A Parallel Algebra for Object Databases

Sandra de F. Mendes Sampaio † Norman W. Paton † Paul Watson ‡
Jim Smith ‡

†*Computer Science Dept., Manchester University of Manchester, M13 9PL, UK*
E-mail: {sampaios, norm}@cs.man.ac.uk

‡*Computing Science Dept., Newcastle University of Newcastle upon Tyne, NE1 7RU, UK*
E-mail: {Paul.Watson, Jim.Smith}@newcastle.ac.uk

Abstract

This paper describes an algebra for use with parallel object databases, and in particular ODMG compliant databases with OQL. Although there have been many proposals for parallel relational database systems, there has been much less work on parallel object databases, and on parallel query processing for object databases. The parallel algebra presented in this paper is an extension of an existing physical algebra for OQL, and has an important role during query optimization, and for describing execution plans. The paper presents not only the algebra, but also its role in the architecture of a parallel database.

1. Introduction

Object databases are becoming fairly well established for use in a range of applications [5]. Furthermore, the ODMG standard [3] is providing a greater level of consistency across products than was evident in the early days of object databases. However, it is still the case that object databases are normally associated with advanced applications with stringent performance requirements, and that performance issues are likely to slow the uptake of object databases in certain domains.

One way of achieving higher performance in databases is by exploiting parallel execution of queries. Most work on parallelism in databases is concentrated on relational database systems, e.g. on parallel database system development, data partitioning techniques, and parallel join algorithms. Little effort has been focused on object databases, and the work done so far is limited. For example, the Monet database system [2] does not exploit inter-operator pipeline parallelism.

One of the aims of the *Polar Project* [13] is to build a parallel optimizer and query evaluator for an object-oriented database system, to be run over an ODMG [3] database server on a low-cost parallel platform based on PCs connected through an ATM network, as well as on a dedicated parallel database machine [14]. In this paper, we describe the parallel algebra used in the *Polar* parallel optimizer. There are not many parallel algebras in the literature, and most of them are fairly complex [4], as they introduce special constructs and distinguished types of operators with different functionalities, for expressing parallelism inside queries. Our parallel algebra is an extension of the physical algebra proposed by Fegaras in [6] to include a parallelism related operator and a pointer-based join operator. We chose to extend this algebra because it is based on the concept of *Monoids* [11], which allow uniform treatment for collections and scalars, and also because the use of this algebra allows us to build the parallel optimizer over the conventional OQL [3] optimizer implemented by Fegaras [6, 7]. In other words, we are using Fegaras' work on non-parallel ODMG query optimization as a starting point, and extending it with parallel optimization and evaluation capabilities.

The paper is organised as follows. Section 2 describes the query processing architecture of the Polar parallel optimizer. The parallel algebra is described in Section 3, and Section 4 concludes.

2. Architecture

Our design approach for building the *Polar* parallel optimizer is to exploit *modularity*. The parallel optimizer is an extension of a non-parallel OQL optimizer, which generates a number of execution plans for a query and selects the least costly one for execution. We use the non-parallel optimizer implemented by Fegaras [6] as a starting point, making a few modifications: the addition of a pointer-based join that takes advantage of the explicit relationships between classes; and the keeping of several of the least costly plans generated during conventional optimization, so that a range of plans can be considered for parallel optimization.

Two main activities are involved in parallel optimization: *partitioning* the plan into subqueries for parallel execution; and *assigning* the previously identified subqueries

to specific processors. The current version of the parallel optimizer partitions the plan where movement of data must occur. For example, consider a *join* operation applied over `Employees` and `Departments`, and suppose that the join predicate is `employee.city = department.city`. When the join operation is parallelized, the input data to the join operator has to be partitioned over the joining attribute, to ensure that the tuples that contain objects with the same value for the joining attribute are allocated to the same node of the parallel machine. In the example, the input tuples containing the objects of the `Employees` and `Departments` extents may be "hashed" on the attribute `city`. In cases when the input data is already partitioned on the required attributes, no movement of data is necessary.

Figure 1 illustrates the components of the system and the main aspects involved in each stage of the optimization.

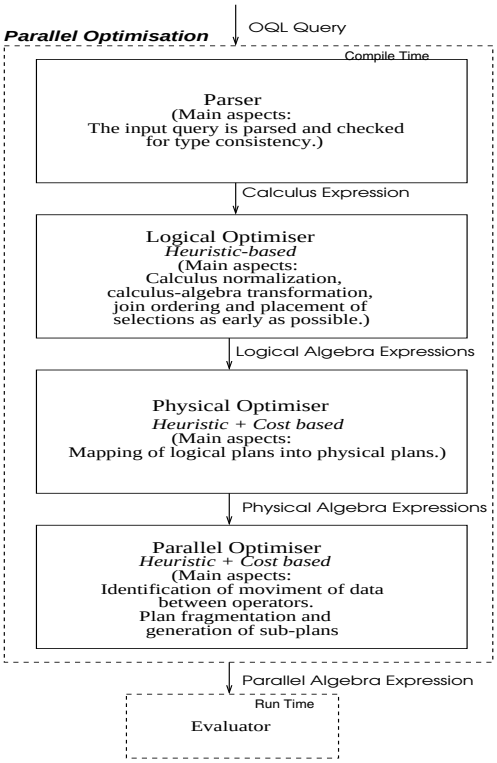


Figure 1. Parallel optimizer components.

As shown in figure 1, the internal representations of an OQL query are mainly algebraic. The calculus expression, derived from the input OQL query by the *Parser*, is translated into a logical algebra expression. At this stage in optimization, the information available to the optimizer relates to the logical operators that compose the query, and a range of orders in which the operators could be executed are identified. During physical optimization, each logical operator is mapped into one or more physical algebra operators. The physical algebra operators are specific algorithms for evaluating the query, and therefore have cost functions associated with them. Such an algebra is system-specific, meaning that different systems may implement the same data model and the same logical algebra, but may use different physical algebras [9]. Table 1 shows the logical and the physical operators used by the conventional optimizer. We describe the physical algebra operators in more detail in section 3. Each logical operator can be implemented by at least one physical operator, and some of the logical operators have more than one possible mapping into physical operators. The physical optimizer may perform one or more mappings for each logical operator, generating a number of physical plans.

Table 1. Logical and physical algebras.

Logical operators	*Physical operators*
Get	Table_scan
	Index_scan
Join	Nested_loop
	Merge_join
Materialize	Materialize
Nest	Nest
	Groupby
Unnest	Unnest
	Sort
Reduce	Reduce
Union	Union
Map	Map

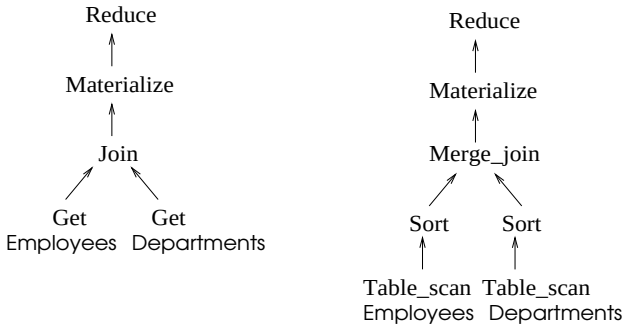
The physical algebra outlined in table 1 is similar to the one described in [6]. However, the Materialize operator [1] has been added to this algebra, providing the execution engine with an operator capable of "bringing into scope" objects from another extent and performing a join between those objects and the input objects, by following the relationships between the input objects and the objects of the "hidden" extent. Note that the Sort operator has no correspondent in the logical algebra. Thus, it is purely a physical operator.

An algebraic query plan is represented as a tree, in which the internal nodes represent operators, and the leaf nodes represent database extents. Figure 2 illustrates logical and physical plans derived from the following OQL query expression.

```
select distinct struct(E: e.name,
                       M: e.manager.name)
from e in Employees, d in Departments
where e.city = d.city;
```

The query retrieves the names of all employees and their managers, for those employees who have the same city as

a department. As illustrated in figure 2, the logical operators Reduce, Materialize, Join and Get can be mapped into the physical operators Reduce, Materialize, Merge join with two Sort operators, and Table scan. Considering the example, the Table scan operators retrieve the objects from the `Employees` and `Departments` extents, and output the resulting tuples (tuples carry the information between operations in the algebra) to the Sort operators. The Sort operators sort the input tuples by their `name` attributes and output their result to the Merge join operator. The Merge join operator concatenates the tuples from its two input streams if they have equal values for their `city` attribute. Therefore, the Merge join operator builds a new tuple type, which results from the combination of the `Employee` tuples and the `Department` tuples. Its resulting tuples are then sent to the Materialize operator. The Materialize operator receives its input tuples, follows the `manager` relationship to retrieve the `Employee` instance who is the manager, and concatenates this `Employee` object with the input tuple. The resulting tuples are output to the Reduce operator, which structures the query result by building a structure composed of a set of names of employees and the names of their managers.



(a) Logical plan (b) Physical plan

Figure 2. Logical and physical query plans.

3. Parallel algebra

There are two general approaches for parallelizing a database query execution engine, which are called in [9] the *bracket model* and the *operator model*. In the bracket model, a generic process template is used by the physical operators for receiving and sending data. In other words, network I/O is implemented as procedures to be called by the operators. In the operator model, parallelism related operators are inserted into a sequential plan, transforming it into a parallel plan. The parallelism related operators have similar interfaces to the other physical operators, but are different in functionality, as they provide mechanisms for parallel query processing. The operator model provides a simple way of parallelizing an existing sequential operator plan by inserting parallelism related operators in the plan. Moreover, as mechanisms for parallelism are encapsulated in the parallelism related operators, the development and maintenance of non parallelism related operators is simplified.

Since we chose to use the operator model approach, we have inserted a "parallelism" operator into the physical (sequential) algebra described in section 2. The parallelizing operator is the Exchange operator.

The Exchange operator, used in the Volcano system [8], provides control functionalities to the execution engine, such as data redistribution and flow control, which are not provided by the physical operators described previously. It is not, however, a data manipulation operator, being responsible solely for the mechanics of parallel execution.

The parallel algebra is, therefore, composed of the data manipulation operators of the sequential physical algebra, plus the parallelism related operator. The algebra is described below:

1. Table_scan(extent, range_variable, predicate)

   Creates a stream of tuples from the given *extent* that satisfy the *predicate*, in which the individual tuples are referred to by the *range_variable*.

2. Index_scan(extent, range_variable, predicate, index, sort_order)

   As with *Table_scan*, this creates a stream of tuples from the given *extent*, but using the given *index* to deliver the tuples ordered by *sort_order*.

3. Nested_loop(left_plan, right_plan, predicate, keep)

   Creates a stream of tuples from the join of the tuples from the right and the left input plans *left_plan* and *right_plan* that satisfy *predicate*. The last parameter, *keep*, specifies how the join operation should behave: if *keep* = *left* the join operation behaves as a left_outer_join operation, otherwise, if *keep* = *right* it behaves as a right_outer_join, and if *keep* = *none* it behaves as a regular join.

4. Merge_join(left_plan, right_plan, predicate, keep, left_sort_order, right_sort_order)

   As with *Nested_loop*, but with the requirement that the left and right plans are ordered by *left_sort_order* and *right_sort_order*, respectively.

5. Materialize(plan, path, predicate1, predicate2)

   Concatenates each tuple of the input *plan* to a tuple containing an object from another extent to which objects in the input are related by the given *path*. *predicate1* filters the tuples from the input *plan*, and *predicate2* filters the tuples generated from the concatenation process.

6. Reduce(monoid, plan, variable, head, predicate)

   Structures the input *plan* according to the structure specified by the *monoid*. Returns the data that results from applying the expression *head* to every tuple in *plan* that satisfies the *predicate*.

7. Nest(monoid, plan, variable, head, groupby, nestvars, predicate)

   Groups the tuples of the input *plan* by a set of attributes *groupby*, applying *head* and nesting the attributes *nestvars* for each resulting group, and keeping only the groups that satisfy the *predicate.*

8. Groupby(monoid, plan, range_variable, head, groupby, nestvars, predicate)
   Similar to *Nest*, but requires that the input *plan* is ordered by the variables in *groupby*.
9. Unnest(plan, variable, path, predicate, keep)
   Concatenates each tuple of the input plan (outer collection, bound to *variable*) to all possible values of *path* (inner collection), keeping only tuples that satisfy *predicate*. If *keep = true*, where there are no values for *path* or no value satisfies the predicate, the tuple is padded with nulls.
10. Sort(plan, sort_order)
    Sorts the input *plan* by *sort_order*.
11. Union(monoid, left_plan, right_plan)
    Merges the (union compatible) streams of the input plans (*left_plan* and *right_plan*). The *monoid* parameter specifies how the result should be structured.
12. Map(plan, variable, function)
    Extends the input stream of *plan* with the binding of *variable* to the result of the application of *function* to the input tuples.
13. Exchange(plan, variable, destination)
    Receives the input tuples of *plan* from different nodes of a parallel machine (each tuple bound to *variable*), and computes the destination node of each tuple, using the *destination* parameter, which is a function applied over the partitioning attribute(s).

The presence of parallelizing operators in the execution engine of a database and the use of an underlying parallel architecture allows parallelism to be exploited in the execution of a query plan. To obtain a parallel query plan from a sequential query plan (the output of a conventional optimizer), exploiting intra-operator and inter-operator parallelism, it is necessary to partition the plan and assign a set of processors for each partition. Since the Exchange operator is associated with data communication among different nodes of a parallel machine, the insertion of this operator in a conventional query plan divides the plan into different sets of operators. Figure 3 illustrates two alternative ways of inserting the parallelism related operator in the plan shown in figure 2(b). There are many possible ways of partitioning a query plan and, therefore, many possible placements for the Exchange operators. [12] distinguishes *attribute sensitive* operators and *attribute insensitive* operators. An attribute sensitive operator is an operator *partitionable* only for partitionings that use a distinguished attribute. On the other hand, an attribute insensitive operator is partitionable for all partitionings (they may be partitioned by any attribute). An Exchange operator must be placed before an attribute sensitive operator if any of its child operators use different partitioning attributes, so that the data is partitioned by the required attributes. Grouping operators and valued-based join operators such as Nested loop are examples of attribute sensitive operators, as they require partitioning by the joining and grouping attributes, respectively. Operations such as Union and Unnest are attribute insensitive, as they usually don't require a distinguished partitioning. Table 2 classifies the parallel algebra operators as attribute sensitive or attribute insensitive operators.

Table 2. Operators classification.

	Attr. Sensitive	*Attr. Insensitive*
Table_scan		
Index_scan		
Reduce		√
Nested_loop	√	
Merge_join	√	
Materialize		√
Sort		√
Unnest		√
Nest	√	
Groupby	√	
Union		√
Map		√
Exchange		√

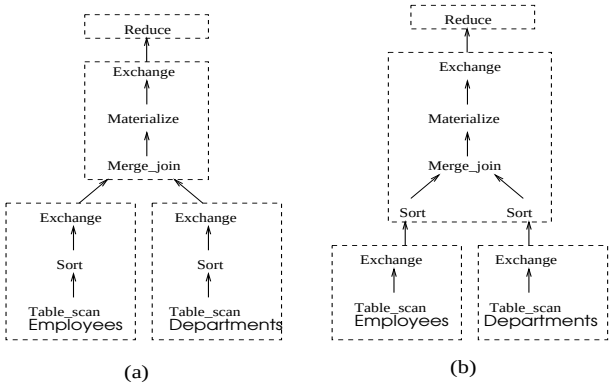


Figure 3. Parallel query plan.

The reason why Exchange operators are placed before the Sort operators (which are attribute insensitive) and not before the Merge join (attribute sensitive operator) in the plan in figure 3(b) is to avoid the possibility of *deadlock*, as the plan in figure 3(a) is not guaranteed to be deadlock-free in all situations (see [9] for an explanation). For cases in which a parallel sort operation (either a single sort operator executed in parallel, or more than one sort operator feeding the same consumer operator) is followed by an operator that depends on ordering (as the Merge join operator in figure 3) and that is executed in parallel, the partitioning of data step should be performed before the sorting of data to avoid any possibility of deadlock.

The operators Table scan and Index scan are usually parallelized based on the partitioning of the data being read. Thus, even if an attribute sensitive predicate is specified in any of these scan operations, no Exchange operator is required to do data repartitioning. The operator Reduce is considered as attribute insensitive, as its role is simply to structure the results of a query. However, an Exchange operator may be necessary to channel data from different

nodes to the node that hosts the Reduce operator, so that aggregations are performed and duplicates are removed.

Depending on the number of processors in a parallel machine, there may be many possible ways of assigning a set of processors to execute each partition of a plan. A compile-time parallel optimizer does not consider runtime information such as current load on processors when assigning processors to subqueries. Thus, such optimizers have to rely on other criteria to make their decisions. Some of the decisions a compile-time optimizer could make are: *(a)* Allocate scan operations based on data location and data distribution information, so that only the processors with relevant data are assigned to execute the scan operations. *(b)* Try to partition the other (non-scan) operators over the processors, in such a way that the processors receive approximately the same load. In this case, the load resulting from other running tasks is not taken into account. *(c)* One of the processors should be assigned for the execution of the Reduce operator, when performing aggregations. *(d)* Try to use the same set of processors when data repartitioning is required, increasing the chance of a number of tuples (output from the producer operators) not having to be moved across nodes (to the consumer operators) and, therefore, saving communication costs.

4. Conclusions

In this paper, we have described the architecture of the parallel optimizer of the *Polar project*, which extends an existing OQL query optimizer [6] with a parallel optimizer module. The parallelization process has two main aspects: deciding how to partition a query plan into a number of subqueries, and deciding a processor assignment for each partition of the plan. A parallel algebra has been provided whose operators are data manipulation operators (the sequential physical algebra) and a parallelism related operator that is responsible for providing mechanisms for parallel query processing.

The combination of data manipulation operators and the parallelism related operator provide the execution engine with the necessary operations to allow the parallel execution of queries. When transferring data between different nodes of the parallel machine, the Exchange operator is used, so that remote communication is performed. When movement of data is not required, the data manipulation operators execute independently on each node.

By distributing data over the nodes of the machine and by implementing the parallel operators in a data-flow execution mode, for example using the *Iterators model* [10], it is possible to exploit the partitioned (intra-operator parallelism) and the pipeline (inter-operator parallelism) parallelisms.

The current status of the Polar optimizer is that the features described in this paper have been implemented, although with straightforward algorithms for both query partitioning and processor allocation. The first prototype of Polar is expected to be completed in the summer of 1999.

Acknowledgement: This work is supported by the Engineering and Physical Sciences Research Council (EPSRC), whose support we are pleased to acknowledge. We are also grateful for the input of our industrial partners Phil Broughton and Nic Holt of ICL.

References

[1] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the Open OODB query optimizer. In *Proceedings of the ACM SIGMOD*, pages 287–296, Washington, DC, USA, 1993.

[2] P. Boncz, F. Kwakkel, and M. L. Kersten. High performance support for OO traversals in Monet. In *British National Conference on Databases*, pages 152–169, Edinburgh, UK, July 1996.

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

[4] C. Chachaty, P. Borla-Salamet, and M. Ward. A compositional approach for the design of a parallel query processing language. In D. Etiemble and J.-C. Syre, editors, *PARLE '92, Parallel Architectures and Languages Europe*, volume 605 of *LNCS*, pages 825–840. Springer-Verlag, Berlin, 1992.

[5] A. Chaudhri and M. Loomis. *Object Databases in Practice*. Prentice Hall, 1998.

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

[7] L. Fegaras. Query unnesting in object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, June 1998.

[8] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 102–111, 1990.

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

[10] G. Graefe. Iterators, schedulers, and distributed-memory parallelism. *Software-Practice and Experience*, 26(4):427–452, April 1996.

[11] T. Grust, J. Kröger, D. Gluche, A. Heuer, and M. H. Scholl. Query evaluation in CROQUE - calculus and algebra coincidence. In *15th British National Conference on Databases*, volume 1271 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 1997.

[12] W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *Proceedings of the 21th VLDB Conference*, pages 239–250, 1995.

[13] P. Watson. The design of an ODMG compatible parallel object database server. In *Vec Par'98*, Porto, Portugal, June 1998. Lecture Notes in Computer Science.

[14] P. Watson and G. W. Catlow. Architecture of the ICL Goldrush MegaServer. In *British National Conference on Databases*, volume 940 of *Lecture Notes in Computer Science*, pages 249–262, Manchester, 1995. Springer-Verlag.