Industrial-Strength Parallel Query Optimization: issues and lessons

Rosana S.G. Lanzelotte[2,1], Patrick Valduriez[1],
Mohamed Zaït[1], Mikal Ziane[3,1]

[1]Projet Rodin, INRIA, Rocquencourt, France
[2]Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Brazil
[3]Universite Paris 5, Paris, France
e-mail: Firstname.Lastname@inria.fr

Abstract

In the industrial context of the EDS project, we have designed and implemented a query optimizer which we have integrated within a parallel database system. The optimizer takes as input a query expressed in ESQL, an extension of SQL with objects and rules, and produces a minimum cost parallel execution plan. Our research agenda has focused on several difficult problems: support of ESQL's advanced features such as path expressions and recursion, modelling of parallel execution spaces and extensibility of the search strategy. In this paper, we give a retrospective on the optimizer project with emphasis on our design goals, research contributions and implementation decisions. We also describe the current optimizer prototype and report on experiments performed with a pilot application. Finally, we present the lessons learned.

1 Introduction

EDS is an ESPRIT project started in 1989 by Bull, ICL, Siemens, ECRC and INRIA with the major goal of producing a parallel database server [11] which exploits recent multiprocessor computer architectures in order to provide high-performance and high-availability database support at a much lower price than equivalent mainframe computers. The thrust of the project is that database management and parallel processing technologies are mature enough to be successfully combined and take a central position in mainstream commercial information systems of the 90s.

Parallel database system architectures range between two extremes, the *shared-memory* (SM) and the *distributed-memory* (DM, also called shared-nothing) architectures. In the SM approach,

any processor has access to any memory module or disk unit through a fast interconnect and inter-processor communication is done via a shared memory. Examples of SM parallel database systems include XPRS [28] and Volcano [7]. In the DM approach, each processor has exclusive access to its main memory and disk unit(s). Inter-processor communication is done via message-passing through an interconnect. Examples of DM parallel database systems include the Teradata's DBC and Tandem's NonStopSQL products as well as a number of prototypes such as Bubba [2] and GAMMA [3]. More experimental study is needed to decide which of these architectures is best for data management under various workloads [32]. Intuitively, SM has less extensibility and reliability because of the common memory. but for a small configuration (e.g., less than 20 processors), it can provide the highest performance because of better load balancing. However, DM can scale up to hundreds of processors. Thus, it appears that DM is the only choice for high-end systems, e.g., requiring thousands of TPS of the TPC-B benchmark. For smaller systems however, SM is an interesting alternative.

For these reasons, it was decided to build two prototypes of the EDS database server. The first one is DM and has been implemented on the EDS parallel computer (also developed in the EDS project by ICL and Siemens). The latter, called DBS3 (Database System on Shared Store) [1], has been implemented on an Encore Multimax SM multiprocessor. However, we insisted that most of the database software be the same for both prototypes to demonstrate portability as well as reduce development costs. In both prototypes, parallel data processing is obtained by uniformly supporting a distributed-memory execution model. In the case of DBS3, this is useful to logically partition the data in order to minimize common memory access conflicts and thus increase parallelism.

The EDS database server targets business data processing applications with mixed workloads of low-complexity (OLTP-like) transactions and more complex decision-support queries because they contribute most to throughput demands. The industrial context of the project naturally led to the choice of standard SQL for providing database capabilities. The rationale was to not attempt to develop and promote yet another database language. However, SQL itself is a moving target, recently extended in SQL2 and subject to further object-oriented extensions in future SQL3. Therefore, in order to develop advanced database technology within EDS, we decided early on in the project to specify our own extensions of SQL. The result was ESQL [6], an SQL2 upward-compatible language with deductive and object-oriented capabilities.

ESQL is intended for traditional data processing as well as more complex applications such as geographical information systems or knowledge-based systems. ESQL's salient features are a rich and extensible type system based on abstract data types, support for complex objects with object sharing by combining collection type constructors and object identity, support for multi-statement queries and a Datalog-like deductive capability provided as an extension of the SQL

view mechanism. Our own implementation of extensions of ESQL has put in place a design capable of supporting the actual standards as they emerge.

With ESQL, the programmer can write high-level queries that are more powerful and complex than relational queries. In particular, they may involve path expressions for navigating through objects and recursive predicates. Thus, to achieve the EDS database server's functionality and performance objectives, it is crucial to compile and optimize ESQL for efficient parallel execution. The ESQL compiler proceeds classically in three subsequent phases that progressively add lower levels of details regarding the execution environment: query analysis and rewriting, query optimization and parallel code generation. The same compiler works on both SM and DM prototypes (only the code generator is different). However, catalog information regarding the execution environment and the database is necessarily different and encapsulated within the optimizer. Throughout compilation, we use an intermediate language called Lera (Language for extended relational algebra) to support ESQL functionality.

The role of the optimizer is to determine for an ESQL query the parallel execution plan that minimizes an objective cost function. This execution plan is the "best" relatively to the set of candidate plans examined by the optimizer but not necessarily the optimal one (e.g., a plan which could be hand-coded). To perform its task, the optimizer must have full knowledge about the parallel execution environment (i.e., operator cost functions) and the database (i.e., data placement, access paths and statistics).

Compared to a relational query optimizer, the EDS optimizer is complicated by three problems. First, ESQL and Lera have higher expressive power than relational languages. Second, the distributed execution model offers a wide range of parallel execution strategies that are difficult to abstract. Third, the required ability of efficiently optimizing both repetitive (compiled) queries and ad-hoc queries (executed only once) implies that the optimization time and therefore the optimization strategy be itself controllable through some form of extensibility.

We have built the optimizer following a decade of experience with commercial optimizers. We describe an optimizer along the three following dimensions: a *search space* which defines in an abstract way the alternative execution plans; a *cost model* which predicts the cost of an execution plan; and a *search strategy* which is used to select the best execution plan in the search space. For each dimension, we have produced original solutions to address the new problems caused by the language, the parallel execution model and the need for extensibility.

In this paper, we report on our experience in building the EDS optimizer which has been successfully integrated with the rest of the ESQL compiler and we stress the industrial context of the EDS project. We follow a retrospective approach to explain how we discovered new problems,

found solutions and were able to implement and validate them in an industrial-strength prototype.

The paper is organized as follows. Section 2 gives the design goals that we set early on for the optimizer. Section 3 gives the optimizer genesis with the design of the optimizer architecture, search space, cost model and search strategies. Section 4 presents our experience with the optimizer prototype and its use in experiments involving a pilot application and benchmarks. Section 5 summarizes the results and the lessons learned.

2 Design Goals and Issues

This section presents the main objectives that influenced the optimizer design. First, we describe the optimization environment, which includes the input language and the parallel execution model. Their main features are outlined, as well as their impact on the optimizer components. As we aimed for an extensible optimizer, we considered some design choices, which are also discussed.

2.1 Query Language

ESQL, the input language for the EDS database server, includes standard SQL2 and integrates most features of deductive and object-oriented databases. The main original features of the language are to support the following capabilities:

- tuple and collection (e.g., set, list, bag, vector) constructors, with associated manipulation methods, provide the basis for building user-defined complex data types;
- nesting and unnesting;
- object identifiers, which provide for object sharing;
- user-defined and system-supplied methods for object manipulation, including path expressions;
- recursive views.

The object-oriented concepts are introduced as SQL extensions, similar to the abstract data type capability of POSTGRES [29]. Keeping the relational basis of the language favors the reuse of known query optimization techniques, mainly for dealing with joins, the most costly relational operator.

The object-oriented features of ESQL prompted us to study specific optimization techniques. Since complex objects are composed of sub-objects, the main optimization issue is to speed up the access to the sub-objects of an object, expressed through path expressions. The standard techniques involved looking for a proper access path (index), but they did not consider all the alternative access orders. This means that they could miss the optimal order, and thus we proposed a technique which allowed considering all these orders [21]. However, when it came to implementation, we adopted a simpler approach. We decided not to change the order in which the user writes path expressions, and we did not implement path indices. We did not deal either with the optimization of user-defined methods, which is still an open problem.

Regarding recursion, we adopted the typical rewriting approach, i.e., prior to cost-based optimization. Even after realizing that this simplified approach may lead to missing the best plan [19], we insisted in adopting it because it was hard to find meaningful counterexamples in our pilot applications. Besides, we felt there were too many (maybe too good) papers on this topic.

Besides coping with the novelties of the input language and the execution environments, we also aimed at guaranteeing good performance for both OLTP-like transactions and more complex decision-support queries, with different requirements (single vs multiple executions of the transaction). This implied paying attention to the trade-off between the optimization cost and the execution cost. A high optimization cost (in terms of CPU time and memory space) is worthwhile if there are multiple executions of a query. On the other hand, if the query is to be executed once, spending too much time on optimization may not be worth doing.

With all those challenges in mind, we decided to emphasize some of them. We decided to work on the impact of parallel execution environments on known relational query optimization techniques.

2.2 Parallel Execution

The optimizer produces an *execution plan* for the input query, to be run in a parallel execution system. Compared to sequential optimizers, a parallel optimizer has to deal with a larger search space, because the parallel execution model allows much freedom for scheduling execution plans. In this section, we investigate the aspects that make a parallel execution plan different from a sequential one, and discuss their impact on traditional query optimization techniques.

2.2.1 Scheduling Execution Plans

Execution plans are typically abstracted by means of operator trees, which define the order in which the operations are executed. Operator trees are enriched with *annotations*, which indicate additional execution aspects, such as the algorithm of each operation. An important execution aspect to be reflected by annotations is the fact that two subsequent operations can be executed in *pipeline*. In this case, the second operation can start before the first one is completed. In other words, the second operation starts *consuming* tuples as soon as the first one *produces* them. Pipelined executions do not require temporary relations to be materialized, i.e., a tree node corresponding to an operation executed in pipeline is not *stored*.

Pipeline and store annotations constrain the *scheduling* of execution plans. They split an operator tree into non-overlapping sub-trees, called *phases*. Pipelined operations are executed in the same phase, whereas a storing indication establishes the boundary between one phase and a subsequent phase. To be more precise, for bushy trees (See Section 3.2) additional information must be added to completely specify the splitting in execution phases. More sophisticated techniques can also be applied for scheduling operator trees [25] but we decided to consider only pipeline and store annotations.

Some operations and some algorithms require that one operand be stored. For example hash-join (See Figure 1) usually consists of two consecutive phases: build and probe. In the build phase, a hash table is constructed on the join attribute of the smallest relation. In the probe phase, the largest relation is sequentially scanned and the hash table is consulted for each of its tuples.

In the left part of Figure 1, the temporary relations *Temp1* must be completely produced and the hash table in *Build2* must be finished before *Probe2* can start consuming R_3. The same is true for *Temp2*, *Build3* and *Probe3*. Thus, this tree is executed in four consecutive phases: build R_1's hash table, then probe it with R_2 and build *Temp1*'s hash table, then probe it with R_3 and build *Temp2*'s hash table, then probe it with R_3 and produce the result.

In the right part of Figure 1, the pipeline annotations are indicated by arrows. This tree can be executed in two phases if enough memory is available to build the hash tables: build the tables for R1, R3 and R4, then execute *Probe1*, *Probe2* and *Probe3* in pipeline.

Hash-based join algorithms have been shown to be the most efficient for equi-joins [24] and are particularly interesting in a parallel environment because they naturally induce intra-operation parallelism which we present below.

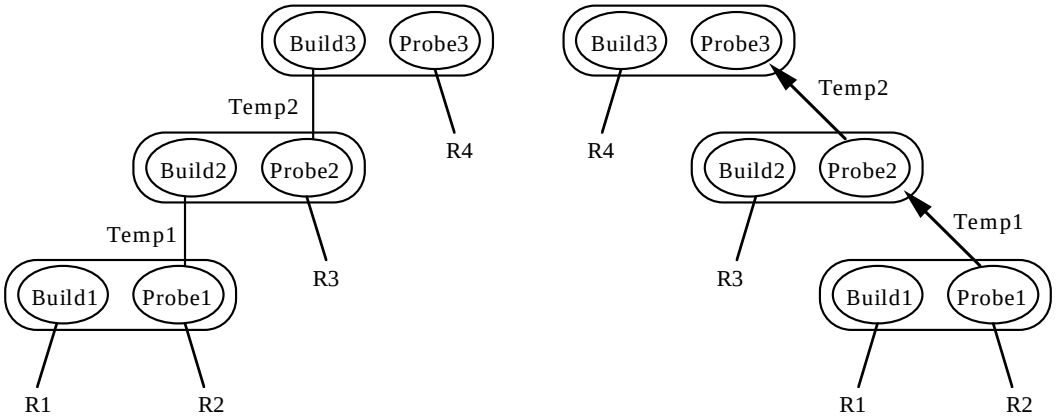

Figure 1: Two hash-join trees with a different scheduling.

2.2.2 Alternative Approaches to Parallelism

In a parallel environment, the execution of plans may involve different kinds of parallelism.

Inter-operation parallelism occurs when two or more operations are executed in parallel. It can be dataflow or independent. We call *dataflow* the form of parallelism induced by *pipelining*. *Independent* parallelism occurs when operations are executed at the same time or in arbitrary order. Independent parallelism is possible only when the operations do not involve the same data.

Intra-operation parallelism occurs when an operation is executed simultaneously on several nodes[1]. This requires that the operands have been previously *partitioned*, i.e., horizontally fragmented, across the nodes. The way a base relation is partitioned is a matter of physical design. Typically, partitioning is performed by applying a hash function on an attribute of the relation, which will often be the join attribute. The set of nodes where a relation is stored is called its *home*. The *home of an operation* is the set of nodes where it is executed and it must be the home of its operands in order for the operation to access its operand. For binary operations such as join, this might imply repartitioning one of the operands. The optimizer might even sometimes find that repartitioning both the operands is of interest. Operator trees bear execution annotations to indicate repartitioning.

2.3 Extensibility

Extensibility was one of the goals of the project, which started when extensible query optimization was still an active research topic. Most extensible optimizers, if not all of them, rely on a rule-based declarative language [8, 23]. Rule-based languages seem to be a convenient way to specify

[1] A multiprocessor machine is modeled as a set of nodes. We call *node* one processor, together with its memory.

or modify some aspects of the optimization search space, e.g., the cost function. However, this approach suffers from several important problems.

- It is not easy to control the application of rules and guarantee termination of the optimization process.
- Using a very general language implies relying on a general, but inefficient, pattern-matching mechanism, unless pattern-matching is hand-coded. In the latter case the approach loses a lot of its appeal.
- Some aspects of query optimization, e.g. search strategies, do not gain extensibility from the rule-based approach.
- Rule-based languages suffer from the so-called brittleness problem, well known as a major problem of expert systems [22]. This means that they completely fail to support any request out of their strict domain of expertise.

Another approach to extensibility consists in relying on object-oriented techniques. These techniques foster extensibility because they help in building models of the application domain which are not polluted by implementation details. Moreover, one solution to the brittleness problem of rule-based languages mixes rules and objects into a more robust model of the application domain. Thus, we tried to use object-orientation intensively in the design and implementation of our prototype to get as much extensibility as possible. For example, we designed a hierarchy of optimization units (fragments of queries to be optimized independently), a hierarchy of algorithms to be used in execution plans, a hierarchy of search strategies, etc.

As far as we know, this approach had not been systematically used to increase extensibility of query optimizers. But, contrary to previous projects, we aimed at having an optimizer that could be extended by ourselves, not by outside implementors[2]. We achieved significant extensibility of the search strategy for join enumeration (See Section 3.4) and access methods, but only a limited extensibility of other aspects of optimization (relational operations, parallel algorithms) (See Section 4.1).

3 Design Decisions

In this section, we present the architecture of the optimizer and the solutions to the problems raised in the previous section. Although the optimizer is organized around class hierarchies, it is

[2]In other extensible database systems, the experience of external implementors proved to be very painful [12].

still meaningful to describe its functional architecture. After exposing the design decisions related to the architecture, we give an overview of the choices we made to design the search space, the cost model, and the search strategies.

Two important assumptions underlie the design of the optimizer. First, we assume that it is possible to optimize well at compile time. Second, we believe that it is not possible to rely only on traditional (sequential) optimization techniques, i.e., the optimizer must care about parallelism.

Regarding the first assumption, the interest of deferring some optimization decisions until execution is well-known [9] and is even more important in the case of parallel execution (especially in DM) because of the need to balance the load among the nodes. Although we believe that some optimization decisions should be deferred till execution time, we left this problem for future work and our optimizer outputs a single query execution plan, built at compile time.

Concerning the second assumption, we adopted a different approach than that of XPRS [13], where the best parallel execution plan is obtained by parallelizing the best sequential plan. In [34] we showed that it would not be reasonable to make the same assumption in DM. Even in SM, the domain of validity of this assumption seems rather narrow (a very large main memory, only hash-based join algorithms, no indices). This is why incorporating some knowledge of parallelism into the optimizer is a more extensible approach and was necessary in our system which aims both SM and DM architectures.

3.1 Architecture

The optimizer is part of the ESQL query compiler, as illustrated in Figure 2. The **query rewriter** analyses the query and transforms it into an equivalent query, to increase the optimization opportunities. For example, it deduces "implied" predicates. Consider the following ESQL query:

Select $R_2.*$ **from** R_1, R_2, R_3, R_4
where $R_1.A = R_2.A$ **and** $R_2.A = R_3.B$ **and** $R_3.C = R_4.C$

The rewriter transforms this query onto one having the additional predicate $R_1.A = R_3.B$. This will enable the optimizer to investigate more join permutations that do not introduce unnecessary Cartesian Products. The rewriter deals also with recursion, by identifying fixpoint recursion and choosing an algorithm for computing it.

The role of the **optimizer** is to find the best possible (ideally optimal) execution plan, i.e., the one whose cost estimate is the minimal among all investigated plans. So, the optimizer builds several plans and compares them by means of a *cost function*. The **parallelizer** expresses in a

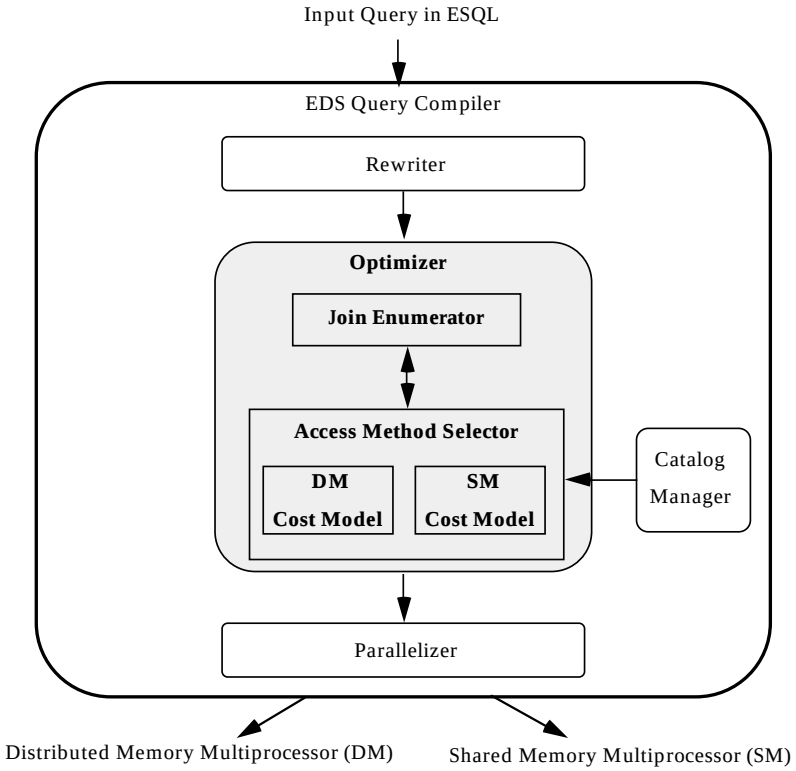

Figure 2: ESQL Query Compiler

low-level language the decisions made by the optimizer. It puts together some operations into the same process and manages the control of parallel execution (synchronizations and terminations).

The interface language between the rewriter and the optimizer is a form of extended relational algebra, called *Lera* (Language for extended relational algebra). It has constructs equivalent to the traditional algebraic operators, a fixpoint operator and an n-way select-project-join operator, called *filter*. The latter has been introduced to enable the rewriter to defer decisions regarding the ordering of selects, projects and joins, that should be postponed until the cost-based optimization step. The sample query is transformed by the rewriter into the following Lera program (notice the inclusion of a deduced predicate $R_1.A = R_3.B$), which uses the *filter* operator:

Result= filter $((R_1, R_2, R_3, R_4)$,
$((R_1.A = R_2.A)\text{and}(R_2.A = R_3.B)\text{and}(R_3.C = R_4.C)\text{and}(R_1.A = R_3.B), (R_2.*))$

To each algebraic operator corresponds one optimizer component. The **Access Method Selector** (AMS) is responsible for dealing with most of the operators, including selections, projections and binary joins. It chooses the best algorithm to perform the operation, its home, and estimates its cost, and the cost of the plan rooted at the operation. The **Join Enumerator** is responsible

for optimizing the *filter* operator. It builds an execution plan, capturing some join permutation, and calls the AMS, which returns a cost estimate for the generated plan.

The boundary between the rewriter and the optimizer is related to the fact that they reason, respectively, in a heuristic-based vs. cost-based fashion. Separating these components constitutes a quite usual approach, although problematic. It reduces the complexity of optimization, by constraining the search space, and eases the overall design. The problem is that some decisions traditionally taken by the rewriter (e.g., pushing selections before joins) should be deferred until the optimization step, to be decided by a cost-based strategy. This is a well-known open problem [12]. On the other hand, the separation between the Join Enumerator and the AMS, which are both part of the optimizer, does not suffer from a similar problem. The reason is that the AMS returns its result to the Join Enumerator which, thus, can explore all the alternatives if necessary and take cost-based decisions.

3.2 Search Space

Figure 3 shows four operator trees, that represent execution plans for the sample query. An operator tree is a labelled binary tree where the leaf nodes are relations of the input query and each non-leaf node is an operator node (e.g., join, union) whose result is an *intermediate* relation. A join node captures the join between its operands. Execution annotations (e.g., join algorithm) are not shown for simplicity. Directed (resp. undirected) arcs denote that the intermediate relation generated by a tree node is consumed in pipeline (resp. stored) by the subsequent node. Operator trees may be *linear*, i.e. at least one operand of each join node is a base relation (See Figure 3.(i), (ii) and (iii)), or *bushy* (See Figure 3.(iv)). We say that an operator tree corresponds to a *complete* execution plan if it captures all the relations of the input query (e.g., all the trees in Figure 3 represent complete plans). Otherwise, we say that the plan is *partial*.

The optimizer *search space* is characterized by the shape of trees it investigates. In most sequential optimizers (e.g., the optimizer of System R [26]), the search space for join enumeration is usually restricted to left-deep trees (See Figure 3.(i)). The reason for this heuristic restriction is that the space including all the tree shapes is much larger, while an optimal or nearly optimal plan can often be found in the much smaller space of left-deep trees. When join algorithms are not symmetric, which is the case for hash-joins, it is useful to distinguish left-deep and right-deep trees (See Figure 3.(i) and (ii)).

In contrast to sequential environments, in a parallel one tree formats other than left or right-deep seem interesting. For example, bushy trees (See Figure 3.(iv)) are the only to allow independent parallelism. Independent parallelism is useful when the relations are partitioned on disjoint homes,

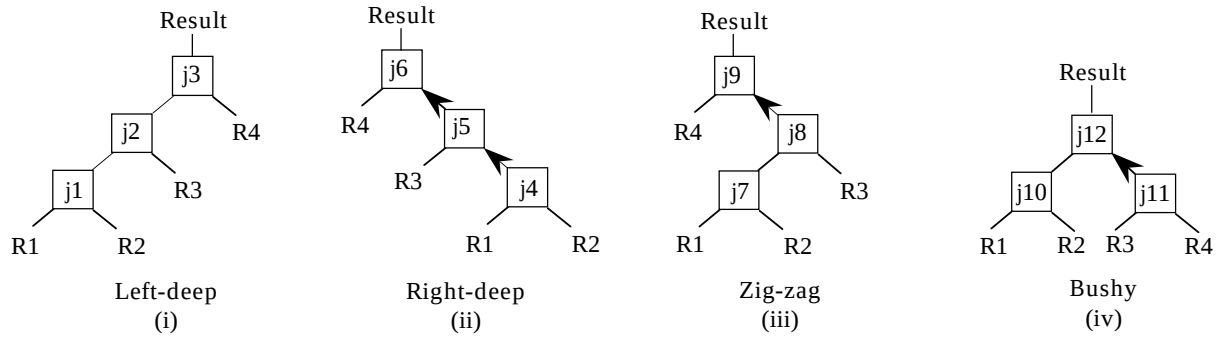

Figure 3: Execution Plans as Operator Trees

in the case of a DM machine. Suppose the sample query is to be executed on a DM machine, and that the relations are partitioned two (R_1 and R_2) by two (R_3 and R_4) on disjoint homes (resp. h_1 and h_2). Then, joins j_{10} and j_{11} could be independently executed in parallel by the set of nodes that constitutes h_1 and h_2.

Intra-operation parallelism is definitely the one to favor as much as possible. If the relations are not too small, a large degree of parallelism can be reached by data partitioning, and if data are uniformly partitioned, load balancing is easily achieved. For example, if relations R_1 and R_2 are partitioned on their join attribute A, the join j_{10} can be executed in an intra-parallel fashion on h_1.

When dataflow parallelism (i.e., due to pipeline) is profitable, zigzag trees, which are intermediate formats between left-deep and right-deep trees, can sometimes outperform right-deep trees due to a better use of main memory in SM machines [33]. A reasonable heuristic is to favor right-deep or zigzag trees when relations are partially fragmented on disjoint homes and intermediate relations are rather large. In this case, bushy trees will usually need more phases and take longer to execute. On the contrary, when intermediate relations are small, pipelining is not very efficient because it is difficult to balance the load between the pipeline stages. In any case it was very useful to be able to change the search space of the optimizer to explore various kinds of trees.

3.3 Cost Model

The optimizer cost model is responsible for estimating the cost of a given execution plan. In the EDS project, we had to deal with two target environments, DM and SM. Thus, we studied the differences between these architectures for the cost model and we finally designed a cost model that may be easily adapted to either architecture. In other words, the cost model may be seen as two parts: architecture-dependent and architecture-independent.

The architecture-independent part is constituted by the cost functions for operation algorithms,

e.g., nested loop for join and sequential access for select. If we ignore concurrency issues, only the cost functions for data repartitioning and memory consumption differ and constitute the architecture-dependent part. Indeed, repartitioning a relation's tuples in DM implies transfers of data across the interconnect, whereas it reduces to hashing in SM. Memory consumption in DM is complicated by inter-operation parallelism. In SM, all operations read and write data through a global memory, and it is easy to test whether there is enough space to execute them in parallel, i.e., the sum of the memory consumption of individual operations is less than the available memory. In DM, each processor has its own memory, and it becomes important to know which operations are executed in parallel on the same processor. Thus, for simplicity, we assume that the set of processors (home) assigned to operations to execute do not overlap, i.e., either the intersection of the set of processors is empty or the sets are identical. This will simplify the formula for response time with DM. It is however possible that two distinct operations have the same home, in which case the formula takes this into account. For example, in our prototype, the catalog allows to specify that two relations are partitioned on the same home, e.g., R_1 and R_2 are partitioned on the same home, and, thus, the select operation on these relations have the same home. In SM, the execution system dynamically balances the load among processors.

To take into account the aspects of parallel execution, we defined the cost of a plan as three components: total work (TW), response time (RT), and memory consumption (MC). TW and RT are expressed in *seconds*, and MC in *Kbytes*.

The first two components are used to express a trade-off between response time and throughput. The third component represents the size of memory needed to execute the plan. The cost function is a combination of the first two components, and plans that need more memory than available are discarded. Another approach [5] consists in using a parameter, specified by the system administrator, by which the maximum throughput is degraded in order to decrease response time. Given a plan p, its cost is computed by a parameterized function $cost_{(W_{RT},W_{TW})}()$ defined as follows:

$$cost_{(W_{RT},W_{TW})}(p) = \begin{cases} W_{RT} * RT + W_{TW} * TW & \text{if } MC \text{ of plan } p \text{ does not exceed the available memory} \\ \infty & \text{otherwise} \end{cases}$$

where W_{RT} and W_{TW} are weight factors between 0 and 1, such that $W_{RT} + W_{TW} = 1$.

A major difficulty in evaluating the cost is in assigning values to the weight of the first two components. These factors depend on the system state (e.g., load of the system and number of queries submitted to the system), and are ideally determined at run time. This is impossible since we perform static optimization. We finally restricted the cost components to response time and memory consumption, i.e., we suppose that only one query is submitted to the system at a time.

The cost function became:

$$cost(p) = \begin{cases} RT & \text{if MC of plan } p \text{ does not exceed the available memory} \\ \infty & \text{otherwise} \end{cases}$$

The response time of p, scheduled in phases (each denoted by ph), is computed as follows,

$$respTime(p) = \sum_{ph \in p} (max_{O \in ph}(respTime(O) + pipe\_delay(O)) + store\_delay(ph))$$

where O denotes an operation and $respTime(O)$ the response time of O. $pipe\_delay(O)$ is the waiting period of O, necessary for the producer to deliver the first result tuples. It is equal to 0 if the input relations of O are stored. $store\_delay(ph)$ is the time necessary to store the output results of phase ph. It is equal to 0 if ph is the last phase, assuming that the result are delivered as soon as they are produced.

The cost model, in a parallel environment depends on dynamic parameters. For example, the amount of available memory may have an impact on the choice of a scheduling strategy. Insufficient memory is a reason that forces an execution plan to be split into more phases. Memory size is usually unknown to the optimizer that operates at compile time. Parallelism introduces another crucial dynamic parameter which is the way the load is balanced among the processors. In some cases, the impact of dynamic parameters is limited. For example, in SM, if we do not suppose inter-operation parallelism, knowing the amount of available memory is not relevant, because only one operation is executed at a time. However, in the general case, some optimization decisions should be made at run time. One solution to this problem is to build several execution plans, put together by means of *choose* operators [9].

To estimate the cost of an execution plan, the cost model uses database statistics and organization information, such as relation cardinalities and partitioning. As usual, these statistics were not maintained automatically by the system, and are manually updated through queries on the metabase. Besides information on base relations, available in the metabase, the optimizer maintains, in a temporary catalog, information on intermediate relations. This is important when the optimized programs contain more than one operation, or when the join has more than two operands. Table 1 shows the contents of the metabase for the relations of the sample query.

3.4 Search Strategies

The optimizer considers each algebraic operation independently since the rewriter has already taken global reorganization decisions. Optimization of all the operations but the n-way select-project-join is quite straightforward: it consists of choosing the algorithm, and the home of the operation.

Relation	Cardinality	Tuple Size	Key	Indexes	Home
R_1	10000	66	$R_1.A$	Btree($R_1.A$)[a]	$R_1.A$(10,hash) on h_1 [b]
R_2	6686	28	$R_2.A$	Btree($R_2.A$)	$R_2.A$(10,hash) on h_1
R_3	25032	24	$R_3.C$	Btree($R_3.C$)	$R_3.C$(10,hash) on h_2
R_4	22249	39	$R_3.C$	Btree($R_4.C$)	$R_4.C$(10,hash) on h_2

Attribute	Type	Size	NDist[c]	Minimum	Maximum
$R_1.A$	integer	4	10000	1	10000
$R_2.A$	integer	4	6686	1	10000
$R_3.B$	integer	4	4944	1	10000
$R_3.C$	integer	4	5005	1	5005
$R_4.C$	integer	4	5005	1	5005

[a] Btree index on attribute $R_1.A$.

[b] Relation R_1 is partitioned on home h_1, composed of 10 nodes, using a hash function on attribute $R_1.A$.

[c] The number of distinct values in the relation.

Table 1: Contents of the metabase for the relations of the sample query.

The crucial issue in terms of search strategy is the join ordering problem, that is NP-complete on the number of relations [14]. A typical approach to solve the problem [26] is to use dynamic programming, which is a standard optimization technique. It is almost exhaustive and assures that the best of all plans is found. It incurs an acceptable optimization cost (time and space) when the number of relations in the query is small. However, this approach becomes too expensive when the number of relations is greater than 5 or 6. For this reason, there has been recent interest in *randomized* strategies, which reduce the optimization complexity but do not guarantee the best of all plans. Another way to cut off the optimization complexity would consist in adopting a heuristic approach [27]. We rather chose to implement randomized strategies, in order to investigate their adequacy in the parallel context, both from the point of view of need and quality of the optimal execution plan.

To implement several search strategies with a good degree of code reuse, we exploited the object-oriented paradigm, adopted in the implementation of the optimizer. We depicted the main algorithms (deterministic, randomized), in which functions are called to implement specific tasks. Taking advantage of late binding, these functions are implemented in several ways, thus changing the behavior of the optimizer. For example, in a deterministic algorithm, the choice of the next state to explore should be the least (resp. last) recently obtained one if the algorithm is breadth-first (resp. depth-first). The class hierarchy for the search strategies is shown in Figure 4. The approach is detailed in [18] and has proved to be very effective. It enabled the implementation of

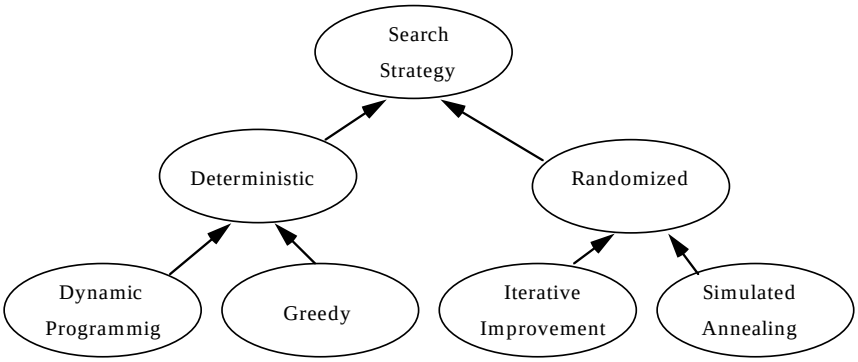

Figure 4: Search Strategy Class Hierarchy.

several strategies with reduced implementation effort. A variant of an existing strategy took just a few hours to be implemented.

Deterministic strategies proceed by *building* plans, starting from base relations, joining one more relations at each step till complete plans are obtained, as shown in Figure 5.(i). A greedy strategy builds only one such plan, by depth-first search, while dynamic programming builds all possible plans breadth-first. To reduce the optimization cost, partial plans that are not likely to lead to the optimal plan are *pruned* (i.e., discarded) as soon as possible. We show below the trace of our optimizer when running the dynamic programming strategy for the sample query in a right-deep search space. Each trace line shows the current node to which base relations are being joined. We trace the progress of the search strategy by displaying generated partial plans, each one characterized by the permutation of relations it captures, as well as its cost.

```
partial plan 2,1        cost = 0.410598
partial plan 1,4        cost = 417.138
partial plan 4,1        cost = 42.0721
partial plan 2,4        cost = 282.176
partial plan 4,2        cost = 28.5767
partial plan 3,4        cost = 189.674
partial plan 2,1,4      cost = 517.19
partial plan 4,1,2      cost = 97.3496
partial plan 2,4,1      cost = 282.808
partial plan 1,4,3      cost = 623.423
partial plan 4,1,3      cost = 189.662
partial plan 3,4,1      cost = 190.305
partial plan 2,4,3      cost = 488.46
partial plan 4,2,3      cost = 189.662
partial plan 3,4,2      cost = 190.305
```

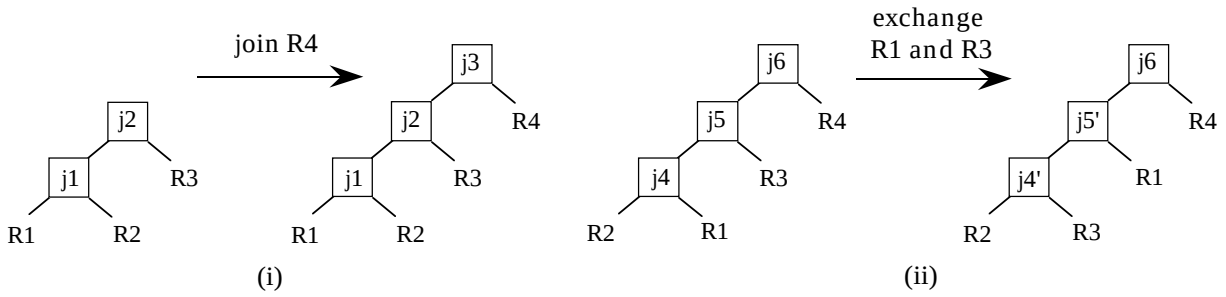

Figure 5: Optimizer Actions in Deterministic (i) and Randomized (ii) Strategies

Final plan 4,1,2,3 cost = 97.4022 Optimization Time: 0.45 seconds

Randomized strategies concentrate on searching the optimal solution around some particular points. They do not guarantee that the best solution is obtained, but avoid the high cost of optimization, in terms of memory and time consumption. First, one or more *start* plans are built by a Greedy strategy. Then, the algorithm tries to improve the start plan by visiting its *neighbors*. A neighbor is obtained by applying a random *transformation* to a plan. An example of a typical transformation consists in exchanging two randomly chosen operand relations of the plan, shown in Figure 5.(ii).

We have implemented different randomized strategies, such as Iterative Improvement [31] and Simulated Annealing [16]. They differ on the criteria for replacing the "current" plan by the transformed one and on the stopping criteria. We show below the trace of Iterative Improvement on the sample query in a right-deep search space. In the case of randomized strategies, our trace show the start state and the new state every time a transformation is accepted.

start state 4,2,1,3 cost = 107.501
exchange 4,1,2,3 cost = 97.4022

Final plan 4,1,2,3 cost = 97.4022 Optimization Time: 3.30 seconds

As the sample query has few relations, dynamic programming performs better than a randomized strategy. But this situation is inverted when the query has 7 relations or more. We see below the cost of the final plans and the elapsed time for optimizing a query with 12 relations on a right-deep search space. The randomized strategy could reduce the optimization time by a factor of 25, finding a plan which is only 0.5% worst.

Dynamic Programming:

Final plan 9,4,2,6,3,1,5,11,10,7,12,8 cost = 133.271 Optimization Time: 518.49 seconds

Iterative Improvement:
Final plan 3,6,4,9,2,5,1,7,11,10,8,12 cost = 139.952 Optimization Time: 21.12 seconds

3.5 Dynamic Programming for Parallel Execution Spaces

When building execution plans through dynamic programming, the optimizer systematically builds and compares *equivalent* partial plans through their cost estimates. It discards "expensive" partial plans which are "equivalent" to a cheaper one. The issue is: which plans are equivalent? At first glance, equivalent plans are those which capture the same relations. In System R, the equivalence criterion considered also the tuple order of the resulting relation. The reason is that in the presence of merge join, a partial plan with a high cost could lead to a better plan, if a sort operation could be avoided. In [33] and [20], we showed that parallelism complicates considerably the equivalence criterion.

To make it clear, let us look at two different partial plans for the sample query:

partial plan 4,2,3 cost = 189.662
partial plan 3,4,2 cost = 190.305

Looking at those trace samples, one would say that partial plan 3,4,2 should be pruned now, because it captures the same relations as partial plan 4,2,3 but has a higher cost. In a centralized environment, this could be an acceptable reasoning. But, let us see what happens when we join relation 1 to both of them.

partial plan 4,2,3,1 cost = 191.843
partial plan 3,4,2,1 cost = 190.937

Surprisingly, partial plan 3,4,2,1 is less costly! The reason is that the resulting relation of partial plan 3,4,2 is located at the same home as relation 1, contrary to partial plan 4,2,3. Thus, some repartitioning is needed for partial plan 4,2,3 before the join with relation 1 can be performed. Therefore, if we had pruned partial plan 3,4,2 in the previous step, we would have missed a good candidate to the optimal plan. So, the captured relations is not the only criterion of equivalence between plans.

At first, we implemented the equivalence criterion of a centralized environment, based on the captured relations only. But there are many features of a parallel partial plan that influence the

cost of subsequent plans built from them. The home of the resulting relation of a partial plan is relevant, when hash-based algorithms are used. As redistributing a relation is very expensive, the home of the resulting relation may impact future plans the same way as the tuple order face to merge join algorithms. The scheduling of a partial plan must also be considered, when comparing partial plans. Thus, when the home of the operation and the scheduling of the plan are not the same, the plans cannot be considered equivalent.

Changing the equivalence criterion reduced the effectiveness of pruning. As a consequence, many more partial plans are retained during the search than in a centralized environment, increasing considerably the optimization cost. Thus, running dynamic programming on reasonable queries (e.g., 8 relations or more) in a parallel bushy search space often caused the optimizer to run out of space. This was one motivation to use randomized strategies.

The equivalence criterion in DM is different from that in SM. In SM, two relations have the same home if they are partitioned on the same attribute, using the same function. In DM, besides this, the set of nodes storing the relations must be the same. Thus, the equivalence criterion in DM is even more restrictive than in SM and the increase in the search space is more significant.

4 Prototypes

In this section, we discuss our experience in implementing the optimizer prototype and its integration in the ESQL compiler. We also explain the experiments conducted using the optimizer and the DBS3 system, and give our impression of the used benchmarks.

4.1 Implementation

Implementation started late 1989, after a great portion of the design phase had been done. First, we implemented a package for list and set management. Then, we implemented the Access Method Selector module together with the interfaces with the other ESQL compiler components, namely the Rewriter, the Parallelizer, and the Catalog Manager. With this, the optimizer was able to treat many Lera operations, except the n-way join and the fixpoint operator.

The Join Enumerator started to be implemented at the beginning of 1991, when Rosana Lanzelotte joined the project. The architecture of the optimizer proved, then, to be very appropriate for modular software development. The Join Enumerator strategies could be easily added to the Optimizer, without changing the previous code. From then on, both modules evolved independently, and no interfacing problems ever occurred.

At the beginning of 1992, we had a very stable implementation and could start measurement experiments.

The choice of C^{++} [30] as the implementation language presented a mix of pros and cons. The pros are related to the extensibility and modularity the project gained and are well known. However, except for the search strategy of the Join Enumerator, we only achieved a limited extensibility which should rather be called adaptability. We realized that C^{++} is not versatile enough to support directly the modeling of a very general query optimizer. Although C^{++} is object-oriented like several knowledge representation languages it is still too close to implementation details.

Completely relying on C^{++} for extensibility is also problematic when one wants to add a new feature without recompilation of the optimizer. For example if a new join algorithm is added, a declarative language is necessary to specify it (especially its cost and when it can be used) without recompiling the optimizer.

Another con is related with the very high learning curve of C^{++}. No one in the team have had a significant previous experience with the language, and it was difficult to use it in an adequate style.

Still another relevant con is that C^{++} does not have a garbage collector! At first, this fact did not bother us much. But, as the experiments with large queries started, memory consumption during optimization proved to be a critical issue. We had to spend significant implementation effort in saving memory and, then, in fixing bugs introduced by the code added to free the "unused" memory. A last con is related to the absence of a programming development environment and of standard packages for manipulating containers. We implemented the later using macros because, at that time, genericity was not supported by C^{++}. This resulted in portability problems as explained in the following section.

Version maintenance has not been a more critical factor because the team was not a large one. Nevertheless, when we had visitor " implementors", we had to take care not to mix up incompatible code in the same version.

4.2 Integration

Although the integration between modules inside the optimizer team has occurred smoothly, the same cannot be said about integrating the optimizer with the modules of the EDS project that were developed by other teams, namely the Rewriter, the Catalog Manager and the Parallelizer.

The implementation part of the EDS project was conducted by many teams using different hardware and software, especially different compilers, C and C^{++}. Indeed, some teams did not

have the same compiler version, particularly the C^{++} compiler along with its tools, lex^{++} and $yacc^{++}$. This was a source of problems during the integration process as every team was not aware of the various requirements of the hardware and software used by other teams. The problem became even harder when the integration is conducted on the machines where the debugging tools can not be used (or were not appropriate) for various reasons, e.g., the debugger did not work when the executable is the result of sources written in C and C^{++}.

Because at that time, genericity was not supported by our C^{++} compiler, we implemented it using macros. Sometimes these macros were too long (tens of lines) and were not accepted by certain preprocessors. Needless to say that the debugging was difficult and that we were obliged to simplify the macros.

The last factor that made the integration difficult is that the interface between certain modules was data structure-based, hence, not reliable. A text-based interface would be safer, but slows down the overall performance. Indeed, the producer must put the output in a defined format, and the consumer has to parse its input according to this format. The optimizer interacts with three modules: the Rewriter and the Parallelizer through a text-based interface, and the Catalog Manager, through a data structure-based interface. About 20% of the code is dedicated to the interfacing tasks.

4.3 PEM Application

The Portfolio Club Experimental Model (PEM) was designed to provide a realistic experimental base for complex query definition, evaluation and benchmarking on the EDS system [17]. PEM constitutes a simplified model for an application on share market and investment/portfolio management. The PEM schema contains 20 relations joinable through foreign keys. There are three kernel relations, "enterprises", "investors" and "holdings", to which disjoint sets of the remaining relations are joinable. Besides, "enterprises" and "investors" are joinable, as well as "enterprises" and "holdings".

The testbed catalogs, used by the optimizer, are generated automatically using three key parameters which correspond to the cardinalities of the kernel relations. The cardinalities of relations vary with respect to each other considerably, as in real applications. The catalog describes also the partitioning of relations, the number of nodes and the attributes used by the partition function. Disjoint subsets of the relations were partitioned on three disjoint homes, containing one kernel relation plus the set of relations whose foreign key is the primary key of the kernel relation.

The benchmarks for database and transaction processing systems (e.g., Wisconsin and AS3AP [10]) were mainly designed to measure the performance of the relational systems to process sim-

ple and complex queries (typical of decision-support applications) on a synthetic database. Key parameters, such as relation cardinality, predicate selectivity, and indexes, are varied. There were three motivations for choosing the PEM application. First, among the existing benchmarks, the largest query contains 4 relations (AS3AP benchmark), whereas we need queries of larger size (e.g., containing more than 10 relations). Second, they allow to test only a subset of the optimizers functionalities, e.g., their ability to add implied selection predicates, to choose an index, and if many to choose the best one. Since the join ordering for queries of larger size became very important as an optimizer functionality, using different search strategies, it is urgent to design new benchmarks, adapted for the task of testing optimizer search strategies. In the mean time, ad-hoc benchmarks were used for the need of a paper. As a consequence, the comparison between published performance became very difficult. Third, the PEM application is based on a real model, in contrast to benchmarks like Wisconsin and AS3AP where relations are identical, i.e., they have the same schema and same statistics for relation attributes. Our optimization search strategies are cost-based, thus, they are very sensitive to statistics on attributes (e.g., the number of attribute distinct values in a relation) and relation tuple size. The latter is especially important for distributed memory environment when data are moved between processors via an interconnect.

The appendix contains an example query, in ESQL, from the PEM application and the corresponding algebraic representation.

4.4 Benchmarks and Experiments

When working on parallel database query optimization, we performed three kinds of experiments, each for a different purpose and using a different benchmark. In [33], we proposed a new format for scheduling linear plans, called zigzag, and compared it to the slicing strategy proposed in [25], using the Wisconsin benchmark.

The main objective of the second experiment was to investigate the impact of parallelism on the optimization cost; and which is the best way to reduce the optimization cost, either to restrict the search space or to use a randomized search strategy.

The experiments, conducted using the PEM application, led to some interesting conclusions. Concerning the first objective, we realized that the size of a parallel search space is remarkably larger than the corresponding non-parallel one. Many more alternative execution plans are investigated for a given input query, due to the fact that different partitioning and scheduling are possible for each PT shape. This causes a considerable increase on the optimization cost. In an exhaustive strategy, as dynamic programming, this increase may incur intractability for queries with more than 8 relations.

Concerning the second objective, our guess was that restricting the search space, an approach adopted by most parallel optimizers, was not the best choice. We found that restricting the search space may often lead to missing the optimal plan. For example, the optimizer finds a better plan for the sample query when running on a bushy space in DM environments, as shown below. The reason is that, as the relations are partitioned on disjoint homes (R_1 and R_2, R_3 and R_4), independent parallelism is worth exploiting.

Right-deep space:	**Bushy Space:**
Final plan 4,1,2,3 cost = 97.4022	Final plan (4,(2,1)),3) cost = 52.5519

Once restricting the search space proved not to be a good choice, how to deal with the increase in the optimization cost? Our conclusion is that exploring large search spaces, e.g., bushy space, using randomized search strategies, specially with some improvements, is better than using the Dynamic Programming strategy in a small search space. Randomized strategies had not been previously proposed to parallel optimization.

The last experiment concerns the validation of the optimizer cost model on the DBS3 system, implemented on the shared-memory Encore Multimax multiprocessor. We conducted this work using the AS3AP benchmark. The results obtained show that the predicted execution time is very close to the time measured after executing the query on the DBS3 system, under various conditions and even when varying key parameters such as the number of threads to implement an operation, predicate selectivity, relation cardinality, and the access method to relation tuples (scan or indexed).

5 Conclusion

In this paper, we have reported on our research and development experience with the parallel query optimizer of the EDS project. This project last from 1989 to 1993 and has successfully delivered a parallel database server. Commercialization plans by the industrial partners are beyond the scope of this paper. The optimizer has to deal with the advanced features of the ESQL query language (objects and recursion) and target parallel execution environments which can be either on distributed or shared memory platforms. As the EDS project involved industrial partners, the goal was to build an industrial-strength prototype, capable of supporting business data processing applications with mixed workloads of short (OLTP-like) transactions and complex decision-support queries.

In order to address all these challenges successfully, we had to capitalize on experience with relational query optimizers and to set priorities. Thus, we decided to adopt simple approaches to

treat object path expressions and recursion, using rewriting techniques, in order to concentrate on the impact of parallelism on optimization.

The requirement of porting the optimizer to two different execution environments led us to think of an extensible design applicable to all the optimizer components, from the cost model to the join enumerator and search strategy. Although the typical rule-based approach proved well adapted to an extensible cost model, this was not so for the join enumerator because we wanted to be able to change the algorithm itself. Therefore, we adopted an object-oriented design approach, which gave us the degree of adaptability, modularity and code reuse needed.

The main research results of our optimizer project have been presented in several papers. We have designed several techniques to optimize the access to objects through path expressions [21]. Having adopted a rewriting approach to deal with objects and recursion, we realized that this could lead to missing the best plan [19]. With respect to parallelism, we proposed a new way of scheduling execution plans, using zigzag trees, which are well suited to shared memory environments with resource contention [33].

We defined a cost metric that captures the main aspects of parallel execution environments, i.e., the operation scheduling and relation home [33]. We have implemented several search strategies adapted for join enumeration in parallel optimization [20]. An important result has been to experimentally show that randomized search strategies enable an optimizer to cope with the much larger search spaces incurred by parallelism.

In retrospect, the industrial context of the project (that we initially perceived as too constraining) was useful to set implementation and experimentation priorities. The early choice of C^{++} as implementation language caused difficult problems because of missing functionalities like garbage collection and genericity. Some of the problems are now better handled with version 3.0 [4]. The optimizer prototype has been validated using the PEM pilot application which features complex join queries and using the AS3AP benchmark. Experiments have been useful to measure the optimizer effectiveness and accuracy. The industrial context has also taught us that we, researchers, don't pay enough attention to software engineering and learn it the hard way at integration time.

A number of open issues make parallel query optimization still an exiting topic. One open problem remains the isolation between some modules of the query processor. For example, the isolation between the rewriter and the optimizer prevents some rewriting tasks from exploiting cost-based choices. Also, since optimization is carried out entirely at compile time, the optimizer cannot consider relevant dynamic parameters such as the current load at run time. Finally, the validation of the cost model has not been completed. Our current objective is to discover the shape of the cost function for our parallel execution environment, to better explain our success in using

randomized strategies [15].

References

[1] B. Bergsten, M. Couprie, and P. Valduriez. Prototyping DBS3, a shared memory parallel database system. In *Proc. Int. Conf. on Parallel and Distributed Information Systems*, pages 226–234, 1991.

[2] H. Boral and al. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[3] D.J. Dewitt and al. The GAMMA database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[4] EDS Database Group. EDS-Collaborating for a high-performance parallel relational database. In *Proc. ESPRIT Conf.*, Brussels, 1990.

[5] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[6] S. Ganguly, W. Hasan, and R. Krishnamurty. Query optimization for parallel execution. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 9–18, 1992.

[7] G. Gardarin and P. Valduriez. ESQL2: an extended SQL2 with F-logic semantics. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 320–327, 1992.

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

[9] G. Graefe and D.J. Dewitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1987.

[10] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 358–366, 1989.

[11] Jim Gray, editor. *The Benchmark Handbook.* Morgan Kaufmann, 1991.

[12] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[13] W. Hong and M. Stonebraker. Optimization of parallelism query execution plans in XPRS. In *Proc. Int. Conf. on Parallel and Distributed Information Systems*, pages 218–225, 1991.

[14] T. Ibaraki and T. Kameda. Optimal nesting for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[15] Y.E. Ioannidis and Y. Cha Kang. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 168–177, 1991.

[16] Y.E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 9–22, 1987.

[17] J. Jorgensen, S.M. Kellett, and N.C. King. Portfolio club experimental model. Technical Report EDS.DD.11I.0005, EDS ESPRIT project, 1990.

[18] R.S.G. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. Int. Conf. on Very Large Data Bases*, pages 363–373, 1991.

[19] R.S.G. Lanzelotte, P. Valduriez, and M. Zaït. Optimization of object-oriented recursive queries using cost-controlled strategies. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 256–265, 1992.

[20] R.S.G. Lanzelotte, P. Valduriez, and M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proc. Int. Conf. on Very Large Data Bases*, pages 493–504, 1993.

[21] R.S.G. Lanzelotte, P. Valduriez, M. Ziane, and J.P. Cheiney. Optimization of nonrecursive queries in OODBs. In *Proc. Int. Conf. on Deductive and Object Oriented Databases*, pages 1–21, 1991.

[22] D. Lenat. Ontological versus knowledge-based systems. *IEEE Transactions on Knowledge-based and Database Systems*, 1(1):84–88, March 1989.

[23] G. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 18–27, 1988.

[24] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 110–121, 1989.

[25] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proc. Int. Conf. on Very Large Data Bases*, pages 469–480, 1990.

[26] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, 1979.

[27] D. Shasha and T. Wang. Optimizing Equijoin queries in Distributed Databases Where Relations Are Hash Partitioned. *ACM Transactions on Database Systems*, 16(2):279–308, 1991.

[28] M. Stonebraker and al. The design of XPRS. In *Proc. Int. Conf. on Very Large Data Bases*, 318-330, 1988.

[29] M. Stonebraker and L.A. Rowe. The design of POSTGRES. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 340–355, May 1986.

[30] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 1986.

[31] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 367–376, 1989.

[32] P. Valduriez. Parallel database systems: open problems and new issues. *Distributed and Parallel Databases - an International Journal*, 1(2):137–165, April 1992.

[33] M. Ziane, M. Zaït, and P. Borla-Salamet. Parallel query processing with zigzag trees. *VLDB Journal*, 2(3):277–301, July 1993.

[34] M. Ziane, M. Zaït, and H. H. Quang. The impact of parallelism on query optimization. In *Fifth Workshop on Foundations of Models and Languages for Data and Objects*, pages 127–138, 1993.

Appendix

In this appendix, we give another example of ESQL query, from the PEM application. This query was used in our experimental studies on search strategies [20]. It retrieves all shares, from *share* relation, and for each share gets information on the corresponding market sector, the dividend's price earning ratio, the share title, the nominal value country, the name and country of the investor, and the current volume of holding.

```
SELECT sector = M.mkt_sector,
       P_E = D. p_e_ratio,
       title = S.share_title,
       registered_in = N.nom_country,
       lives_in = I.investr_ctry,
       held_by = I.investr_name,
       holds = H.port_holding
FROM mkt_sector M,
     dividend D,
     share S,
     nominalvalue N,
     investor I,
     port_holding H
WHERE S.share_id = N.share_id
AND   N.share_id = D.share_id
AND   S.mkt_sectr_id = M.mkt_sectr_id
AND   S.share_id = H.share_id
AND   H.investr_id = I.investr_id
```

The corresponding *Lera* program, produced by the Rewriter, is[3]

```
Result=filter(
       (mkt_sector (M), dividend (D), share (S), nominalvalue (N), investor (I),
        port_holding (H)),
       ((S.share_id=N.share_id) AND (N.share_id=D.share_id) AND
        (S.mkt_sectr_id=M.mkt_sectr_id) AND (S.share_id=H.share_id) AND
        (H.investr_id=I.investr_id) AND (S.share_id=D.share_id) AND
```

[3]Notice the inclusion of three deduced predicates: S.share_id=D.share_id, N.share_id=H.share_id, and D.share_id=H.share_id.

```
 (N.share_id=H.share_id) AND (D.share_id=H.share_id)),
(M.mkt_sector, D. p_e_ratio, S.share_title, N.nom_country, I.investr_ctry,
 I.investr_name, H.port_holding))
```

The involved relations are partitioned as follows:

- *mkt_sector* is partitioned on 1 node by hashing on attribute *mkt_sectr_id*;
- *dividend*, *share*, and *nominalvalue* are partitioned on 10 nodes by hashing on attribute *share_id*;
- *investor* and *port_holding* are partitioned on 10 nodes by hashing on attribute *investr_id*.

We show below a sample of the trace of our optimizer when running the dynamic programming strategy for the above query in a right-deep search space. Recall that each trace line shows the current node to which base relations are being joined. We trace the progress of the search strategy by displaying generated partial plans, each one characterized by the permutation of relations it captures, as well as its cost.

partial plan 0,2,3,1	cost = 0.877922
partial plan 0,2,1,3	cost = 1.0598
partial plan 1,2,0,3	cost = 0.964922
partial plan 1,2,3,0	cost = 1.02169
partial plan 0,2,1,5	cost = 517.19
partial plan 0,2,5,1	cost = 417.744
partial plan 5,2,0,1	cost = 97.3449
partial plan 1,5,2,0	cost = 283.334
partial plan 0,2,3,5	cost = 764.605
partial plan 0,2,5,3	cost = 417.744
partial plan 5,2,0,3	cost = 122.086
partial plan 2,5,3,0	cost = 418.409
partial plan 5,2,3,0	cost = 122.731
partial plan 0,2,5,4	cost = 623.397
partial plan 5,2,0,4	cost = 189.662
partial plan 2,5,4,0	cost = 629.741
partial plan 5,2,4,0	cost = 195.98
partial plan 0,2,1,3,5	cost = 191.778
partial plan 0,2,5,1,3	cost = 417.753

partial plan 5,2,0,1,3	cost = 99.4584	
partial plan 1,5,2,0,3	cost = 283.343	
partial plan 0,2,5,1,4	cost = 420.696	
partial plan 5,2,0,1,4	cost = 97.354	
partial plan 0,2,5,4,1	cost = 624.029	
partial plan 5,2,0,4,1	cost = 190.294	
partial plan 0,2,5,3,4	cost = 420.696	
partial plan 5,2,0,3,4	cost = 122.095	
partial plan 0,2,5,4,3	cost = 624.029	
partial plan 5,2,0,4,3	cost = 190.294	
Final plan 5,2,1,3,4,0	cost = 99.4674	Optimization Time: 02:47 seconds

In the following program, produced by the optimizer, T_i, i=1..4, represents an intermediate relation, the *pipe* annotation on a relation indicates that the join operation can start consuming the relation tuples before it (relation) is completely produced, whereas the *stored* annotation indicates that the join operation has to wait till the relation is completely produced. For simplicity, we replaced the attribute name by the attribute rank in the relation. For each join operation, the optimizer specifies its home, the join algorithm, and the repartitioning of the operands, if needed.

T_1 = **join** (port_holding {pipe}, share {stored}, (share.1 = port_holding.3),
(port_holding.4, port_holding.1, port_holding.3, share.2, share.1,
share.3))
{``join takes place on *share* home'', ``send *port_holding* to *share* home'',
``perform the join using the nestedLoop algorithm''}

T_2 = **join** (T_1 {pipe}, mkt_sector {stored}, (T_1.6 = mkt_sector.1), (T_1.4, T_1.1,
T_1.5, T_1.2, T_1.3, mkt_sector.2))
{``join takes place on *mkt_sector* home'', ``send T_1 to *mkt_sector* home'',
``perform the join using the nestedLoop algorithm''}

T_3 = **join** (T_2 {pipe}, dividend {stored}, (T_2.3 = dividend.1) AND (dividend.1 =
T_2.5), (T_2.6, T_2.1, T_2.2, T_2.3, T_2.4, T_2.5, dividend.4, dividend.1))
{``join takes place on *dividend* home'', ``send T_2 to *dividend* home'',
``perform the join using the nestedLoop algorithm''}

T_4 = **join** (T_3 {pipe}, nominalvalue {stored}, (T_3.4 = nominalvalue.1) AND
(nominalvalue.1 = T_3.8) AND (nominalvalue.1 = T_3.6), (T_3.1, T_3.7,
T_3.2, T_3.3, T_3.5, nominalvalue.5))
{``join takes place on *nominalvalue* home'', ``no transfer'',
``perform the join using the nestedLoop algorithm''}

Result= **join** (T_4 {pipe}, investor {stored}, (investor.1 = T_4.5), (T_4.1, T_4.2,
T_4.3, T_4.6, investor.10, investor.3, T_4.4))
{``join takes place on *investor* home'', ``send T_4 to *investor* home'',
``perform the join using the nestedLoop algorithm''}