Rule-Based Parallel Query Optimization for OQL Using a Parallelism Extraction Technique

Sandra F. Mendes
Departmento de Computação, Campus do Pici
Universidade Federal do Ceará, Fortaleza, Brazil
E-mail: sandra@lia.ufc.br

Pedro R. Falcone Sampaio
Computer Science Dep., Oxford Road
University of Manchester, M13 9PL, UK
E-mail: sampaiop@cs.man.ac.uk

Abstract

This work deals with query optimization for parallel object-oriented databases using rule-based optimizers. The solution enacted departs from an existing non-parallel database query optimizer, and extends it with a parallelism extraction module that modifies the original query optimizer to generate query plans for a parallel OODBMS architecture. The ideas are applied in the design of a parallel optimizer for OQL. The implementation approach shows that the parallelism extraction technique combined with a rule-based optimizer generator tool can help to produce a parallel optimizer from a non-parallel one with minimum effort.

1 Introduction

In the field of parallel DBMS, the bulk of the research is mainly targeted to the relational data model and primarily concerned with providing parallel algorithms for algebraic operators, partitioning techniques, hardware architectures, and resource allocation strategies [2], while issues such as query processing for OODBMS and parallel query optimization still remain largely open.

Some of the challenges involved in engineering a query optimizer for a parallel OODBMS relate to providing answers to the following issues— which extensions to conventional (non-parallel) query processing steps are needed; which features of the parallel architecture can be exploited during optimization; which implementation approaches can be employed in the design of the parallel query optimizer; and how to build a parallel optimizer that can be easily adapted to work with different parallel architectures, providing support for fast prototyping and experimentation.

This work tries to explore this gap in the context of parallel query optimization for OQL [1]. Our approach departs from a conventional (non-parallel) rule-based query optimizer and employs a parallelism extraction technique to unveil opportunities for independent parallelism and pipelined parallelism that can be exploited during query optimization for parallel OODBMS.

The parallelism extraction technique is based on analyzing the physical algebra generated by a non-parallel optimizer. The aim of the analysis is to factorize the implementation code of the non-parallel algebraic operators in terms of more primitive operations defined by the algebraic execution engine of a target parallel DBMS physical algebra. More primitive operations increase the chance of exploiting pipelined execution and the number of possible alternative parallel schedules that can be explored by the scheduler.

After the analysis and factoring of the non-parallel physical operators, the non-parallel optimizer is extended to incorporate rules that reflect the additions necessary to generate query plans for the parallel algebra. The query plans represent the input to the scheduler, that is responsible for allocating resources for parallel execution.

The approach, thus, combines a software reengineering phase of analysing a sequential physical algebra, and an engineering phase that takes the original logical query processing tree as input and generates parallel execution plans for the new physical algebra (parallel algebra).

The ideas described in this paper are employed in the implementation of the front-end of a parallel optimizer by extending a non-parallel main memory OQL implementation [3] using the optimizer generator tool OPTGEN [4]. Our approach shows that the parallelism extraction technique combined with a rule-based optimizer generator can help to produce a parallel query optimizer from a non-parallel one with minimum effort.

2 Related Work

This work is influenced by the ideas presented by Hasan in [7] regarding optimization of SQL queries for parallel machines, and the work of Hong on two-phase paral-

lel query optimization [9]. We follow the same approach adopted by Hasan, which divides parallel query optimization into two phases comprised of join ordering and query rewriting (JOQR) and parallelization. Given this division, we address the issue of implementing parallelism extraction in the parallelization sub-phase, for the OQL language.

Figure 1 shows the sub-phase of parallelism extraction in the context of parallel query processing according to [8].

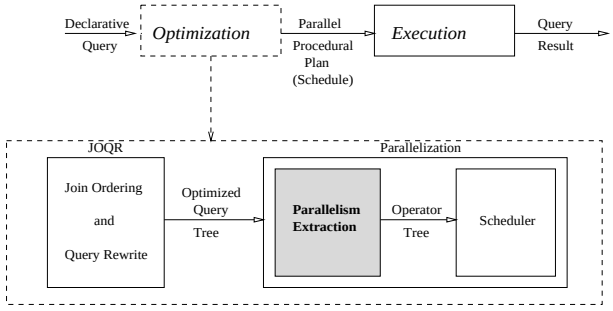

Figure 1. Parallel Query Optimization

Our approach is similar to Hong's in the sense that we depart from a conventional query optimizer, but we also combine Hasan's proposal of supporting atomic operators in the parallel physical algebra and annotations in the query trees regarding parallelism opportunities, obtained through the process of parallelism extraction.

Our main contributions rely on the fact that, differently from Hasan, we have shown how to perform the parallelism extraction process to divide the non-parallel physical algebra in atomic units using an existing optimizer; we have implemented the changes necessary to generate plans for the new physical operators using a rule-based optimizer generator tool, and we have shown how to annotate the query execution plans in the context of the optimizer. Finally, we are original in the sense that we elect the OQL language as the domain of our investigation.

3 Incorporating Parallelism Into Query Optimization

A conventional (non-parallel) OO query optimizer generates a query execution plan (QEP) indicating the order of execution of operators and the algorithms chosen for computing each operation [10]. Such a plan can be abstracted as an annotated query tree, where each node represents an operator that is annotated with the method for computing the operation, and each edge represents the data flow between the operators. Figure 2 shows a possible QEP generated by an OODBMS optimizer for the following OQL query:

```
select struct(E: e.name, D: e.dept.name)
from e in Employees;
```

This OQL query retrieves the name of all the employee objects from the extension of the class *Employees* and their corresponding department's name. Its result is a bag containing tuples whose components are an employee's name and its corresponding department name.

As shown in figure 2 the physical algebra algorithms applied for computing the query are: Reduce, which is a generalization of the relational operator Project. It evaluates an expression like <*E=e.name,D=e.dept.name*> for every input element and converts the output to a bag, set or list, depending on the input parameters and the expected results; Nested_loop, used for computing a Join operation; and Table_scan, which scans the extension of the class *Employees* and delivers the objects to the operator Nested_loop.

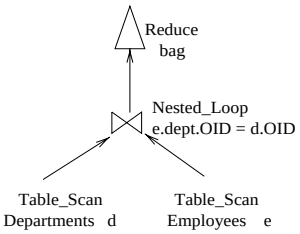

Figure 2. A QEP representation

The primary goal of applying parallelism extraction is to expose the opportunities for transforming QEPs in a way that takes advantage of an underlying parallel architecture. The ultimate result is a QEP that preserves the original semantics of the query and also incorporates information regarding the available parallelism through annotations in the query trees.

The information regarding parallelism opportunities can be obtained by performing the following steps [7]:

1. identifying the atomic units of execution
2. identifying the timing constraints between the units

Atomic units of execution for the target parallel architecture are pieces of code resulting from an appropriate factoring of the code that implements the operations specified in the non-parallel QEPs. The atomic units represent the interface between the scheduler and the optimizer in the parallel query processor and define the set of new operators that will be used in the construction of parallel QEPs.

In other words, this set of atomic operators will compose a new physical algebra for the parallel architecture, created by the factoring of the code associated with the non-parallel physical algebra that implements the logical algebra operators.

It is important to note that the logical algebra (the interface between the query language and the query compiler) remains the same, and the changes occur at the level of the physical algebra, by replacing the original non-parallel physical algebra with the new physical algebra obtained through the parallelism extraction process. In our work, the logical algebra that represents the interface between the OQL and the query compiler is the nested relational algebra described in [5].

As a first example of how the factoring of code can be executed, consider the operations specified in the QEP tree represented in figure 2. The tree denotes a query plan defined using the non-parallel algebraic operators (physical algebra) defined in [3].

This example shows a brief description of the code that implements the physical operator Reduce, referred in the plan of figure 2. This operator combines the functionality of a Project and a Select in the relational algebra. The operation parameters are: *monoid*: the collection type that will be used to structure the output of an operation; *plan*: the input plan of an operation; *variable*: the variable that gets bound to objects or collections after the execution of an operation; *head*: the projection expression to be evaluated for each input of the operation; *predicate*: a filter condition that selects the objects that will have the projection expression evaluated and stored in the output monoid.

The operator Reduce is divided in terms of two atomic operations of the parallel algebra. The splitting is based on analyzing the implementation of the original operators and identifying timing constraints between subparts of the implementation, reducing the inter-operator timing constraints to simple forms (precedence and parallel constraints), where the latter captures pipelined execution. These subparts will form new operators of the parallel physical algebra:

Operator: Reduce(monoid,plan,variable,head,predicate)

Code Description:

1. Evaluation of the expression *head* for each tuple of the input stream of *plan* that satisfies *predicate.*
2. The operator reduces of the entire stream to a value (a stream of one value bound to variable) or a collection (a stream of tuples, where each tuple binds *variable* to the value of *head*), depending on the output monoid.

Factoring Result:

1. Evaluate_head(plan,head,predicate) corresponding to step1 in the code description.
2. Reducing(monoid,plan,variable,head) corresponding to step2 in the code description.

Another example involves the splitting of a non-parallel Nested_loop, where the step of reading a plan into memory is separated from the matching of tuples (given that a naive approach [6] was used in the sequential implementation of the Nested_loop).

Operator: Nested_loop(left_plan,right_plan,predicate,keep)

Code Description:

1. Reading of the left plan (plan of smallest cardinality) into memory.
2. Concatenation of the tuples in the input streams of the left and right plans if they satisfy the *predicate*[1]. If *keep=left*, then it behaves like an left-outer join (it concatenates the left tuple with null values), if *keep=right* it behaves like a right-outer join, while if *keep=none*, it is a regular join.

Factoring Result:

1. Read(left_plan) corresponding to step1 in the code description.
2. Match(left_plan,right_plan,predicate,keep) corresponding to step2 in the code description.

In the next example, the Table_scan operator of the non-parallel algebra is analyzed to identify the parallelism opportunities. It is not further divided in terms of more primitive operators due to the atomic nature of scan operations when considering inter-operator parallelism:

Operator: Table_scan(extent_name,range_variable,predicate)

Code Description:

1. Creation of a stream of tuples, where each tuple has only one component, *range_variable*, whose value is an object from the extent.

Factoring Result:

1. Table_scan(extent_name,range_variable,predicate) corresponding to step1 in the code description.

After splitting the operators, the next stage of parallelism extraction deals with the timing constraints between the components of the new physical algebra. Timing constraints between operators also limit the inter-operator parallelism and therefore affect the parallel optimization step.

The following types of timing constraints can be identified between two units of execution:

- Parallel Constraints
- Precedence Constraints

A *parallel constraint* between two physical operators requires both operators to start at the same time and terminate at the same time. For example, consider the physical operators described above, Table_scan and Read. The Table_scan operator may pipe its output (a stream of tuples) to the Read operator, which is going to read the data into memory. Such concurrency eliminates the need to buffer intermediate results, although it places a parallel constraint on the computation.

A *precedence constraint* requires one operator to start after the other one terminates. For example, consider the physical operators (described above) Read and Match. The Read operator works by reading all its input into memory and the Match operator uses the data in memory to start the matching process with its other input. Since the Read operator's input must be fully read into memory before the

[1]For each read tuple of the right plan, all tuples in the left plan are read

Match operator starts its work, there is a precedence constraint in the computation.

The timing constraints between two units of execution can be identified by the analysis of their behavior (how they work) and relationship (if one of them passes data directly to the other) inside a QEP. Figure 3 illustrates the operator tree resulting from the application of the parallelism extraction technique on the non-parallel QEP shown in figure 2. Different types of parallelism may be applied given the information provided in the tree: *partitioned* parallelism may be used for any (physical) operator of the tree, as decided by the scheduler. *Pipelined* parallelism may be used for operators connected by pipelining edges. Two subtrees with no transitive precedence constraints across them may run independently, characterizing the *independent* parallelism.

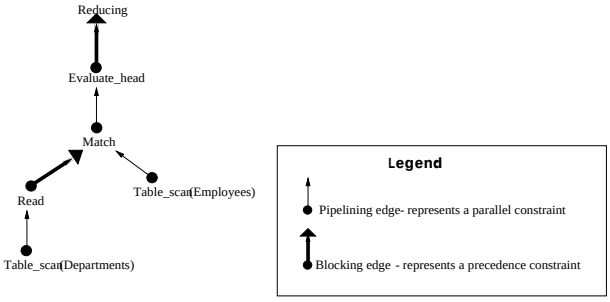

Figure 3. An operator tree exposing parallelism opportunities

As an example, the subtrees rooted at Read and Table_scan may run independently; operators Table_scan and Read may use pipelined parallelism; any operator may use partitioned parallelism. The blocking edges indicate that operators must execute sequentially. Evaluate_head and Reducing are included in this case.

4 Applying Parallelism Extraction

In this section we provide examples of the application of the parallelism extraction technique using OPTGEN. We start by analyzing the non-parallel optimizer implementation for OQL and we identify the parallelism opportunities. In order to obtain the units of execution for the parallel architecture, the sequential physical algebra code that implements the logical algebra operators are split into lower level atomic pieces that are relevant to the target architecture. For brevity, we are not going to present the factoring of all operations of the sequential physical algebra. Here follows a description of the factoring of the selected ones:

Operator: Reduce(monoid,plan,variable,head,predicate)
described in section 3

Operator: Nest(monoid,plan,variable,head,groupby, nestvars,predicate)

Code Description:

1. Grouping of the input stream of *plan* by the variables in *groupby*.
2. Creation of an extended tuple for each different binding of the *groupby* variables with the values of the *nestvars* variables.
3. Application of the operator *Reduce(monoid,plan, variable, head,predicate)* for each different binding of the *groupby* variables.

Factoring Result:

1. Group(plan,groupby), corresponding to step 1 in the code description
2. Nesting(plan,nestvars), corresponding to step 2 in the code description
3. Evaluate_head(plan,head,predicate), corresponding to step 3 in the code description
4. Reducing(monoid, plan,variable,head), corresponding to step 3 in the code description

Operator: Sort(plan,sort_order)

Code Description:

1. Formation of runs (the input data are written into initial sorted runs).
2. Merging of the runs (the runs are merged into larger and larger runs until only one run is left, the sorted output).

Factoring Result:

1. Form_runs(plan,sort_order), corresponding to step 1 in the code description.
2. Merge_runs(plan,sort_order), corresponding to step 2 in the code description.

Operator: Merge_join(left_plan,right_plan,predicate,keep, left_sort_order,right_sort_order)

Code Description:

1. Concatenation of the tuples in the input streams of the left and right plans if they satisfy the *predicate*. If *keep=left*, then it behaves like an left-outer join (it concatenates the left tuple with null values), if *keep=right* it behaves like a right-outer join, while if *keep=none*, it is a regular join. It requires the left plan be sorted by *left_sort_order* and the right plan by *right_sort_order*.

Factoring Result:

1. Merge_join(left_plan,right_plan,predicate,keep, left_sort_order,right_sort_order), corresponding to step 1 in the code description.

Operator: Table_scan(extent_name,range_variable,predicate)
described in section 3

After identifying the atomic units of execution (parallel physical algebra operators) by the factoring of the code that implements the logical algebra used in the non-parallel optimizer, the logical algebra must be mapped to the new physical operators to allow the construction of operator trees. This mapping is implemented using the OPTGEN rule language (OPTL), through the writing of physical rules. Some of the physical rules written for this purpose are as follows:

```
{ order=order(); }
reduce(‘M,‘e,‘v,‘head,‘pred)
 = REDUCING(‘M,EVALUATE_HEAD(‘e,‘head,‘pred),‘v,‘head)
  : { size = ˆe.size;
      cost = ˆe.cost+ˆe.size;
      order = ˜e.order; };
```

This rule translates a Reduce operation of the logical algebra into a Reducing (‘M,Evaluate_head (‘e,‘head,‘pred),‘v,‘head) evaluation plan. The identifier *order* is an inherited attribute that represents the required order of the stream of tuples which is input into this evaluation plan. Such attributes provide context during term-rewriting. In the present rule, the required order is empty. *size*, *cost* and *order* are synthesized attributes, which form the physical properties of the produced plan. The values for such attributes are computed when a rewrite is complete. The symbol ‘ is a escape operator, whose operand is expected to be an expression that provides the value to substitute the operand identifier.

```
{ order=order(); }
nest( ‘M, ‘e, ‘v, ‘hd, ‘gv, vars(...r), ‘pred )
  = let list<Expr>* other_nestvars = r->append(all_nesting_vars(e))
     in REDUCING(‘M,EVALUATE_HEAD(NESTING(GROUP(‘e,‘gv),
                vars(...other_nestvars)),‘hd,‘pred),‘v,‘hd)
  : { size = ^e.size;
     cost = ^e.cost+(^e.size*^e.size);
     order = #<order()>; };
```

This rule translates a Nest of the logical algebra into a Reducing (‘M, Evaluate_head (Nesting (Group(‘e,‘gv), vars(...other_nestvars)),‘hd, ‘pred),‘v,‘hd) evaluation plan. The expressions *vars(...)* and *vars(...other_nestvars)* represent lists of variables. *let* binds the variables denoted by *vars(...other_nestvars)* to values of the same type as the one defined for them.

```
{ order=order(‘o,...os); }
(‘x<-{ order=order(); })
  : is˙plan(x) && !subsumes(#<order(‘o,...os)>,^x.order)
  = MERGE_RUNS(FORM_RUNS(‘x,order(‘o,...os)),order(‘o,...os))
  : { size = ^x.size;
     cost = ^x.cost+(^x.size*log_cost(^x.size));
     order = #<order(‘o,...os)>; };
```

This rule indicates that if the expected order of a plan, which is represented by the inherited attribute *order*, is not empty, then any plan *x* should be sorted over the expected order. The physical plan Merge_runs (Form_runs (‘x,order(‘o,...os)),order(‘o,...os)) does this work. The expression *(‘x<-{ order=order(); })* means that the initial order for *x* is empty. The C++ code between the first *:* and = is the guard of this rule. It means that the rule is executed only if the guard evaluates to true. The rule is executed if *x* is a plan and the order of *x* does not subsume the expected order.

The operators Table_scan and Merge_join were not broken into smaller units during the factoring process because they were already considered atomic. For this reason, there was no need for writing physical rules to map logical operators into these operators, since they already existed in the non-parallel optimizer.

op-op	*table scan*	*merge join*	*form runs*	*merge runs*	*eval. head*	*red.*	*group*	*nest.*
table scan		par.c.	par.c.		par.c.		par.c.	
merge join		par.c.	par.c.		par.c.		par.c.	
form runs				pre.c.				
merge runs		par.c.	par.c.		par.c.		par.c.	
eval. head						pre.c.		
red.		par.c.	par.c.		par.c.		par.c.	
group								pre.c.
nest.					par.c.			

Table 1. Timing constraints

The timing constraints between the physical operators, identified by the analysis of the operators’ behavior and relationships in a procedural plan, are presented in table 1.

In table 1 the operator in each row represents the one from which data are output. The operator in each column represents the one whose input data came from a row operator. *par.c.* and *pre.c.* indicate that there is a parallel constraint and a precedence constraint between the corresponding row and column operators, respectively; *blank* indicates that the corresponding row operator never outputs data directly to the respective column operator in an operator tree.

In order to include the information regarding timing constraints between physical operators in the optimizer, a module composed of C++ classes and functions was added to the C++ program that composed the optimizer. This module annotates the operator trees with parallelism information in operator trees. This information is obtained from a table structure also made available in the module, similar to the one described in table 1.

The result of the application of the parallelism extraction technique in the query optimizer using OPTGEN is the production of QEPs that unveil information regarding the available parallelism, which was not represented in the sequential QEPs (before the application of the technique).

The QEPs resulting from parallelism extraction are ready to go through the scheduling step, in which resources are allocated to each operator of a QEP, taking into account the order and the timing constraints between the operators.

The following OQL query samples were tested in the parallel OQL optimizer implemented using OPTGEN after the application of the parallelism extraction technique. The corresponding physical plans are also shown:

Query 1: `sum(select e.age from e in Employees);`

```
REDUCING(sum,
  [PrC] EVALUATE_HEAD( [PaC] TABLE_SCAN(Employees,e,and()),
                             project(e,age),and()),
  X_582,project(e,age))
```

The physical plan generated from the OQL query 1 shows the physical operators used in the evaluation of the query and the timing constraints between them. For example, between the operators Table_scan and Evaluate_head there is a parallel constraint, indicated by *[PaC]*; between the operators Evaluate_head and Reducing there is a precedence constraint, indicated by *[PrC]*.

Query 2:
```
select struct(E:e.name, D:e.dept.name)
       from e in Employees
```

REDUCING(bag,
[PrC] EVALUATE_HEAD(
[PaC] MERGE_JOIN([PaC] TABLE_SCAN(Departments,X_524,and()),
[PaC] MERGE_RUNS([PrC] FORM_RUNS(
[PaC] TABLE_SCAN(Employees,e,and()),
order(OID(project(e,dept)))),
order(OID(project(e,dept)))),
and(eq(OID(project(e,dept)),OID(X_524))),right,
order(OID(X_524)),order(OID(project(e,dept)))),
struct(bind(E,project(e,name)),bind(D,project(X_524,name))),and()),
X_482,struct(bind(E,project(e,name)),bind(D,project(X_524,name))))

As in the physical plan generated from query 1, the physical plan generated from query 2 shows the physical operators used in the evaluation of the query and the timing constraints between them. Both plans represent the potentially best plans for the evaluation of their corresponding queries, considering the cost model used by the optimizer.

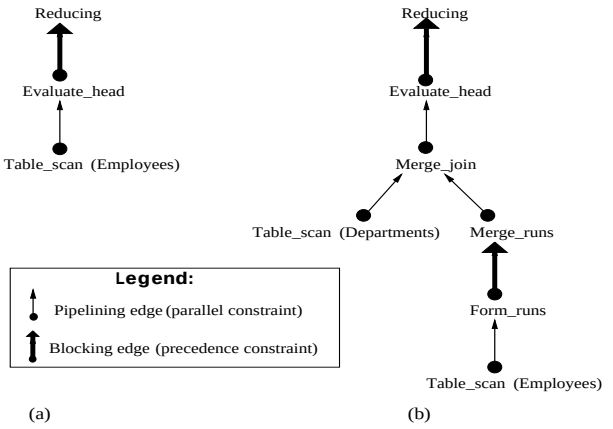

Figure 4. (a) Query 1. (b) Query 2

Figure 4 shows the plans generated for OQL queries 1 and 2 in the form of graphs. Figure 4(b) is the query plan generated for the same query that appears in section 3. The query plan generated by the optimizer shown in figure 4(b) has a different choice of physical operators than the example plan described in section 3 (illustrated in figure 2). This reflects the work of the optimizer in choosing more efficient physical operators in the query execution plan generated for the same query. The plans also reflect the annotations indicating parallel [PaC] and precedence constraints [PrC] that unveil to the scheduler the parallelism opportunities.

5 Summary

This work addresses the issues related to incorporating parallelism extraction techniques into the optimization step of a non-parallel query optimizer to implement modules of a parallel optimizer for OQL. We have adopted a solution based on the use of a rule-based optimizer generator tool and its associated rule language OPTL.

Although we provided a solution based on the use of OPTL as the query rewriting rule language, the ideas behind parallelism extraction can also be applied to build parallel query optimizers, regardless of the data model or query language supported. The use of a rule based optimizer and the analysis of an existing implementation for OQL using the parallelism extraction technique provides a simple and fast way to prototype a parallel optimizer for OQL.

Using the parallelism extraction technique, we have obtained a target parallel physical algebra that abstracts the atomic operations (pieces of code) supported by a scheduler for a parallel OODBMS that can be easily realized using current parallel architectures. The resulting plans, besides unveiling parallelism opportunities, are easily adaptable within the context of different parallel architectures.

References

[1] R. Cattel and D. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufman, 1997.

[2] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[3] L. Fegaras. An experimental optimizer for oql. Technical Report TR-CSE-97-007, CSE, University of Texas at Arlington, 1997.

[4] L. Fegaras. *The OPTGEN Optimizer Generator*. Department of Computer Science and Engineering, The University of Texas at Arlington, http://ranger.uta.edu/∼fegaras/optimizer, 1997.

[5] L. Fegaras. Optimizing large oodb queries. In *Proc. of the 5th Intl. Conference on Deductive and Object-Oriented Databases (DOOD' 97)*, 1997.

[6] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[7] W. Hasan. *Optimization of SQL Queries for Parallel Machines*. PhD thesis, Stanford University, Department of Computer Science, 1996.

[8] W. Hasan, D. Florescu, and P. Valduriez. Open issues in parallel query optimization. *SIGMOD Record*, 25(3), September 1996.

[9] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *Proc. of the 1st Intl. Conference on Parallel and Distributed Information Systems*, Florida, USA, 1991.

[10] M. T. Ozsu and J. A. Blakeley. *Modern Database Systems*, chapter Query Processing in Object-Oriented Database Systems, pages 146–174. Addison-Wesley, 1995.