploited in the phases ❷ and ❹, query processing has to be synchronized at the end of these phases. In the initial two phases, all qualified root objects are selected, e. g. via scans or appropriate access paths. Subsequently, the referenced component objects of all root objects are fetched in a **single request** with a potentially large list of OIDs ❹, before the result objects are aggregated in phase ❺ (see Fig. 5c). This scenario is similar to relational semi-join processing. However, instead of accessing all available component objects having matching values in the join attribute we directly request only such objects that are referenced by qualified roots.

The most important advantage of this processing scheme is that all referenced objects can be retrieved in a single set-oriented operation. As a consequence, every page storing some of the referenced objects has to be accessed exactly once. At the same time, the connect algorithm exhibits five different phases of dynamic behavior or resource consumption. While the first one only prepares a request and is very short, the second phase comprises an I/O-intensive operation to fetch the root objects. After that, the set of references to component objects is derived which implies some computational overhead. The next phase is again I/O-intensive since all referenced component objects have to be fetched. And finally, composite object construction and condition evaluation is again CPU-intensive.

Especially when (expensive) user-defined functions are invoked within a query, an interleaving of CPU- and I/O-intensive processing steps is desirable. However, the SDA execution model prevents any kind of inter-operator parallelism among the different phases. Instead of enabling interleaved execution in phases ❸ - ❺, SDA always requests all referenced component objects at a time. Therefore, it does not offer the flexibility for iterative expression evaluation on subsets of component objects (see section 3.2).

#### 4.2.2 Object-wise Deref Algorithm (ODA)

By processing every root object separately, we gain the flexibility of interleaving processing phases and may avoid these disadvantages of set-oriented processing. The query engine controls condition evaluation and component object requests for every root object independently. For each qualified root object, a separate request is invoked with a typically short list of OIDs ❹ resulting in **multiple requests** (n) to fetch all component objects needed for the connect invocation. The access strategy within a request can be chosen depending on the quantifier of a given expression (see Sect. 3.2), that is, all referenced objects may be requested in a single or in multiple invocations. For each qualified root object, phases ❸ - ❺ are executed in a loop until qualification or disqualification is decided. This object-wise pipelining scheme (see Fig. 5d) allows for inter-operator parallelism among the connect operator and the operators for fetching and materializing component objects thereby interleaving the execution of CPU- and I/O-intensive operations and offering the full flexibility for query execution (see Sect. 3).

However, ODA may lead to a huge amount of individual requests for component objects. Since these requests are context-free, they may repeatedly refer to the same page, thereby provoking extra I/Os. As already known from [Yao77], the probability for repeated access to a single page increases rapidly with the number of objects stored within this page. Of course, clustering referenced objects, i. e., objects referenced by the same root object and stored in the same or consecutive pages on disk, can improve such situations. Moreover, a careful implementation of the algorithm can ensure that every object will be fetched only once even if it is referenced from multiple objects. For example, an extra data structure may be maintained to keep information about every object already delivered. Although such a mechanism burdens the connect execution with extra costs, experiences with our implementation in the PRIMA project have shown [Ge97b] that the flexibility gained can more than compensate for these costs.

### 4.3 Execution of Operator Trees

So far, we have only described the processing schemes of a single connect operation. In the following, we discuss how the navigational processing scheme can be extended to multiple navigation phases. We call a connect operator invoking another connect a "super-connect". An invoked connect operator producing input for the super-connect is called "sub-connect". To explain the underlying principle, it is sufficient to refer to a two-level navigation, e. g. from *function* to *module* and then to *program,* as specified in the following query:

*SELECT f.*, f.exported_to -> *, f.exported_to->linked_by -> **
*FROM function f*
*WHERE EXISTS (SELECT * FROM UNNEST (f.exported_to) AS m WHERE*
*(EXISTS (SELECT * FROM UNNEST (m->linked_by) AS p WHERE p.name > 'p2')))*

Obviously, since the connect operation has to observe the complex-object structure, it is not a commutative operation. Fig. 6 describes the two possible operator trees for such a 'nested' navigation.

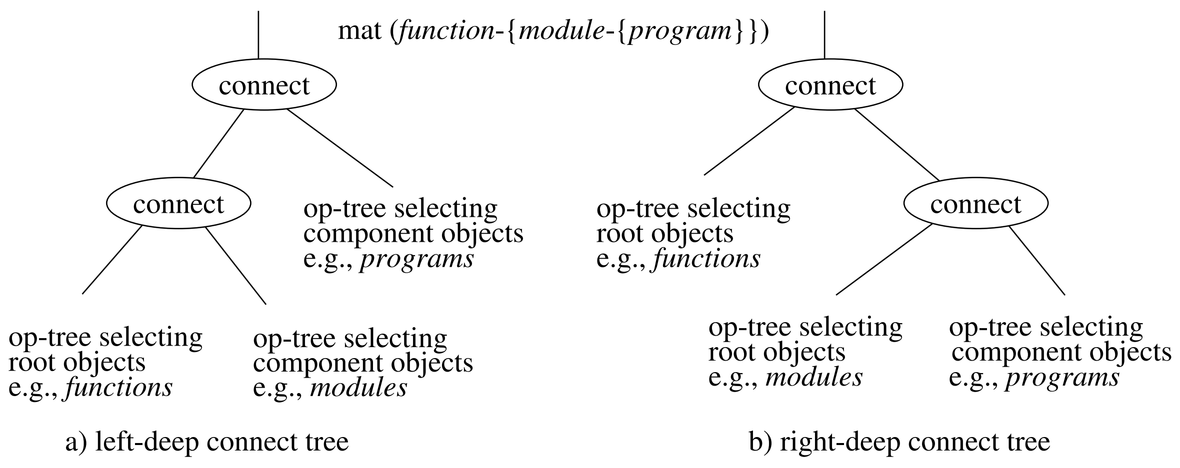



Fig. 6: Combination of multiple connect operators

Although not commutative, JA allows for a symmetric execution, i. e., root and component objects can be processed in parallel or in some order. JA does not exploit any knowledge about reference semantics, but reads all component objects before connecting them to the root objects. As a consequence, component objects can be excluded only by the connect operator; only such components matching a referencing root will be materialized to a result tuple and, hence, delivered to the COQ. In general, the execution of the two operator trees shown in Fig. 6 can exhibit a completely different performance behavior because the number of sub-connect result tuples may substantially vary.

Using a deref algorithm, the set of references determines the number of components to be delivered. Therefore, deref algorithms only derive components that are needed in later processing steps. In both examples in Fig. 6, after having read *functions*, the execution proceeds with the selection of component objects and ends with requests for referenced *programs*. The two plans differ in the point of time to accomplish the connection between *functions* and *modules*. Following the execution plan in Fig. 6a, these connections will be materialized immediately after having read the referenced *modules*. For a given RT-NS, referenced *programs* will not be requested before all references to *modules* are materialized. In contrast, after having read *function* objects in the execution plan of Fig. 6b, we invoke the sub-connect operator thereby passing OIDs for super-connect component objects. Furthermore, these OIDs will be passed on to the operator tree selecting identified *module* objects. The sub-connect can again exploit JA, SDA, or ODA to materialize the composite objects requested.

Note that the super-connect in Fig. 6a cannot materialize references between *modules* and *programs* before root objects appear in the COQ of the sub-connect, that is, first the entire root objects have to be constructed before the first *program* objects can be connected. Similarly in Fig. 6b, a component object has to be materialized completely before it can be connected to its referencing root. This is a restriction that is inherent to our connect operator.

Although our algorithms cover a wide range of powerful navigation strategies, in a few cases, more flexible execution scenarios are desirable, e. g. with two nested ALL quantifiers (e. g., *'m2' < ALL (SELECT m.name FROM UNNEST (f.exported_to) AS m)* AND *'1999/5/5' < ALL (SELECT p.date FROM UNNEST (f.exported_to->linked_by) AS p)*). For such a situation, a nested-loop approach with iteration on *modules* in an outer loop and iteration on *programs* in an inner loop is the optimal processing mode according to our criteria (see Table 1 in Sect. 3.3, variable requests for referenced objects). However, algebra operators require the evaluation of the entire result (which contains all referenced component objects) at a time. Therefore, the outlined nested-loop algorithm cannot be accomplished by (nested) connect operations (see Fig. 6), since the super-connect in both operator trees would expect pipelining of partial results from the sub-connect. The same holds for all other suggested binary objects that assign referenced component objects to referencing root objects, e. g., the expand operator [KM90] or the materialize operator [BMG93]. Only more powerful operators like the CSM in PRIMA [Sch93] or assembly in [KGM91], which exploit additional structural knowledge of objects and expressions on referenced component objects, can offer such flexible execution mechanisms.

### 4.4 Summary

We have discussed parallel navigation algorithms. For this purpose, we have introduced the binary connect algebra-operator which is a kind of asymmetric join operator. All three algorithms use a single request to select the root objects whereas they behave differently in the number of requests and their invocations for the component side. JA uses a single component selection without any OID restriction, therefore, root and component selection can be performed in parallel. SDA is characterized by a single component selection restricted by a list of OIDs which implies that access to the component side has to be delayed until the end of the root side selection. ODA has the most complex structure and, in turn, the greatest flexibility. Access of the component objects is performed for each qualified root object separately which requires n selection requests for the component side. Each of these requests can be scheduled as soon as the OID list is derived from the related root object thereby allowing pipelined execution of the various component side selections.

As compared to JA, deref algorithms allow for better adaptation to given query predicates. Object-wise processing (ODA) is very flexible and, as a consequence, makes it possible to minimize CPU resource consumption for complex predicate processing, whereas set-oriented processing (SDA) allows the minimization of disk I/O. Finally, we have exemplified the restrictions stemming from the use of a binary operator. Our connect operator only considers single reference attributes and evaluation of conditions on materialized objects. These limitations could be overcome only by semantically richer operators. Nevertheless, in most situations the introduced connect operator offers sufficient flexibility for navigational query processing.

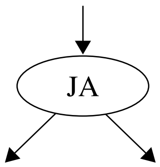
## 5 Use of Navigation Algorithms

So far, we have described various navigation strategies for query processing as well as three different connect algorithms. In order to integrate and implement these concepts in ORDBMSs, they have to be refined. Especially important questions are how to use them in QEPs and how to enable the optimizer to perform the adequate processing decisions. To shorten our discussion, we refer to the well-known principles of complex-object query processing [HMS92] and runtime parameter estimations [Gr93]. Since the optimization know-how of relational algebra operations can be directly applied, the key point is the use of connect algorithms in an query evaluation framework. For this reason, we only discuss the issues of single connect and nested connects in more detail. Starting with a single connect we show to which extent well known RDBMS optimization principles can be adapted to our connect algorithms. In addition, we investigate more thoroughly those algorithms which do not have equivalents in RDBMSs. Finally, we extend our presentation to QEPs with nested connect operators.

### 5.1 Single Connect Operator

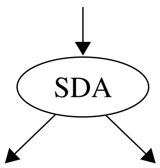
For the three connect algorithms (JA, SDA, ODA), we will discuss their integration into a query evaluation framework. To facilitate the (complicated) matter, we start with optimization aspects of a single connect operator.

#### 5.1.1 Join Algorithm (JA)




Because JA does not exploit reference semantics but uses the OIDs only for value-based comparison, it is very closely related to relational join processing. In principle, root object processing, component object processing, as well as result construction can be done in parallel. The most important difference to relational query processing is that optimization and execution have to deal with multi-valued references. Therefore, it is more difficult at query optimization time to estimate accurate result set sizes. A detailed description of an execution and optimization framework can be found for example in [LDM93].

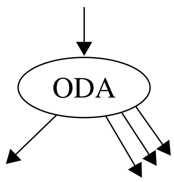
### 5.1.2 Set-oriented Deref Algorithm (SDA)




From an optimizer's point of view, SDA is very similar to JA. Root object construction is exactly the same, also performed within a single operator invocation. The "component object operator" normally processes smaller object sets, i.e., only those objects that are referenced by at least one root object. Hence, it is the task of the optimizer to estimate the number of referenced components. This corresponds to the determination or estimation of the number of join partners in JA.

Parallel execution is not an issue here because SDA inherently requires sequential processing of root and component objects.

### 5.1.3 Object-wise Deref Algorithm (ODA)




Again, a single invocation of our connect operator induces a single call to the "root operator" and a single COQ as input. The number of referenced component objects to be requested is the same as with SDA and corresponds to the number of join partners with JA (if early qualification or disqualification are not applied). However, the execution characteristics are different as compared to SDA and JA, because ODA leads to multiple independent requests for component objects. These requests can be executed concurrently as well as in parallel to root object processing and result construction.

ODA directly exploits OIDs for object-wise requests which introduces several new aspects for the cost estimation. If available, clustering plays an important role. Clustered storage structures for RTNSs (or clustered substructures holding root/component-related objects) ideally support object-wise processing, because they provide locality of page references for iterative object processing in the DB buffer. However, physical clusters can only represent disjoint RTNS structures in a natural way. Physical clustering of overlapping RTNSs creates high degrees of object replication [SS89]. Moreover, physical clustering of predefined RTNSs is counterproductive if these clusters do not match the RTNSs dynamically defined in a query. Especially in the case of bidirectional links, the number of valid RTNS sets definable in queries increases rapidly and the probability, in turn, decreases that a given RTNS cluster matches the scope of a given query. If available, clusters matching the required RTNSs greatly improve query evaluation. However, it is more realistic to anticipate type-wise clustering (if any) which can support JA and SDA as well. For example, type-wise clustering corresponds to clustering tuples of the same table in relational systems.

As already mentioned, object-wise pipelining may require multiple accesses to the same page because of independent and context-free invocations in an operator tree. This undesired reference behavior can be avoided by using a single operator invocation which extracts all relevant objects in a page at once. Of course, the number of page requests is not a critical performance issue when processing data-intensive applications. Rather, the number of page transfers (physical I/Os) to the DB buffer is more indicative. To describe such disk-I/O behavior, [Ge97a] has developed a cost model that covers parallel, set-oriented, and context-free requests for objects of a single type. Furthermore, the cost model has been validated by comparing estimated I/O numbers to those observed in a real system. The main observation of this experiment is that the number of extra I/Os provoked by repeated page references can increase substantially. The amount of extra I/Os does not only depend on the buffer size, but also on the

- number of objects stored in a single page,
- fraction of objects to be retrieved from all objects of that type in a database,
- number and size of individual operations requesting objects, and
- degree of parallelism, i. e., how much these operations overlap in time.

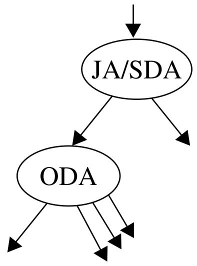
Another result described in [Ge97a] is: Only if navigational object requests are grouped in a small number of parallel operator invocations, the amount of page requests from disk decreases substantially and becomes almost as low as in the case of a single invocation. In any other case, the number of extra I/Os remains almost as high as for sequential processing of single object requests, more or less independently of the degree of parallelism. Only if very few objects are selected, the amount of extra I/Os is negligible, due to the low probability that object requests are directed to the same pages.

In object-relational query processing, both I/O and evaluation of complex predicates can become a bottleneck for query evaluation. While join and set-oriented processing can perfectly optimize I/O behavior, context-free requests of ODA allow for parallel processing of both CPU-intensive calculations as well as pipelining between I/O- and CPU-intensive tasks. These context-free requests compete for limited resources, thereby possibly causing additional resource consumption (e. g. I/O). As shown in [Ge97b], pipelining in the presence of context-free object requests may substantially increase physical I/O. But, on the other hand, this competition can lead to much better resource utilization. That means, the execution dynamics of connect algorithms not only exhibits strongly varying I/O as well as CPU requirements but also influences simultaneous processing of other operators (except when operators are executed sequentially). Due to such load dependencies (available buffer size, precise operation execution timing, etc.), accurate a priori optimization and parallelization are very limited. Moreover, improper cost estimation for user-defined functions magnifies this problem.

### 5.2 Nested Connect Operators

So far, we have explained execution characteristics and drawbacks on how to estimate execution costs limited to single connect operators. To generalize our results, we now discuss the use of nested navigation which may lead up to n-level connects. In principle, hierarchical nesting may create three different patterns already known from relational join processing: left-deep, right-deep, and bushy. Note the combination of connect operators is orthogonal, that is, each of our navigation algorithms can be chosen independently.

### 5.2.1 Left-deep tree

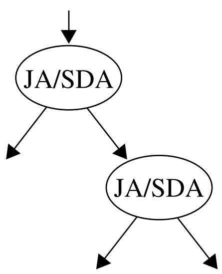



In the case of left-deep nesting, component objects of connect operators are always elementary (not complex) objects while root objects can be complex, i.e., constructed by a connect. Root objects become more and more complex with every connect they are passed through as input. For all connect algorithms, an invocation of a super-connect creates a single request for root objects, i.e., a single sub-connect only. Therefore, a single root-object input queue is sufficient for every connect operator. The number of component-object input queues depends on the chosen algorithm. JA and SDA only need a single queue, whereas the number of queues for ODA is determined by the number of incoming root objects. However, the number of component-object input queues in a specific connect does not affect these numbers in other connects.

On the other hand, the algorithm for one connect influences the resources (I/O, CPU) remaining available for other connects. While JA can utilize resources for root- and component-object processing in parallel, SDA requires all root objects to be constructed before component objects are accessed. As already explained, resource and cost estimations for ODA are very sophisticated because they heavily depend on runtime parameters. The problem increases with the number of parallel connects executed by ODA.

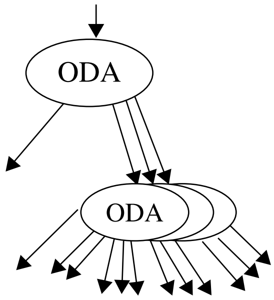
The degree of parallelism between connect operators in a left-deep operator tree depends on the chosen algorithm. While JA and ODA enable parallel execution of sub-operators as well as pipelining between super- and sub-connects, SDA is limited to strict sequential execution. When SDA is chosen for a sub-connect, its resulting COQ is filled with first results very late as compared to JA and ODA. This has no impact on the overall execution, when SDA is also used in the super-connect (which then does not continue with component object requests before all root objects are read). Using JA for the super-connect, component objects can be processed independently. However, late arrival of root objects causes late construction of the first result object in the super-connect as well. When using SDA for the sub-connect and ODA for the super-connect, root objects arrive late but with a high frequency in the super-connect. As a consequence, processing of component objects starts late with many parallel requests.

### 5.2.2 Right-deep tree




At first, let us assume that the super-connect is executed using SDA. Then, the root object processing of the super-connect is finished before the sub-connect is invoked. In this case the sub-connect can be considered almost independently from its super-connect. There is only the result construction phase left in the super-connect when component objects arrive from the sub-connect.

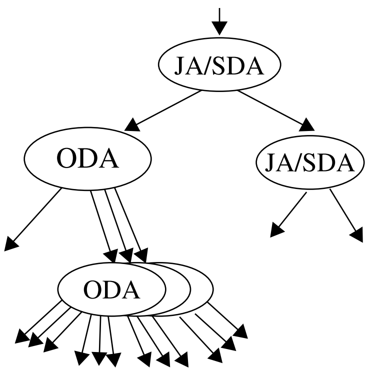
When JA is applied in the super-connect, the situation is exactly the same, if root and component objects are constructed sequentially. If both are independently processed in parallel, the only difference for the sub-connect may be a limited availability of resources.




Using ODA in the super-connect all problems already described for a single connect operator arise because ODA generates many parallel connect execution instances. Each of these execution instances constructs a set of component objects referenced. When ODA is used for the sub-connect as well, the inaccuracy of the cost estimation “explodes“ because every sub-connect, in turn, fetches its root and component objects in separate requests.

The potentially vast number of component objects demands for a query execution architecture where multiple requests can be handled by a single execution thread. This is necessary to keep administration overhead as low as possible.

### 5.2.3 Bushy tree




In the bushy case, operator trees for root objects as well as for component objects may contain further connect operators. Such operator trees show both types of problems already described for left- and right-deep trees: difficult prediction of execution costs as a function over time as well as rapidly growing inaccuracy for cost estimations.

## 5.3 Summary

We have explored the use of navigation algorithms in a query optimization framework with special emphasis on their parallel execution. For JA, we can adopt the well-known principles of query optimization and execution used in RDBMSs. For SDA, we can apply the same mechanisms, however, SDA execution only allows for sequential processing of sub-operators. For ODA, prediction of execution costs is only possible having accurate knowledge of object clustering and temporal behavior of independent component requests because it is very sensitive to these parameters. Thus, reliable cost estimation for nested connects with ODA is almost impossible before runtime.

Since “regular“ join algorithms are generally applicable for connect operations, our two navigation algorithms increase the number of possible QEPs substantially. If j is the number of available join algorithms and n the number of connect operators in a given operator graph, the number of conceivable QEPs grows from $j^n$ to $(j+2)^n$. As a consequence, early pruning of inefficient execution plans becomes even more important than in relational systems.

# 6 Optimizing Navigation - a Simplified Approach

In the previous sections, we have seen that ODA complicates a priori query optimization at compile time, because cost estimations become much more unreliable and the number of feasible QEPs increases substantially. In the following, we will sketch a simple method for connect algorithms and their execution cost estimates to be applied in an optimizer framework. This method is more intended to visualize the problems than to present a perfect solution. To further simplify the discussion, our cost estimates only refer to I/O demands.

Resource requirements and time overlap of operators are important optimization aspects for parallel execution. In order to discuss and visualize the execution costs for the connect operator, we introduce a simple method which allows for the calculation of resource requirements and timing conditions. For each operator node, we represent the anticipated resource consumption by a rectangle (time t, resources r) approximating the overall cost R. Additionally, a delay parameter d specifies the possible execution start of an operator. Finally, since the applicable algorithms have different resource requirements and different starting times of suboperators, a fourth parameter i determines the select algorithm on the right side (JA, SDA, ODA). Hence, a quadruple (t, r, d, i) characterizes a select execution (see Fig. 7, where *functions* are connected to *modules).*

Two simplifying assumptions are necessary to make QEP optimization tractable. As a first simplification, we adopt fixed resource consumption for each operator. Time-varying modeling would complicate the optimization procedure substantially and would not be helpful for obtaining a more realistic runtime characterization [WG93]. In our illustrations, the aggregated cost R of the connect operation is considered to be the sum of the related suboperator costs.[8] After having estimated the expected cardinalities of objects, the optimizer can directly derive the execution costs for object requests[9] (select operators). As a second simplification, we refer to worst-case assumptions for execution time and resource consumption of the connect operator. For example, ODA optimization always assumes pipelined execution, i. e., sequential requests of referenced objects.

For our explanation, it is accurate enough to refer to abstract time and resource units; it is not important whether these factors refer to elapsed time (ns, ms, sec), processor cycles or something else.

## 6.1 Single Connect Operators

Fig. 7 illustrates examples of execution costs for the various connect algorithms graphically. In our example, a single connect operator calls a suboperator selecting root objects (*functions*) whereas another suboperator retrieves the referenced objects (*modules*). As indicated, the suboperator which delivers root objects (*functions*) takes 10 resource units over 7 time units. JA which reads all *modules* in parallel to the select operation on *functions* is assumed to require 5 resource units over 10 time units (10, 5, 0, JA). Since SDA selectively reads only referenced objects, we assume 3 time units. However, it cannot start

[8] Note, since the connect operator itself does not invoke any I/O, its execution costs can be approximated by 0 in our context.

[9] We don‘t want to discuss how these costs are derived. For our purpose, it is sufficient to rely on a simple cost model based on attribute selectivities, page mapping, etc. [Se79], even if more advanced cost models [Sch93] and estimation techniques (e.g. histograms [PI97]) can provide more accurate estimates.

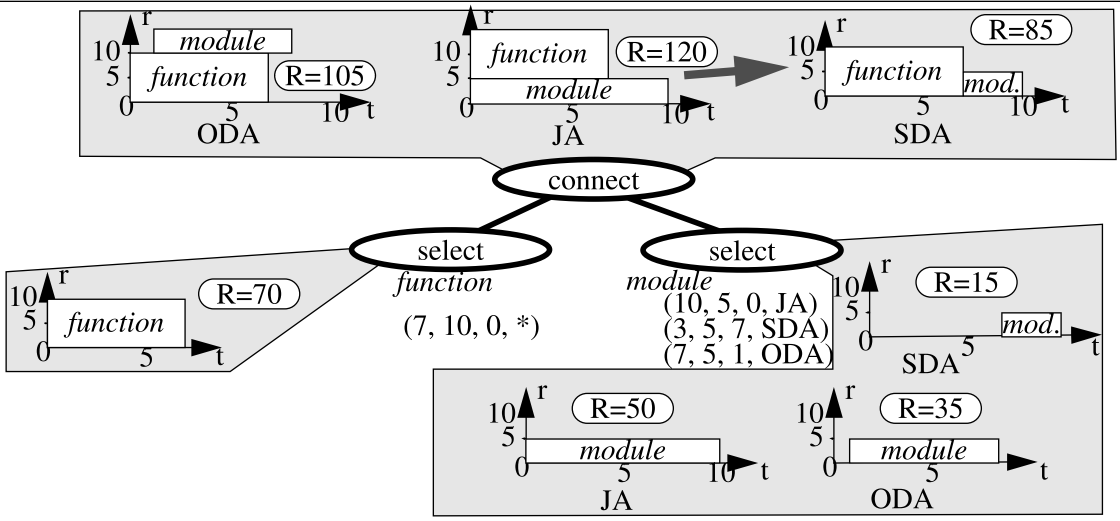



Fig. 7: Single connect QEP - illustration of execution costs

before the *function* select has been finished. Therefore, this implementation is characterized by (3, 5, 7, SDA). Finally, ODA may lead to multiple I/Os of the same page and is therefore supposed to use more resources than SDA, but less than JA. Moreover, it cannot start before the first *function* has been submitted to the connect operator. Therefore, we describe the select operation here by (7, 5, 1, ODA). Hence, the aggregate costs R delivered by the three connect strategies are 105, 120, and 85, respectively.

If the available resources exceed the maximum of the required resources, the graphs directly expose the execution times (response times), that is, 8, 10, and 10 time units, respectively. Otherwise, in situations where the required resources exceed the available resources, our approach assumes a linear dependency between execution time and resources, i. e., we extrapolate the execution time (range-wise for constant resource requirements) by a factor (required_resources/available_resources). As a result, Fig. 8 illustrates some indicative response-time curves for our three navigation algorithms in the given scenario, depending on the amount of available resources. Up to 11.25 resources, SDA results in the shortest response times. Thereafter, ODA exhibits the shortest response times (for available resources r between 10 and 15, response time is 1+(15/r)*6+1 and is less than 10 for r > 11.25).

Fig. 8 shows that JA always delivers the worst response times. Hence as a sub-QEP in a more complex query, JA can never be the optimal choice. No matter how many resources are available, at least one of the other algorithms shows better response times. To exclude such a QEP, a simple heuristics can be used: For a given connect, algorithm A1 is superior to algorithm A2 if predicted response time, total execution costs, as well as maximum resource requirement of A2 are not lower than those of A1, and at least one of these parameter values exceeds the corresponding value of A1.

The parameters to apply this heuristics can be directly obtained from our graphs in Fig. 7. Compare, for example, JA and SDA: the overall resource consumption for JA exceeds that for SDA (120 > 85), the predicted response time is the same for both (10 = 10), and the maximum resource requirement at a time is larger for JA (15 > 10). Therefore, JA can be omitted from further optimization steps. Fig. 7 in-

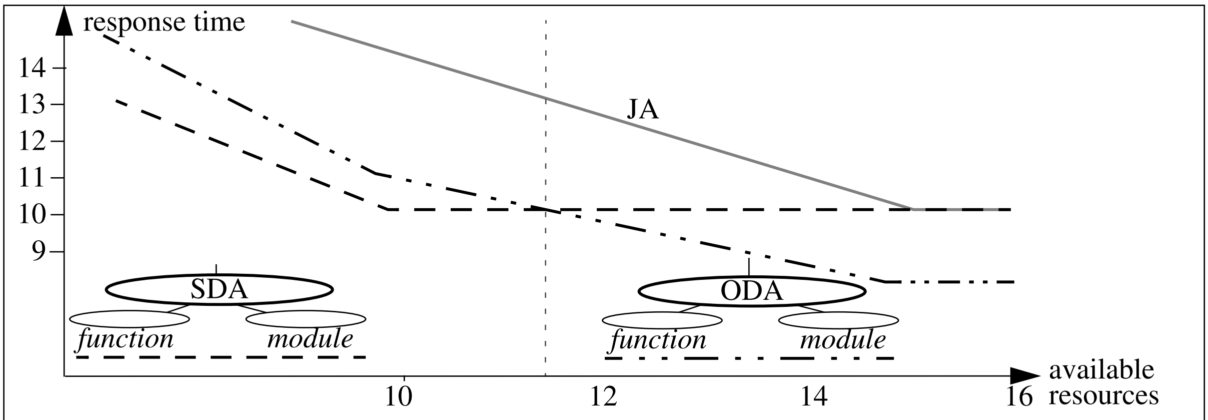



Fig. 8: Resource-dependent execution costs for connect operators

dicates such a dependency by an arrow pointing from JA to the superior SDA. Similarly, ODA outperforms JA. However, it is sufficient to find a single algorithm that outperforms the one to be excluded.

To summarize the discussion: Our approach is able to determine the optimal ranges for the connect algorithms depending on the availability of resources at runtime. In order to be applicable in more complex situations, it has to be extended to the use of nested connects in QEPs.

### 6.2 Nested Connect Operators

So far, we have explained how to estimate execution costs of a single connect operator. To generalize our observations, we discuss the use of nested navigation which may lead up to n-level connects.

Here we sketch the situation only for left-deep patterns thereby starting from our results for single connects in Fig. 8. In order to find all range-wise optimal QEPs, all QEPs which are superior at a given level of nesting for a certain range of available resources have to be extended for the next level of connect nesting with JA, SDA, and ODA respectively. In the general case, a complete 3-way decision tree has to be anticipated which results in the consideration of $3^n$ conceivable QEPs (at compile time). In principle, all their response time curves could be calculated to determine the optimal QEPs and their related ranges. However, to save evaluation overhead, we apply our early pruning heuristics of inferior operator combinations and a level-to-level optimization. At the next level k (1< k <n) we consider only QEPs which have qualified at level k-1 as a superior solution in a certain range of available resources.

To illustrate our proceeding, we extend the example of Fig. 7 by another connect operator assembling *program* objects to intermediate composite objects (mat (*function*-{*module*})) (see Fig. 9). Compared to the single connect QEP, a new aspect is the execution start of the select operation in the super-connect. The corresponding “?“ in Fig. 9 indicates that the starting time for the select operation depends on the sub-connect (whose strategy will be chosen at runtime). Accordingly, its execution time also varies with the sub-connect strategy.

With the assumptions of our introductory example (Fig. 7), we exclude the JA sub-connect from further optimization considerations. As illustrated in Fig. 9, only the “range-wise optimal“ sub-connect QEPs are aggregated to obtain super-connect QEPs. The remaining 6 QEPs in Fig. 9 are again checked by our early pruning heuristics. Highlighted by bold arrows from inferior to superior QEPs, the inferior

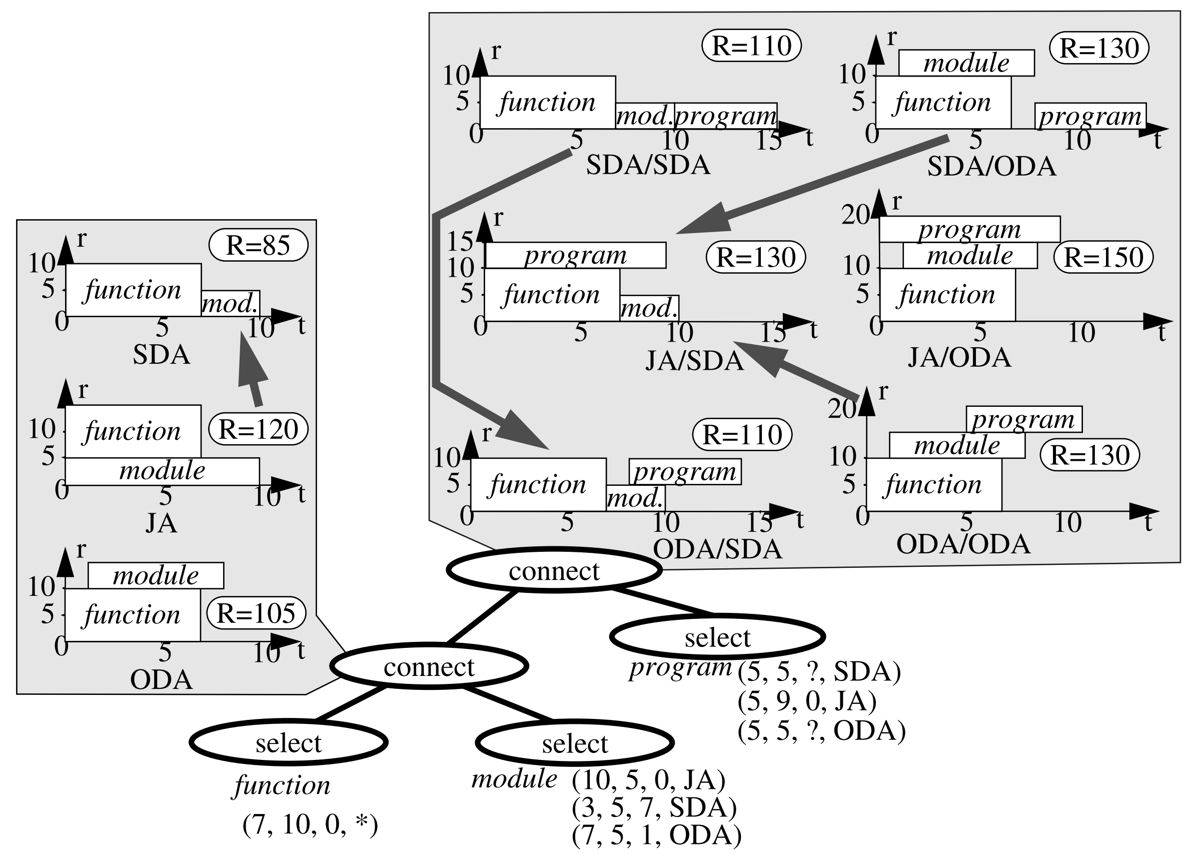


Fig. 9: Nested connect QEP - illustration of execution costs

QEPs embody equal total execution cost, equal or higher maximum resource requirements, and in all cases their predicted response time leads to the pruning. Without loss of optimal solutions, the inferior QEPs can be ignored in subsequent steps.

As illustrated in Fig. 10, resource-dependent execution costs can be derived by the optimizer thereby considering the combined super- and sub-connect resource consumption. Given the calculated response times as a function of the resources available, the optimizer can determine at compile time which nesting of algorithms leads to range-wise optimal combinations. In our example, the combinations ODA/SDA, JA/SDA, and JA/ODA are optimal in the ranges $< 10.5$, $< 17$, and $\geq 17$, respectively.

Based on the resource requirements specified in the QEP and the given amount of free resources at runtime, the appropriate navigation algorithms can be selected.

### 6.3 Dynamic Execution of Connects

As we have pointed out in the preceding sections, cost estimations are based on some necessary simplifications and may be unreliable especially for ODA. As a consequence, results of our optimization process can only serve as a guideline to choose an initial query processing strategy. But once a QEP has started, we cannot guarantee at all that query execution costs anticipated at compile time closely match those observed at runtime. Wrong estimates for intermediate result sizes and worst case assumptions can lead into both directions, under- as well as over-estimation of processing costs. Therefore, we need an

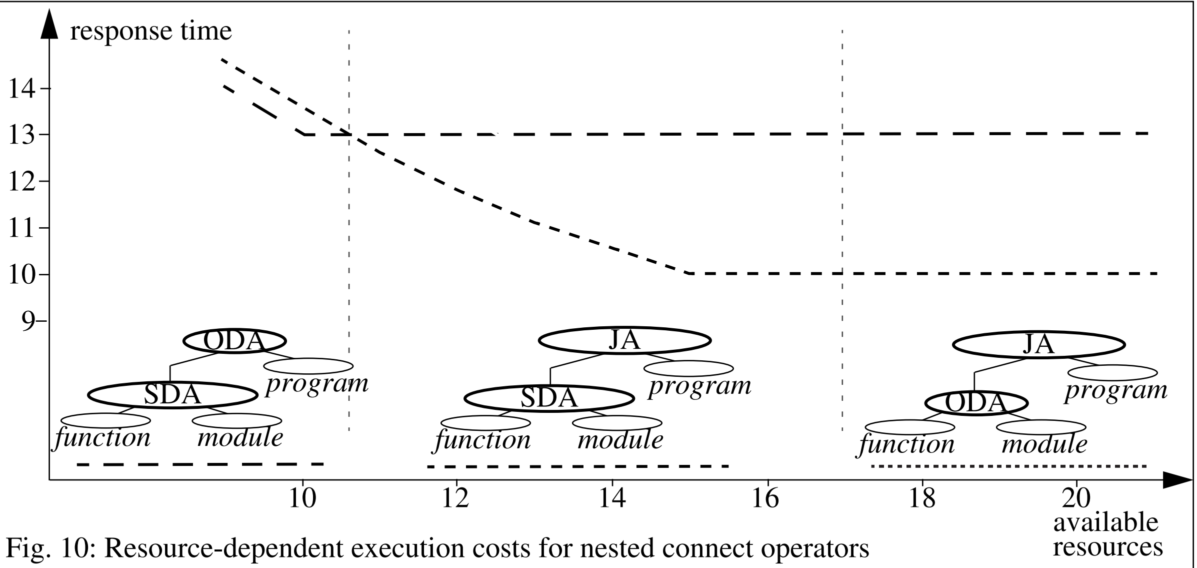


Fig. 10: Resource-dependent execution costs for nested connect operators

adaptive execution strategy that can react on upcoming bottlenecks as well as on idle I/O- and CPU- resources. Fortunately, our deref algorithms are very flexible to adjust to dynamically changing runtime conditions. In the sequel, we will discuss this flexibility for the various connect algorithms.

### 6.3.1 Join Algorithm

Once a connect operator has chosen a join algorithm for its execution, it is almost impossible to dynamically reverse this decision without any penalty. Only if the sub-operators are executed sequentially and if the component sub-operator has not already been invoked, it is possible to switch to our deref implementations. However, these preconditions do normally not hold when JA executes a connect. JA‘s main advantage, the parallel execution of both sub-operators, would be dismissed and, in turn, sequential execution of both sub-operators would make SDA preferable because of selective access to components. Nevertheless, if such a situation occurs, switching to a deref implementation simply means to accept incoming root objects and to request component objects depending on the deref algorithm used. Referring to Fig. 5, when step 2 has been invoked with JA for root objects, execution can continue with steps 3 to 5 of our deref variants.

If JA has already requested component objects, a switch to deref algorithms does not make sense, because this would mean to throw away the work the component sub-operator has already performed.

### 6.3.2 Set-oriented Deref Algorithm

SDA‘s strength is the optimal use of I/O resources, however, it is confined to sequential processing. If it turns out at runtime that I/O- as well as CPU-resources are idle, query processing should enable some degree of parallelism.

Assume that connect execution has already started with SDA and that root objects are still delivered. Then it is easy to switch to JA which basically means to request component objects in parallel, not by exploiting reference semantics but by using value-based processing. Moving from SDA to ODA is trivial, since referenced component objects can immediately be requested instead of waiting until all root objects have arrived. When SDA has already received all root objects and has requested the referenced component objects, then there is no need to switch to any of the other algorithms because processing of incoming components is the same for all algorithms.

#### 6.3.3 Object-wise Deref Algorithm

ODA allows for the highest degree of parallelism and the highest flexibility of navigation. At the same time, it consumes the largest share of I/O resources. If it turns out that I/O is a bottleneck, shifting to another algorithm can be advantageous.

Assume connect execution follows ODA and we want it to change to SDA. This simply means to delay component object requests for newly incoming root objects and to buffer them in the same way as with SDA. Further component requests will not be issued before all roots have arrived. Of course, SDA would only request components for root objects that have arrived after the change. In general, a shift to JA is not helpful, because some component objects have already been requested and potentially delivered with ODA. Because JA does not exploit reference semantics, these component objects would be materialized and delivered again.

### 6.4 Summary

We have discussed the selection of navigation algorithms during query optimization. It turned out that accurate cost estimation for various connect algorithms, including pipelined execution of operators, cannot be calculated before runtime. Since their resource requirements depend on the actual interleaved execution order, even their runtime cost estimates remain very inaccurate.

For this reason, query optimization at compile time can only rely on worst-case assumptions, i. e., sequential execution of independent context-free object requests. Starting from this assumption we have outlined how to model the execution of a single connect operator and we have also extended this model to the processing of nested connect operators. Moreover, we have shown how to determine the optimal connect plan at runtime, thereby taking actual load characteristics into account. So far, our optimization approach considers only I/O resources. Nevertheless, the same approach can be used to model the CPU behavior in order to gain more accuracy.

Because of the inherent inaccuracy of cost estimations for ODA at query optimization time, we decided to apply worst case predictions. We feel, however, that there is a need at query execution time to select among a small number of feasible QEPs or even to switch from one connect algorithm to another one. ODA and SDA easily allow for such dynamic changes.

## 7 Summary and Outlook

Direct modeling and implementation of multi-valued references in object-oriented and object-relational database systems enables a new kind of operation for query processing called navigation which allows for direct access to referenced objects. In contrast to user-implemented, application-specific navigation through a database, we have primarily considered strategies for system-supported navigation for query processing which is invisible to applications. First of all, we have discussed various strategies to exploit references and navigation to optimize query processing. These strategies depend both on RTNSs, i. e., the result tuple navigation scopes, and on given predicates. Navigation makes much more execution strategies possible than simple value-based join operations do. After having presented various navigation scenarios, we have outlined the advantages and disadvantages of the connect operation and its most important implementations. Our deref algorithm reveals its strengths when processing small sets of objects or computationally expensive predicates, whereas the classical value-based join is only useful when a large set of referenced objects is processed. Navigation can keep RTNSs as small as possible, thus preventing retrieval of referenced but not needed component objects. Moreover, processing strategies adjusted to quantifiers and logical connections of predicates can avoid unnecessary processing steps. However, accurate a priori cost modeling is impossible for ODAs which enable pipelining and processing of very small data granules. I/O estimations strongly depend on the actual system load (CPU and I/O) as well as on the actual execution orders of the (parallel) execution steps. Therefore, our cost estimations and the presented optimization framework are based on a worst-case execution model. At the compile time, the optimizer generates information to be exploited at runtime in order to determine the optimal load-dependent algorithm.

The worst-case assumption may lead to an overestimation of the actually required resources. To effectively exploit the available resources and to gain high flexibility during query execution, we provide a dynamic adjustment of QEPs at runtime, that is, different connect algorithms may be selected by dynamically switching operator trees. Thus, in contrast to [Gr94], dynamic query execution does not only mean to determine optimal execution algorithms at query time or operator start-up time, but also to replace the algorithm because of wrong (worst-case) estimates at optimization time or because of workload changes. However, due to space limitations, we cannot go into more detail.

The work described in this paper presents only a small excerpt of our effort in the area of complex objects [Ge97b]. As outlined here, we process expressions which are quantified on referenced component objects. However, in a highly meshed database, complex objects or component objects may overlap. Query results may contain overlapping structures. Therefore, descriptive modification operations on complex structures and aggregation within descriptive operations have to realize such overlapping, so that unexpected results can be prevented. With projections on referenced component parts, value-based identification of overlapping structures is impossible. As a consequence, object identities have to be maintained while processing complex structures. Therefore, modeling as well as processing extensions have to be investigated that cover the representation of complex structures with overlapping structures of heterogeneous elementary objects. Experiences with our PRIMA system confirm that such processing extensions substantially increase the demands for CPU resources and that, in turn, applications directly benefit from the use of parallel navigation.

Integration of user-defined functions, methods, stored procedures, etc. make cost estimation for query optimization purposes more complicated and less reliable. For this reason, query optimization can no longer rely on a comparatively small number of well understood, simple, and value-based operations on large homogeneous data sets (tables). With the enhancement of modeling capabilities, predictions about processing characteristics of a query become more and more inaccurate. Furthermore, parallel execution of such functions, which themselves can be very expensive in terms of CPU as well as I/O requirements, magnify these problems. The more application semantics the user can integrate into code to be executed during query evaluation, the more the system has to be able to react on varying execution patterns. As shown here, this even holds for the load-dependent behavior of deref algorithms. Therefore, extensibility in ORDBMS query processing has to be complemented by adequate flexibility provided by the processor platform. Most importantly, systems should utilize smaller execution units than current systems do. However, the administration overhead will be substantially higher than when compared to conventional process concepts.

Finally, our discussion did not consider the location of objects in a distributed system. In cases of distributed object storage, object-wise deref navigation may be inappropriate due to the huge communication costs. However, these communication costs have to be balanced with the execution costs when requesting larger sets of objects which do not contribute to the query result.


## Acknowledgements

We would like to thank B. Mitschang, N. Ritter, and N. Zhang for their helpful comments and suggestions on an earlier version of this paper.


## References


BMG93 Blakeley, J., McKenna, W., Graefe, G.: Experiences Building the Open OODBM Query Optimizer, Proc. ACM SIGMOD Conf., 1993, 287-296.

Ch96 Chamberlin, D.: Using the New DB2: IBM's Object-Relational Database System, Morgan Kaufmann, 1996.

De96 DeWitt, D.: Parallel Object-Relational Database Systems: Challenges & Opportunities, invited lecture at the 4th Int. Conf. on Parallel and Distributed Information Systems, 1996.

DG92 DeWitt, D., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems, Comm. of the ACM 35:6, 1992, 85-98.

DNSV96 DeWitt D., Naughton, J., Shafer, J., Venkataraman S.: Parallelizing OODBMS traversals: a performance evaluation, The VLDB Journal 5:1, 1996, 3-18.

Ge95 Gesmann, M.: Strategies for the Dynamic Construction of Complex Objects in Query Processing (in German), Proc. GI-Fachtagung “Datenbanksysteme in Büro, Technik und Wissenschaft“ (BTW), Informatik aktuell, Springer-Verlag, 1995, 282-301.

Ge97a Gesmann, M.: A Cost Model for Parallel Navigational Access in Complex-Object DBMSs, Proc. 5th Int. Conf. on Database Systems for Advanced Applications (DASFAA), 1997, 1-10.

Ge97b Gesmann, M.: Parallel Query Processing in Complex-Object DBMSs - Processing Concepts, Implementation Aspects, and OS embedding (in German), Shaker, Aachen, 1997.

Gr93 Graefe, G.: Query Evaluation Techniques for Large Databases, ACM Computing Surveys 25:2, 1993, 73-170.

Gr94 Graefe, G.: Volcano, an Extensible and Parallel Query Evaluation System, IEEE Trans. Knowledge and Data Engineering 6:1, 1994, 120-135.

HMMS87 Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, Proc. 13th Int. Conf. on Very Large Databases, Brighton, 1987, 433-442.

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

IBM99 IBM DB2 Universal Database (Version 6.0), IBM Corp., 1999.

Informix99 Informix Dynamic Server 2000, Reference, Informix Software, Inc., 1999.

KGM91 Keller, T., Graefe, G., Maier, D.: Efficient Assembly of Complex Objects, Proc. ACM SIGMOD Conf., 1991, 148-157.

Ki96 Kim, W.: Object-Relational — The Unification of Object and Relational Database Technology, Whitepaper, UniSQL Inc., Austin, 1996.

KM90 Kemper, A., Moerkotte, G.: Advanced Query Processing in Object Bases Using Access Support Relations, Proc. 16th Int. Conf. on Very Large Databases, Sidney, 1990, 290-301.

LDM93 Lieuwen, D., DeWitt, D., Mehta, M.: Parallel Pointer-based Join Techniques for Object-Oriented Databases, Proc. 2nd Int. Conf. on Parallel and Distrib. Inf. Systems, 1993, 172-181.

Mi89 Mitschang, B.: Extending the Relational Algebra to Capture Complex Objects, Proc. 15th Int. Conf. on Very Large Databases, Amsterdam, 1989, 297-305.

MPTW94 Mohan, C., Pirahesh, H., Tang, W. G., Wang, Y.: Parallelism in Relational Database Systems, IBM Systems Journal 33:2, 1994, 349-371.
ODMG97 The Object Database Standard: ODMG V2.0, R. Cattell (ed.), Morgan Kaufmann, 1997.
OHUS96 Olson, M.A., Hong, W.M., Ubell, M., Stonebraker, M.: Query Processing in a Parallel Object-Relational Database System, IEEE Data Engineering Bulletin 19:4, 1996, 3-10.
Oracle98 Oracle8 Sever SQL Reference, Oracle Corp., June 1998.
Pe96 Peithner, K.: Optimization of Declarative Queries in Objectbases (in German), infix, 1996.
PI97 Poosala, V., Ioannidis, Y.: Selectivity Estimation Without the Attribute Value Independence Assumption, Proc. 23th Int. Conf. on Very Large Databases, Athens, 1997, 486-495.
SB99 Stonebraker, M., Brown, P.: Object-Relational DBMSs — Tracking the Next Great Wave, Morgan Kaufmann, 2nd Edition, 1999.
Sch93 Schöning, H.: Query Processing in Complex-Object DBMSs (in German), DUW - Deutscher-UniversitätsVerlag, Wiesbaden, 1993.
SC90 Shekita, E., Carey, M.: A Performance Evaluation of Pointer-Based Joins, Proc. ACM SIGMOD Conf., 1990, 300-311.
Se79 Selinger, P. et al.: Access Path Selection in a Relational Database Management System, Proc. ACM SIGMOD Conf., 1979, 23-34.
SQL99 IANSI-X3H2-98-525/WG3: FRA-007: Database Language SQL - Part2: Foundation (Working Draft), Sept. 1998.
SS89 Schöning, H., Sikeler, A.: A Cluster Mechanism Supporting the Dynamic Construction of Complex Objects, Proc. 3rd Int. Conf. on Found. of Data Organization and Algorithms, 1989, 41-46.
WG93 Wilschut, A., van Gils, S.: A Model for Pipelined Query Execution, Proc. of the MASCOTS93 Symposium, San Diego, CA., 1993.
Ya77 Yao, S. B.: Approximizing the Number of Accesses in Database Organizations, Comm. of the ACM 20:4, 1977, 260-261.