*VLDB Journal,* 2,277-301(1993), Michael Carey and Patrick Valduriez, Editors
©VLDB

# Parallel Query Processing With Zigzag Trees

**Mikal Ziane, Mohamed Zaït, and Pascale Borla-Salamet**

*Received December 1, 1992; revised version received February 1, 1993; accepted March 15, 1993.*

**Abstract.** In this article, we describe our approach to the compile-time optimization and parallelization of queries for execution in DBS3 or EDS. DBS3 is a shared-memory parallel database system, while the EDS system has a distributed-memory architecture. Because DBS3 implements a parallel dataflow execution model, this approach applies to both architectures. Using randomized search strategies enables the exploration of a search space large enough to include zigzag trees, which are intermediate between left-deep and right-deep trees. Zigzag trees are shown to provide better response time than right-deep trees in case of limited memory. Performance measurements obtained using the DBS3 prototype show the advantages of zigzag trees under various conditions.

**Key Words.** Search space, pipeline, fragmentation, cost function.

## 1. Introduction

A Database System on Shared Store (DBS3) is a shared-memory parallel database system (Bergsten et al., 1991) being developed by Bull Research Center and Institut National de Recherche en Informatique et en Automatique (INRIA) in the context of an Esprit project. The goal of DBS3 is to provide high-performance for queries expressed in ESQL (Gardarin and Valduriez, 1992), a conservative extension of SQL with object and deductive capabilities. It is optimized towards both on-line transaction processing (OLTP) and decision-support (more complex) queries. OLTP queries require high-throughput, while decision-support queries require good response times.

Two important considerations have guided the design of DBS3. First, DBS3 implements a parallel dataflow execution model based on fragmented data placement similar to distributed-memory (shared-nothing) systems like BUBBA (Boral et al.,

Mikal Ziane, Ph.D., is Assistant Professor, Université Paris 5, Paris, France, and Researcher, Projet Rodin, Institut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, F-78153 Le Chesnay Cedex, France; Mohamed Zaït, is a Ph.D. student, Projet Rodin, INRIA; Pascale Borla-Salamet, Ph.D., is Researcher, Bull Research Center, Les Clayes sous Bois, France.

1990) and GAMMA (DeWitt and Gray, 1990). This allows us to take advantage of automatic load balancing while reducing access conflicts to the shared memory. Second, the ESQL compiler translates a query into an optimized parallel program that exploits both inter- and intra-operation parallelism and yields decentralized execution control.

In this article, we describe our approach to parallel query processing in DBS3, which relies on the compile-time optimization and parallelization of execution plans. As in centralized query processing (Selinger et al., 1979), the major problem to be addressed is that of dealing with a large space of parallel execution plans. In the centralized case, the search space gets larger as the query becomes more complex; e.g., it includes many joins (Swami, 1989), it deals with complex objects (Lanzelotte and Valduriez, 1991), or it includes recursion (Lanzelotte, 1992).

Even for a reasonable query (e.g., $< 10$ joins), a parallel dataflow execution model with fragmented data placement yields a very large range of alternative execution strategies. We believe that considering a large search space, rather than relying on restrictive heuristics, is important because a large variety of parallel execution plans provides a superior trade-off between response time minimization and throughput maximization.

Optimizing queries for parallel execution is considered an open problem (DeWitt and Gray, 1990). Hong and Stonebraker (1991) drastically restricted the optimization search space by adopting two heuristic assumptions: the buffer size independent hypothesis and the two-phase hypothesis. However, these assumptions rely on a restricted execution model (left-deep trees only, no inter-operation parallelism). Ganguly et al. (1992) addressed the problem of minimizing response time subject to constraints on throughput, and extended a dynamic programming search algorithm to cope with dependencies among optimization choices. However, the authors considered only left-deep trees and did not mention any experimentation.

Our approach explores a search space large enough to include intermediate formats between left-deep and right-deep trees (Schneider and DeWitt, 1990). These formats, called *zigzag trees,* can lead to a better response time than those of right-deep trees in cases of limited memory. To avoid a prohibitive optimization time, we use randomized search strategies (Lanzelotte and Valduriez, 1991). Our approach applies both to shared-memory and distributed-memory architectures, avoids the need for a centralized scheduler, and enables compile-time optimization of dataflow control (Borla-Salamet et al., 1991). Performance measurements obtained using the DBS3 prototype show the advantages of zigzag trees under various conditions.

This article is an extended version of Ziane et al. (1993). The main additions are: (1) a comparison to segmented right-deep trees (Section 2.3), (2) a more detailed execution model (Section 3.2), (3) a description of the compilation process (Section 4.1), (4) a discussion of the dependencies among optimization choices (Section 4.2.3), and (5) one more experiment varying the size of intermediate results.

This article is organized as follows. Section 2 analyzes alternative approaches to parallelism and shows the advantages of zigzag trees. Section 3 describes our parallel data and execution models. Section 4 presents our query processing approach, including our cost model and search strategies. The experiments are detailed in Section 5. Section 6 concludes the work.

## 2. Shape and Scheduling of Operator Trees

After analyzing alternative approaches to parallelism, we show that zigzag trees can lead to better response time than sliced right-deep trees (Schneider and DeWitt, 1990) under some conditions. We discuss the use of bushy trees, and compare zigzag trees to segmented right-deep trees (Chen et al., 1992*a*).

### 2.1 Alternative Approaches to Parallelism

Hong and Stonebraker (1991) suggested that a parallel query execution plan is a parallelization of some sequential plan. Graefe (1990) was able to create parallelism in a sequential plan without modifying the sequential algorithms by adding an *exchange operator.* Such approaches ease the adaptation of sequential techniques to parallel environments. However, using a sequential (intermediate) plan might induce arbitrary restrictions. Our approach directly exploits the potential parallelism of queries.

Although the input query of an optimizer typically is represented by a graph of operations, such graphs impose only very loose constraints on their execution. The execution of a plan is described in terms of atomic actions which are, in our context, tuple productions or consumptions. An operation graph imposes only a partial order on the set of actions, and such freedom introduces parallelism in plan execution. The partial order reflects the obvious constraint that a tuple must be produced before it is consumed, as well as other constraints imposed by the semantics of some operations (e.g., the difference operation cannot produce any tuple before its second argument has been completely consumed, unless it is sorted).

Execution models usually constrain further the execution of operation graphs. For example, most of them limit the possibility of being consumed in pipeline to one operand. Most (sequential or parallel) algorithms require that one operand be completely produced before any tuple of the other operand can be consumed. This is the case for hash-based join algorithms, which typically consist of two consecutive phases: build and probe. Note that some constraints are introduced by the algorithms while the semantics of the operations would allow more freedom. For example, a hash-join algorithm was proposed only recently that allows both join operands to be consumed in pipeline (Wilschut and Apers, 1991).

We consider only non-recursive queries, which are typically represented by operator trees. We concentrate on trees of join operations and consider only hash-based algorithms because they have been shown to be the most efficient for equi-joins (Schneider and DeWitt, 1989). Schneider and DeWitt (1990) adopted a convenient notation for capturing trees of hash-join algorithms in which the right operand is consumed in pipeline while the left operand is blocked. We explicitly denote such constraints in our parallel execution plans by annotating arcs with *pipe* or *seq*.

Most execution models require that plans be sliced into sequential phases. In such models, each operation of the execution plan is completely executed during one specific phase, thereby requiring that the execution of an operation cannot overlap with two phases. Consequently, a slicing strategy determines non-overlapping fragments (subtrees) of the execution plan, whose operations are executed simultaneously. For linear trees, i.e., trees in which at least one argument of each join operation is a base relation (as opposed to an intermediate relation), specifying a pipe or a seq annotation for each arc is enough to determine a slicing strategy. Schneider and DeWitt (1990) compared two extreme cases among the different slicing strategies of linear trees, namely the strategies associated with left-deep and right-deep trees. In this article, we also consider intermediate cases which we call zigzag trees.

In a left-deep tree, each pipe operand is a base relation, which means that a pipeline is only possible between a hash-join and a selection operation. In this strategy, plans are sliced into fragments that consist roughly of a join and a selection (Hong and Stonebraker, 1991). To be more precise, in hash-based join algorithms, the seq annotation is indicated on an arc between the build node and the probe node. Thus, a fragment may include the probe part of one join and the build part of the next one. Note that in Hong and Stonebraker (1991) a fragment may also consist of two selection nodes consumed by a *nestloop* node, itself consumed by a probe node. A pipeline is not restricted to a single operand for nestloop nodes because XPRS actually implements such a "pipeline" execution by a single operation, i.e., without real parallelism. Finally, considering only left-deep trees (Hong and Stonebraker, 1991) is consistent with the decision to avoid inter-operation parallelism, because each fragment is made of a single operation.

In a right-deep tree, each blocked operand (annotated with *seq*) is a base relation. Thus, all the intermediate relations can be consumed in pipeline if enough resources (memory and processors) are available (Schneider and DeWitt, 1990). This means that right-deep trees can be executed in only two phases, while bushy trees often need more phases. If the usual constraint, which requires that only one operand of an operation be consumed in pipeline, does not hold, then a right-deep tree may be consumed in a single phase. However, it is possible to execute a right-deep tree with more phases, e.g., in case of limited memory, using static right deep scheduling (Schneider and DeWitt, 1990). This technique is expressed easily in our model by transforming some pipe annotations into seq annotations.

## 2.2 Advantages of Using Zigzag Trees

*Static right deep scheduling* slices a right-deep tree into phases, so that each resulting fragment is expected to fit in memory, and spools to disk the temporary results between two phases. In this section, we propose an alternative approach which avoids spooling intermediate results to disk using zigzag trees.

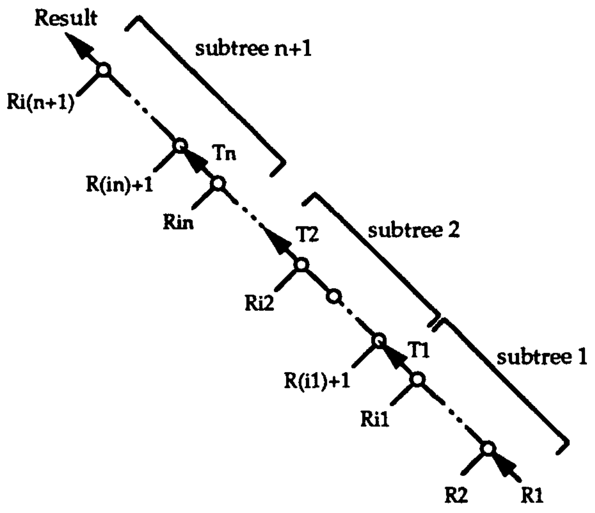
It is sometimes possible to avoid unnecessary I/Os, by keeping in memory the temporary relation resulting from each subtree (but the last one) of a sliced right-deep tree. This implies taking into account the size of the temporary relation when deciding where to cut the right-deep tree. While saving I/Os, the disadvantage of this approach is that less permanent relations can fit into memory together; thus the right-deep tree has to be sliced into more pieces. This technique sacrifices some parallelism, possibly at the expense of response time, to avoid unnecessary I/Os. The advantages can outweigh the costs, depending on the I/O system contention, and the technique can be used to help balance I/O and CPU utilization. Furthermore, it is possible to better exploit the idea by using zigzag trees instead of sliced right-deep trees.

With the preceding technique, intermediate relations are used in a right-deep tree as probing relations. This makes their presence in memory not very useful, because probing relations are consumed in pipeline. By using these temporary relations as building relations, we transform right-deep trees into zigzag trees. In Schneider and DeWitt (1990), the intermediate formats considered between right-deep and left-deep trees are bushy trees. However, right-deep and left-deep trees are two extreme formats for linear trees. Bushy trees capture the same level of abstraction as linear trees, while zigzag trees are strategies for linear trees in which base relations are either blocked or consumed in pipeline.

Zigzag trees may be more advantageous than sliced right-deep trees in cases of limited memory, especially when temporary relations are not staged to disk. The rationale is to "turn right" instead of simply slicing the right-deep trees. Turning right means that the temporary relation produced by the already-built right-deep subtree will be used as a building relation in the following hash-join operation, rather than as a probing relation, as it would if the right-deep tree had simply been sliced. Note that choosing a temporary relation as a building relation is particularly useful when the build phase can be avoided, i.e., when it is already hashed on the attribute of the next join. Then another right-deep subtree is built in the same way and possibly turns right again if it runs out of memory. The process is repeated the same way, building right-deep subtrees and turning right, as many times as necessary to fit each right-deep subtree into memory.

When a temporary relation is kept in memory, the memory occupation of the zigzag trees is better than the memory occupation of sliced right-deep trees. This is because zigzag trees use the next permanent relation to join as a probing relation which does not have to be loaded in memory. Instead, the temporary relation is actually traded with a permanent relation. If we compare the right-deep tree of Figure 1 and the zigzag tree of Figure 2, we see that the memory necessary to execute these plans is the maximum memory requirement of each phase, i.e., of each subtree. If we denote the memory space used to hold a relation and its hash table in memory by the relation name itself, we get the following formula for the memory occupation of the right-deep tree of Figure 1:

**Figure 1. Sliced right-deep tree with n+1 subtrees**



$$max(T_1 + (\sum_{j=2}^{i_1} R_j), T_1 + T_2 + (\sum_{j=i_1+1}^{i_2} R_j), \ldots, T_n + (\sum_{j=i_n+1}^{i_{n+1}} R_j))$$

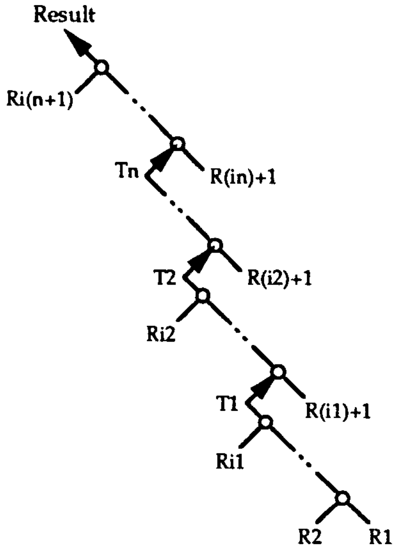
and the following formula for memory occupation of the zigzag tree of Figure 2:

$$max(T_1 + (\sum_{j=2}^{i_1} R_j), T_1 + T_2 + (\sum_{j=i_1+2}^{i_2} R_j), \ldots, T_n + (\sum_{j=i_n+2}^{i_{n+1}} R_j))$$

For each phase but the first one, the right-deep subtree needs to load one more relation than the corresponding zigzag subtree. In cases of queries with many joins or accessing large relations, this memory savings may lead to a zigzag tree with fewer phases than a right-deep tree, and possibly a better response time. This holds if the temporary relations of right-deep trees are not spooled to disk. If they are, less memory is necessary but at the expense of additional I/Os.

The impact of available memory is neglected in Hong and Stonebraker (1991) who formulated the following *buffer size independent hypothesis*: the choice of the best sequential plan is insensitive to the amount of buffer space available, as long as

**Figure 2. Zigzag tree with n+1 subtrees**



the buffer space is above the hash-join threshold. This assumption is combined with the *two phase hypothesis*, which states that the best parallel plan is a parallelization of the best sequential plan. These are heuristic simplifications which make sense in a restricted model (i.e., left-deep trees only and no inter-operation parallelism). However, when pipeline parallelism is allowed, the choice between a left-deep, a right-deep, and a zigzag tree depends on the amount of available memory and should be decided at runtime. We plan to implement such a strategy with choose-plan operators (Graefe and Ward, 1989). However, in our current prototype, an upper bound is statically set to the amount of memory available for a query.

## 2.3 Bushy and Segmented Right-Deep Trees

Bushy trees comprise all tree formats that are not linear, i.e., trees in which at least one node has both operands that are intermediate relations. Using bushy trees is problematic for at least two reasons. First, their search space is much larger and thus might be untractable to explore, especially with a parallel execution model. Consequently, we use randomized search strategies, and Ioannidis and Cha Kang (1991) showed that exploring a search space with bushy trees can be easier than a space of left-deep trees alone, at least for some cost functions. The second problem, as Schneider (1990) pointed out is that, unlike linear trees, bushy trees still allow different slicing strategies once pipeline and blocked operands are specified. To avoid this open problem, we have adopted a default strategy to slice bushy trees into phases: operations are executed in the first possible phase. This provisional simplification avoids further explicit slicing of bushy trees, until their scheduling is better understood.

Chen et al. (1992*a*) proposed interesting heuristic schemes to determine the execution of segmented right-deep trees. Those formats are very similar to our zigzag trees. Schneider (1990) also briefly mentioned a format identical to zigzag trees. However, note that segmented right-deep trees are a bit more general because each right-deep segment can be attached not only at the first join operation of the next segment, but also in the middle of it. If all segments are attached to the first operation of the next segment, segmented-right trees reduce to zigzag trees, otherwise they are bushy trees. Also note that Chen et al. (1992*a*) did not consider the possibility of avoiding writing to disk the intermediate results of each pipeline segment. We think that this decision is important in cases of limited memory to get a minimum number of phases and thus possibly reduce execution time. It is precisely with this combination of techniques that we have run experiments on the DBS3 prototype.

## 3. Data and Execution Models

The data and execution models represent the way the optimizer sees data and execution. They are an abstraction of the actual data representation and low-level execution. At this level, concurrency control and recovery mechanisms are transparent; thus, we consider only intra-query parallelism.

### 3.1 Distributed and Shared-Memory Models

We chose a distributed memory (DM) execution model for shared-memory (SM) because (a) our database system could work on both the shared-memory architecture of the DBS3 prototype and the distributed memory architecture of the EDS machine with minimal effort; (b) code fragmentation, mandatory in DM, reduces data conflicts in SM; (c) cache coherency control in SM shares similarities with message passing in DM; (d) the mapping from DM to SM takes advantage of efficient SM features (fast communication through shared memory, synchronization points via shared variables); and (e) this mapping, when performed at compile time, leaves enough latitude for the execution system to make good runtime decisions, e.g., for dynamic load balancing.

In both DM and SM models, data are horizontally fragmented and operations are cloned into several threads to allow intra-operation parallelism. In a DM system, a *Processing Element* (PE) includes a processor and its memory, while it is a processor in SM. A DM model imposes static links between data fragments, PEs and threads: (1) a data fragment is linked to a PE where it is stored; (2) a thread is linked to a PE which will execute it; and thus, (3) a data fragment is linked to a thread that will consume it. In the SM model, any PE can access any data fragment, because memory is shared, and the system dynamically assigns threads to PEs to favor load balancing. Thus, any thread can access any fragment, thereby creating access conflicts.

With the DM model, the mapping between the tuples of a relation and the PEs where they are stored is called the *physical home* of the relation. The home of a relation does not change at runtime but data can be transferred, which creates intermediate data with a different home. With the SM model, no static information regarding fragmentation need be attached to a relation, because re-fragmentation is supposed to be much less expensive than data transfers in DM. However, if a relation does not hold in main memory, re-fragmentation actually can be quite expensive.

With our DM model, the *logical home* of a relation is the mapping between its tuples and its data fragments. A mapping between the fragments and the PEs will link the logical and physical homes. When implementing our DM model on a SM system, relations are statically fragmented and information regarding such fragmentation is kept in a catalog. The link between data fragments and PEs disappears in our SM system, but data fragments still are linked to threads. Although the execution system can allocate any thread to any PE, a thread can only access some specified data fragments. This method allows us to keep the advantages of automatic load balancing while reducing access conflicts. In the following sections, *home* denotes logical home.

## 3.2 Execution model

The Execution Model is a parallel dataflow execution model that supports local data operators, transfer operators, and control operators. These operators are embedded in a parallel algebraic language, called Parallel LERA (Borla-Salamet et al., 1991), that enables sophisticated combinations to support complex queries (e.g., multiple joins, division, fixpoint, aggregation). A program in this language embeds its own parallel execution control using control operators. An alternative way to coordinate a parallel execution is to rely on a distributed coordinator which is part of the system and not of the program. This coordinator has to be general enough to deal with all situations, and therefore suffers in performance for specific cases where tuned optimizations are possible. A detailed description of how a control code is added to the program in DBS3 can be found in Borla-Salamet (1991).

An execution plan is represented initially as a directed graph of operators. Note that trees are needed to support multi-join queries, but more general queries may be translated into operator graphs. An operator operates on fragmented relations via operator instances. A data fragment is accessed only by one operator instance at a time. Thus, no specific concurrency control is needed within a parallel transaction. Each arc is labeled with the kind of synchronization that exists between the operators, say $Op_1$ and $Op_2$. There are mainly two such synchronizations: pipe and seq. Pipe specifies that two operators are executed in a producer-consumer mode with $Op_1$ producing messages to be consumed by $Op_2$. Seq specifies that $Op_2$ waits until $Op_1$ completes before starting its own execution. Finally, each of these synchronizations may be *local* or *global*. Local means that the synchronization of $Op_1$ and $Op_2$ results in the independent synchronization of each couple of operator instances $< Op_1i, Op_2i >$ and global means that a global synchronization mechanism for all the operator instances is needed.

The control associated with a global pipe execution is ensured by means of control operators. When a producer operator instance completes, a single end-of-stream message is sent to a centralized control operator that waits for the completion of all the active producer operator instances. When this condition becomes true, the centralized control operator broadcasts an end-of-stream message to all active consumer operator instances, indicating that they will not receive any more messages.

The control associated with a global seq execution is very similar to that of global pipe. The difference is that the centralized control operator that detects the completion of the first operator must broadcast a trigger message instead of an end-of-stream message to all the second operator instances. This trigger message will activate all these operators.

Global control operators are similar to the well-known *barrier* in SM architectures. In our implementation, however, all operator instances are not always active (not all data fragments are accessed), and also several operator instances may be supported by a single thread. Our control takes these differences into account and optimizes control for each specific case.

For simplicity, we note a local synchronization with a single arrow (e.g., $Op_1 \overset{pipe}{\rightarrow} Op_2$ and a global synchronization with a double arrow (e.g., $Op_1 \overset{seq}{\Rightarrow} Op_2$. All queries start and end with the following global sequential synchronization: $\overset{start}{\Rightarrow}$ and $\overset{end}{\Rightarrow}$. If two or more operators are running locally, they are glued together by the code generator into a single code fragment. We illustrate the operators that are combined into a single code fragment with an underbrace. Vertical braces denote a parallel execution. We now illustrate the execution model with some examples.

*Selection.* If relation $R$ is declustered in $n$ fragments, the operation *Select*($R$) is equivalent to the union of $n$ operations *Select*($Ri$), with $i = 1,n$, where each individual operation can be done in parallel. However, if the select predicate contains placement attributes, fewer nodes than $n$ (ideally one) need be involved. The associated execution graph is very simple:

$$\overset{start}{\Rightarrow} \underbrace{\underline{Select\ R}} \overset{pipe}{\Rightarrow} \underbrace{\underline{Store\ Res}} \overset{end}{\Rightarrow}$$

If the result has to be stored where it is produced (co-located with the home of R), then the last global pipe becomes a local one, and the two last operators may be glued together in a single code fragment, as follows:

$$\xRightarrow{start} \underbrace{Select\ R \xrightarrow{pipe} Store\ Res} \xRightarrow{end}$$

*Joins.* Parallelizing binary operations is more complex because, for optimal parallelism, each operand relation must be declustered the same way. For example, if $R$ and $S$ both are declustered in $n$ fragments using the same function on the join attribute, the operation $Join(R,S)$ is equivalent to the union of $n$ parallel operations $Join(Ri,Si)$, with $i = 1,n$. We call this an "ideal join, " and its execution graph is:

$$\xRightarrow{start} \underbrace{NestedJoin\ R,S} \xRightarrow{pipe} \underbrace{Store\ Res} \xRightarrow{end}$$

The following join strategies all are based on the hash-join algorithm. The generic operator $Join(A,B)$ means: (1) redistribute (if necessary) $A$ on the join attributes and build, on the fly, a partial index $I_A$ per data fragment, and (2), redistribute $B$ (if necessary) on the join attributes and, in pipeline mode, probe each tuple of $B$ using the index $I_A$. Note that $Join(A,B)$ does not produce the same implementation as $Join(B,A)$.

If the compiler chooses to use such an algorithm for the previous query, the produced execution graph will be:

$$\xRightarrow{start} \underbrace{Scan\,R \xrightarrow{pipe} Build\,I_R \xrightarrow{seq} Scan\,S \xrightarrow{pipe} Probe\,I_R} \xRightarrow{pipe} \underbrace{Store\ Res} \xRightarrow{end}$$

Note that the local algorithm "nested join" in the first example may be implemented as the combination of the local operators of the second example. If the "ideal" condition is not satisfied, parallel join algorithms (Gardarin and Valduriez, 1984; DeWitt and Gerber, 1985) attempt to make such a condition available by reorganizing the relations. A reorganization means dynamically creating a secondary home of a permanent relation. Since this reorganization is performed for a subsequent operation, we create on the fly a partial index per data fragment to accelerate local processing. If $S$ is not declustered on the join attribute, but $R$ is, we obtain what we call an "Assoc-Join." The corresponding graph is:

$$\xRightarrow{start} \underbrace{Scan\,R \xrightarrow{pipe} Build\,I_R} \xRightarrow{seq} \underbrace{Scan\,S} \xRightarrow{pipe} \underbrace{Probe\,I_R} \xRightarrow{pipe} \underbrace{Store\ Res} \xRightarrow{end}$$

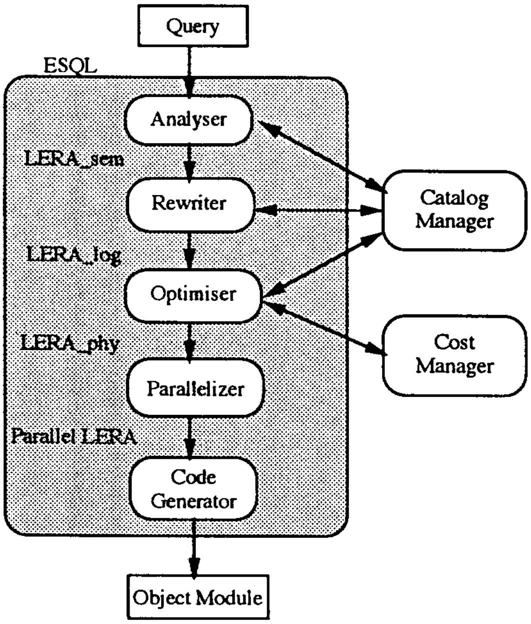
If none of the relations is declustered on the join attribute, and it becomes necessary to reorganize both relations, then this join algorithm is called "hash-join." In this case, each basic operator is implemented as an independent code fragment.

## 4. Query Processing

### 4.1 Overview of the Compiler

The ESQL compiler transforms an ESQL query (Gardarin and Valduriez, 1992) into an object module optimized for execution on the parallel execution system. The compiler proceeds in several subsequent translation phases that progressively add

**Figure 3. Compiler architecture**



lower level details regarding the execution environment. Therefore, the compilation process is divided among semantic analysis, rewriting, optimization, parallelization, and code generation. The purpose of this architecture is to perform compile-time optimization and parallelization while keeping the run-time system simple and hopefully very efficient. Within the compiler (Figure 3), the five layers use different variants of an algebraic language (Language for Extended Relations Algebra, LERA) (Borla-Salamet, 1993) to express their output.

The analyzer performs syntaxic and semantic analysis of the input ESQL query and produces the corresponding algebraic program, i.e., a graph of relational operations. The data definition statements are directly executed by the catalog manager, while the data manipulation statements must go through the remaining layers of the compiler. The rewriter simplifies and transforms the algebraic program to facilitate optimization. For instance it eliminates common sub-expressions and groups select-project-join operations in N-ary nodes. The optimizer takes all final decisions for executing the program in the parallel execution system and integrates them in an annotated algebraic program expressed on physical data. These decisions concern primarily the ordering of operations, the selection of the best method to access each relation and the choice of the best parallel algorithm per operation. The optimizer implements the techniques previously described.

The parallelizer translates an optimized program into a parallel program according to the execution model (Section 3). This translation has two goals. First, it must guarantee the correct execution of the parallel program by adding control operations for synchronizing data operations and detecting terminations. Second, it must yield decentralized scheduling of the programs to increase throughput. Similar to BUBBA (Boral et al., 1990) and Volcano (Graefe, 1989*b*), our approach to parallelization avoids using a centralized scheduler. Furthermore, it allows peep-hole optimization that can yield significant reduction of the number of control messages (Borla-Salamet et al., 1991). Following the optimizer directives, the parallelizer proceeds by expressing the parallel algorithms, deciding the necessary scheduling of the program, and introducing the control operations to implement the corresponding schedule. Thus, the parallelizer generates a self-scheduling parallel program, avoiding the need for a run-time scheduler.

The code generator produces the final executable object module for the parallel execution system. This involves producing a number of object code fragments, each one corresponding to one or more operations. This module can be compiled and linked in the usual way.

## 4.2 Cost model

In this section, we define the cost of a plan and the cost functions for the most important operators that compose a query execution plan, namely *Select* and *Join*. Note that we make an important simplifying assumption, namely that the set of processors assigned to operations does not overlap.

*4.2.1 Cost Model for DM and SM.* In the preceding section, we pointed out the main differences between the execution models of SM and DM and established the principles for mapping DM to SM. In this section, we exploit this analysis to define a cost model. However, we give both the SM and DM cost formulas when necessary, i.e., when the system differences matter. If we ignore concurrency issues, only the cost functions for data reorganization and memory consumption differ. Indeed, reorganizing 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 assigned to operations to execute does not overlap. This will simplify the formula for response time with DM. It is possible, however, that two distinct operations have the same home, a case which the formula takes into account. For example, in our prototype, the catalog specifies that two relations are declustered on the same home. In SM, the execution system is expected to dynamically balance the load among processors.

**Figure 4. Cost Model Parameters**

| | |
|---|---|
| $\mid R_i \mid$ | number of tuples of relation $R_i$ |
| $\| R_i \|$ | size of relation $R_i$ in Kbytes |
| $nodes(R_i)$ | number of nodes (or fragments, for SM) of relation $R_i$ |
| *page_size* | size of a memory page in Kbytes |
| *mess_size* | size of a message in Kbytes |
| *mess_time* | time needed to send one message |
| *com_st_up* | communication start up |
| *build_time* | time to insert a tuple into a hash table |

*4.2.2 Cost Functions.* We define the cost of a plan as having three components: total work, response time, and memory consumption. The first two components 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 (Ganguly et al., 1992) uses a parameter specified by the system administrator, by which the maximum throughput is degraded to decrease response time.

In the following, $R_i$ refers to a base relation of the physical schema, $O$ refers to a relational operator, and *degree*($O$) refers to the degree of parallelism of operator $O$.

We use the following functions:

-
$$isMat(O) = \begin{cases} 1 & \text{if the output of operation } O \text{ is materialized} \\ 0 & \textit{otherwise} \end{cases}$$

- *send_cost*(*data_size*): the cost of sending data of size data_size across the interconnect, i.e., *send_cost*(*data_size*) = *round* (*data_size*/*mess_size*) * *mess_time* + *com_st_up*
The function *round*() rounds up its parameter to the next largest integer.

- *decluster_cost*($R_i$,$n$):
. *DM*: the tuples of relation $R_i$ are declustered on a target home composed of $n$ nodes, with cost:
$decluster\_cost(R_i, n) = n * send\_cost(\| R_i \| \;/\; (nodes(R_i) * n))$
. *SM*: the tuples of relation $R_i$ are reorganized into $n$ fragments with cost:
$decluster\_cost(R_i, n) = build\_time * \mid R_i \mid * log_2(\mid R_i \mid)$

This function computes the response time of the *decluster* operation. We assume that the operations of each home node are perfectly overlapped.

*Cost functions for global algorithms.* The parallelization of simple operations, such as *Select* and *Join*, is based on a global and a local algorithm. The global algorithm specifies the home of the operation,[1] the reorganization and the transfer of input relations if necessary, and the mode of consumption of each operand (pipelined or materialized). The local algorithm specifies the implementation of the operation at each node, such as a join method for *Join*. These decisions are expressed as annotations attached to the operation.

The cost of a global algorithm represents the cost of reorganizing input relations (in DM, it is the cost of transfers to the consumer nodes). The cost of a local algorithm represents the access and processing cost on one node of the operation home. We give only cost functions for some global algorithms; the cost for local algorithms and the remaining global algorithms are given in Zaït (1990). Let us denote $cost(alg, R_i)$ as the cost of global algorithm *alg* to reorganize relation $R_i$, and $cost(local)$ as the cost of the local operation algorithm.

- *Ideal:* tuples of the input relations are distributed on their join attributes and, in DM, share the same home. Thus, no reorganization is needed, i.e.,

$$\forall \; R_i \quad cost(Ideal, R_i) = 0$$

- *Assoc:* one input is declustered on its join attribute and the other must be reorganized (and, in DM, sent to the home of the first one, which is also the operation home). So, the cost of reorganizing $R_i$ is:

$$cost(Assoc, R_i) = \begin{cases} 0 & if\ R_i is\ already\ organized \\ decluster\_cost(R_i, nodes(R_j)) & otherwise \end{cases}$$

- *Hash:* the two inputs are reorganized on their join attributes using the same distribution function (and, in DM, sent to the home of their consumer operation). So, the cost of reorganizing $R_i$ into $n$ buckets (and, in DM, sending it to a home composed of $n$ nodes) is:

$$\forall \; R_i \quad cost(Hash, R_i) = decluster\_cost(R_i, n)$$

1. Referred to as cloning annotation in Ganguly et al., 1992.

*Cost Functions for Operations.* For each operation the optimizer chooses the best global algorithm and the best local algorithm. We give the cost for only two operations of the target language, i.e., selection and join. Let us denote $respTime(O)$, $totWork(O)$, and $memCons(O)$ as the response time, total work, and memory consumption of operation $O$, respectively, and *local* (respectively *global*) the local (respectively global) algorithm of an operation. The computation of total work is the same for all operations:

$$totWork(O) = degree(O) * cost(local) + \sum_{R_i \in input(O)} nodes(R_i) * cost(global, R_i)$$

- $Sel_{selpred}(R_i)$: the cost of performing a selection, using predicate selpred, on tuples of $R_i$, is

$$respTime(Sel_{selpred}(R_i)) = max(cost(local), cost(global, R_i))$$
$$memCons(Sel_{selpred}(R_i)) = nodes(R_i) * page\_size$$

  We assume that the operand $R_i$ is consumed in pipe mode and, thus, one page of each $R_i$ fragment needs to be present in memory. The reorganization of $R_i$, if needed, is performed simultaneously to the selection processing.

- $Join_{pred}(R_i, R_j)$: $R_i$ and $R_j$ are respectively outer and inner operands of the join operation that satisfy predicate pred. The cost of joining the tuples of $R_i$ and $R_j$ is:

$$respTime(Join_{pred}(R_i, R_j)) = cost(global, R_j) + max(cost(local), cost(global, R_i))$$
$$memCons(Join_{pred}(R_i, R_j)) = nodes(R_i) * page\_size + \| R_i \|$$

We assume that, to perform the join operation, the inner operand must be materialized, if it is not a base relation, whereas the outer is consumed in pipeline mode. The reorganization of $R_i$ is performed simultaneously with the join processing, after the reorganization of $R_j$ has completed.

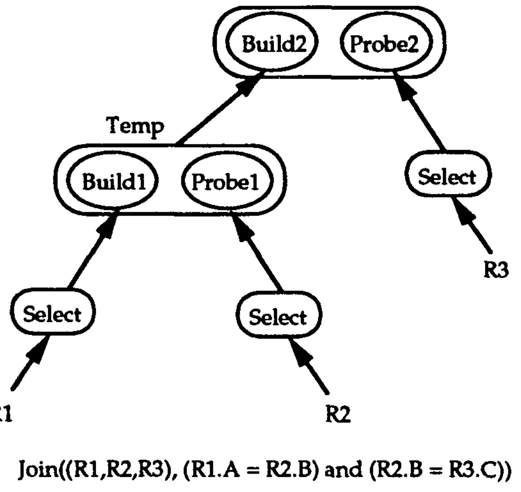
Given a plan rooted at operation $O$, denoted $O(child_0, child_1, \cdots, child_{k-1})$, the cost of a plan p is computed as follows:

$$totWork(p) = totWork(O) + \sum_{i=0}^{k-1} totWork(child_i)$$

$$respTime(p) = max(respTime(N), max_{i=0}^{k-1}(respTime(child_i) * (1 - isMat(child_i))) + max_{i=0}^{k-1}(respTime(child_i) * isMat(child_i))$$

$$memCons(p) = memCons(O) + \sum_{i=0}^{k-1} memCons(child_i) * (1 - isMat(child_i))$$

**Figure 5. Query execution plan**



*4.2.3 Fragmentation and Dependencies Among Optimization Choices.* Some proposals (e.g., Hong and Stonebraker, 1991), assume that a build node always is necessary before each probe node for hash-join algorithms. In our system, if two consecutive joins are on the same attribute, we are able to avoid the build node of the second hash-join, because the intermediate result is already hashed on the proper attribute. This is possible only when the intermediate relation is the building relation in the next join. The second build node can be avoided because the temporary relation, temp, is already hashed on $R_2.B$ (Figure 5). Not only is redistribution not necessary but, due to the way our execution model is implemented, the intermediate relation is already in (the equivalent of) a hash table.

The ability to avoid such nodes is an important aspect of an execution model. First, the use of a temporary relation as a building relation (in left-deep or zigzag trees) is particularly interesting when it is already hashed. But more important is the impact on the cost model: if a subplan avoids a build node it may be better than cheaper subplans that do not, i.e., the principle of optimality (Ganguly et al., 1992) is violated. The way an intermediate result is hashed introduces what we call a *global dependency*, i.e., a dependency between the choice of a subplan and other optimization choices. Other global dependencies are the following: the order of tuples in intermediate results when sorting algorithms are used (Selinger et al., 1979); the site of intermediate relations in distributed systems (Lohman et al., 1985); and resource contentions (Ganguly et al., 1992). In shared-memory systems such as XPRS (Stonebraker et al., 1988), the way a relation is hashed, or more generally fragmented, is much less important than it is in distributed-memory systems because it does not imply data transfers. However, the mere cost of hashing may not be negligible and the fact that processors usually access shared-memory through caches may raise problems similar to those of distributed-memory architectures.

### 4.3 Search Strategies

Given an input query, the optimizer search strategy explores the space of possible execution plans that implement the input query, and seeks one that minimizes a cost function. If the search space is too large, however, it would be prohibitive to explore it exhaustively. Thus, the strategy must balance optimization effort with the quality of the resulting execution plan. As the solution space gets larger when parallel execution of a query is considered, the choice of a search strategy becomes an important issue. Several search strategies have been proposed for query optimization to deal with different query types (simple vs. complex) and with different requirements (ad-hoc vs. repetitive). Enumerative strategies consider many points in the solution space, but try to reduce the solution space by applying heuristics (Selinger et al., 1979). This approach can lead to the best possible solution, but faces a combinatorial explosion when the search space is composed of parallel plans. To investigate large search spaces, randomized strategies have been proposed that try to improve a start solution until obtaining a local optimum, or until meeting some other stopping condition (Swami, 1989).

Our optimizer search strategies (Lanzelotte and Valduriez, 1991) are variations of several known strategies: dynamic programming, simulated-annealing, and iterative-improvement. Thus, instead of statically cutting off the search space because of its size, we implemented several search strategies to cope with the different sizes of search spaces, and let the optimizer choose the appropriate one, according to the query type (ad-hoc vs repetitive and simple vs. complex). We definitely think that compiling heuristics, as proposed in Chen et al. (1992*a*, *b*) is essential. However, the fanout of parallel architectures and execution models is quite large and it is difficult to find general heuristics. This problem is especially true with our approach because our optimizer is targeted for both shared-memory and distributed-memory architectures. Thus, we chose to rely on randomized strategies as long as general heuristics have not emerged.

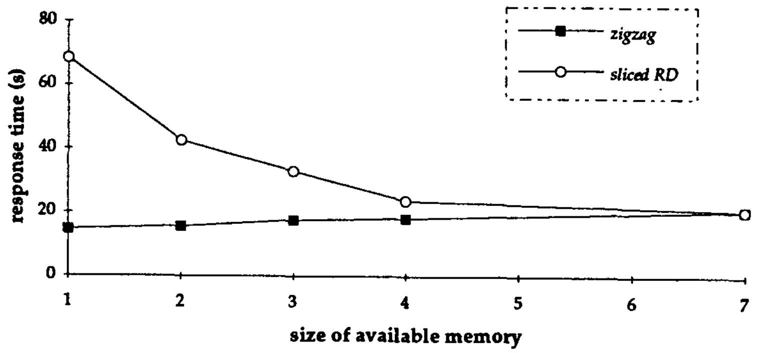
## 5. Performance Measurements

In this section, we show the advantages of zigzag trees over right-deep trees under various conditions through experiments on the DBS3 prototype. Our results, however, should be considered as preliminary due to the limitations of our hardware, especially the fact that we used only one disk.

The target machine for the testbed was the multiprocessor Encore MULTIMAX 520. This machine was configured with 10 NS32532 processors (8.5 Mips, each having 256 KB cache memory), 96 MB main memory and 1 GB disk storage. Processors, memory and I/O boards were interconnected by a 100 MB/s bus.

We used a database composed of eighteen relations, generated automatically following the specifications of the standard Wisconsin Benchmark (Bitton et al., 1983). Tuples were 208 bytes long, and relation cardinality was fixed to 10,000 tuples

Figure 6. Sliced-RD trees vs. zigzag trees for varying sizes of available memory

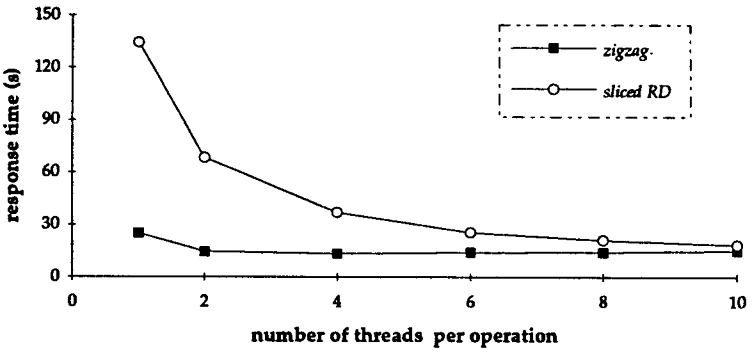


(except for the probing relation of the first join, which had 1000 tuples). The size of the intermediate relation of each join was the size of smaller operand relation, due to the join predicate. Thus, the size of intermediate relations is determined by the size of the first probing relation. Relations were declustered by hashing on their first attribute and the number of fragments was 30. However, in the next to last series of experiments (Figure 9) we let the number of fragments vary. Also, in the last series of experiments (Figure 10) we let the cardinality of the first relation vary; consequently we had a cardinality of intermediate results. Several of our experiments were made with more tuples on a previous version of our prototype and led to similar behavior. With the new version of prototype, we decided it was more interesting to experiment with different values of the parameters than using bigger relations.

We conducted our experiments using join queries with eight to eighteen relations. We chose a chain form for the query, i.e, each relation but the first and the last is connected to exactly two other relations by one join predicate. This allows comparison of execution plans even if they are of different formats (zigzag vs. sliced right-deep), because the optimizer always produced the same join ordering. In these experiments we were interested only in the response time component of the cost function.

Right-deep trees can lead to a better response time than left-deep trees under different assumptions (Schneider and DeWitt, 1990). If relations are fully declustered and the disks are not fully utilized, right-deep trees have a better response time because scans (as well as build nodes) can be done in parallel. We expected a similar advantage for zigzag trees over sliced right-deep trees because zigzag trees allow the execution to be done in fewer phases. However, because our prototype has

**Figure 7. Sliced-RD trees vs. zigzag trees for varying numbers of threads/operation**

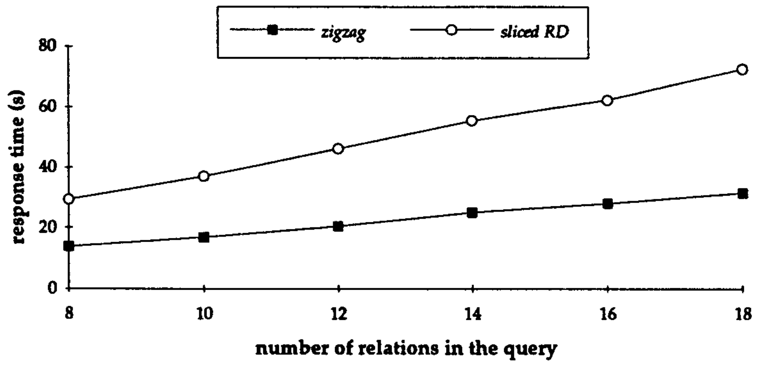


only one disk at the moment, we have concentrated on different assumptions. The other case in which right-deep trees yield a faster response time is when relations are partially fragmented without overlap in a DM system. In such a case, right-deep trees yield an almost constant response time when the number of joins in the query increases, as long as enough memory is available. The reason why the probe nodes can also be done in parallel is that the physical homes of the operations do not overlap, allowing pipeline parallelism to occur without time sharing.

In a SM system, inter-operation parallelism is mainly useful to help balance CPU-bound and I/O-bound operations (Hong, 1992). However, that work does not take into account the case in which the number of processors is high compared to the size of the relations to join. This can happen when selectivities of select operations are very high. In such a case, the overhead of intra-operation parallelism limits the number of processors that should be used for each join. On the other hand, pipeline and independent parallelism should be included in the optimization search space.

To reproduce conditions in which pipelining can be effective on a shared-memory prototype with ten processors, we artificially restricted intra-operation parallelism by reducing the number of threads of each operation. In five series of experiments we took into account five parameters: the size of available main memory, the number of relations in the query, the number of threads per operation, the number of fragments per relation, and the size of intermediate results. For each series of experiments, one of the parameters varied on the x axis while the others were fixed. The values for the fixed parameters were the following: the memory can hold one hashed base relation (plus the intermediate result), thus the number of joins is seven. To avoid time sharing, the number of threads per operation was set to two for the first and the fifth experiment and to five for the third and fourth ones.

**Figure 8. Sliced-RD trees vs. zigzag trees for varying numbers of query relations**



In each series of experiments we saw how a zigzag tree can take advantage of its smaller number of phases to yield a better response time than a sliced right-deep tree (sliced-RD tree). Figure 6 illustrates the case in which the x axis was the number of base relations that can be held in memory (with their hash table). To be more precise, a number N on this axis means that N base relations *and the intermediate relation*, can be held in main memory. We see that the sliced-RD tree often needs one more phase than the zigzag tree, and thus more time to complete. In Figure 7, the number of threads varies from one to ten. With such limited memory the sliced-RD can execute only one join at a time while the zigzag tree can execute two joins in parallel. As a result the zigzag tree yields a better response time, but not, however, one that is exactly half the time of the sliced RD. The reason is that, when the number of threads exceeds 5, time sharing occurs during the execution of the zigzag tree, and the phases of each tree are not similar. The first phase of the zigzag tree contains only one join operation. In Figure 8, the number of relations in the query varies from eight to eighteen. The response time of both curves increases linearly because of the memory limitations: for each additional two joins, two more phases are necessary with a sliced-RD and one for the zigzag tree.

In the fourth experiment (Figure 9), response time decreased when the number of fragments increased, because the local algorithm used to join the fragments was nestloop. When the number of fragments increases, they get smaller and the benefit of hashing is higher. In our execution model, however, a pipeline queue was associated with each fragment, which introduced some overhead. Zigzag trees seem to take less advantage of an increasing number of fragments, probably because they have a higher degree of pipeline. In the last experiment (Figure 10), the size of the first probing relation varied from 1000 to 20,000. However, since the size of each intermediate relation was the size of the smaller join operand, the size of

Figure 9. Sliced-RD trees vs. zigzag trees for varying numbers of fragments/relation

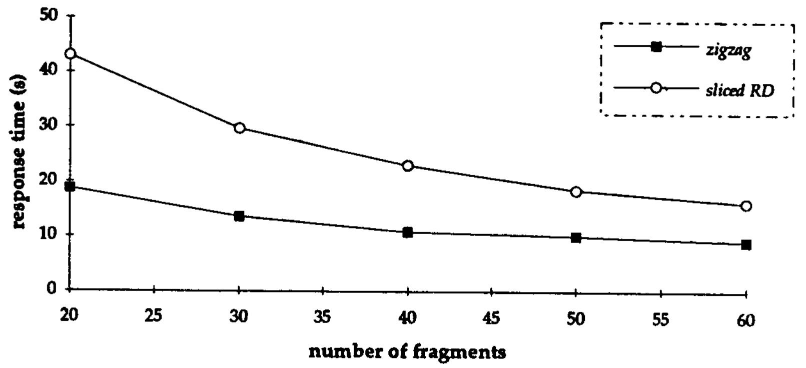
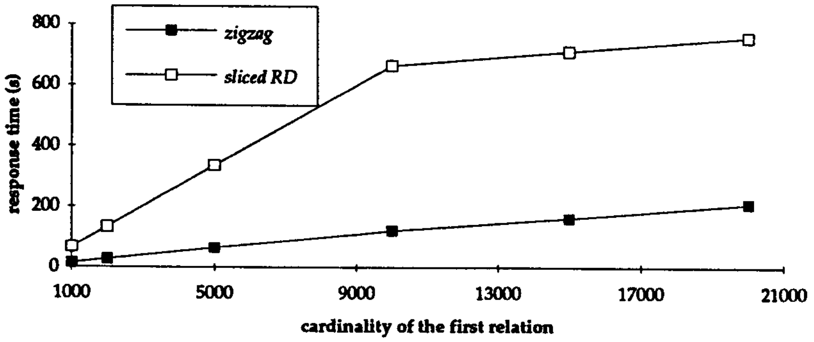


Figure 10. Sliced-RD trees vs. zigzag trees for varying first relation cardinality



intermediate results actually varied from 1000 to 10,000. We had decided that the available memory could hold one relation and the intermediate result, therefore it could have been possible that the optimizer chose a different plan according to the size of the intermediate results. However, for simplicity, we assumed that the available memory could hold just one base relation and one intermediate relation but not two base relations. Thus, we could keep the same plan, and the size of the intermediate result should actually be understood as being just less than 10,000. Of course, such a situation is very favorable to zigzag trees and should be considered as a limit case. While it is not easy to see (Figure 10), the ratio of the execution time of the right-deep tree to the time of the zigzag tree does not change very much up to 10,000 tuples. After that point, the size of the intermediate results does not change any more, and only the first join takes more time. However, this additional time is proportionally bigger for the zigzag tree.

In summary, when pipeline parallelism occurs without time-sharing, zigzag trees can yield a better response time than sliced right-deep trees. This advantage, obtained on our SM prototype, is probably even higher on a DM machine with several disks and partial declustering.

## 6. Conclusion

We have described our approach to the compile-time optimization and parallelization of queries for execution in DBS3, a shared-memory parallel database system. Our approach enables exploring a search space large enough to include zigzag trees which are intermediate between left-deep and right-deep trees. The problem of efficiently searching a large search space is solved by using randomized search strategies (Lanzelotte and Valduriez, 1991).

Unlike the XPRS approach (Hong and Stonebraker, 1991), which essentially reduces the optimization search space, our approach is more general. Because DBS3 implements a parallel dataflow execution model, this approach applies to both shared-memory and distributed-memory architectures. Furthermore, it avoids the need for a centralized scheduler and enables compile-time optimization of dataflow control.

Performance measurements run using the DBS3 prototype have shown that, in cases of limited memory and when temporary results are not spooled to disk, zigzag trees can yield a better response time than sliced right-deep trees. Our results, however, should be considered preliminary due to the limitations of our hardware. Similar experiments should be run on a distributed memory system for which zigzag trees are expected to be even more advantageous.

## Acknowledgements

This work was partially funded by the Esprit projects, EDS and IDEA. The authors are grateful to Patrick Valduriez for his constant supervision and detailed comments on this work.

## References

Bergsten, B., Couprie, M., and Valduriez, P. Prototyping DBS3, a shared-memory parallel database system. *Proceedings of the International Conference on Parallel and Distributed Information Systems,* Miami Beach, Florida, 1991.

Bitton, D., DeWitt, D.J., and Turbyfill, C. Benchmarking database systems—A systematic approach. *Proceedings of the International Conference on Very Large Databases,* Florence, Italy, 1983.

Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., and Valduriez, P. Prototyping BUBBA, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering,* 2(1):4-24, 1990.

Borla-Salamet, P., Chachaty, C., and Dageville, B. Compiling control into queries for parallel execution management. *Proceedings of the International Conference on Parallel and Distributed Information Systems,* Miami Beach, Florida, 1991.

Chen, M.-S., Lo, M., Yu, P.S., and Young, H.C. Using segmented right-deep trees for the execution of pipelined hash joins. *Proceedings of the International Conference on Very Large Databases,* Vancouver, British Columbia, 1992*a*.

Chen, M.-S., Yu, P.S., and Wu, K.-L. Scheduling and processor allocation for parallel execution of multi-join queries. *Proceedings of the International Conference on Data Engineering,* Tempe, Arizona, 1992*b*.

DeWitt, D.J. and Gerber, R. Multiprocessor hash-based join algorithms. *Proceedings of the International Conference on Very Large Databases,* Stockholm, Sweden, 1985.

DeWitt, D.J. and Gray, J. Parallel database systems. The future of database processing or a passing fad? *ACM SIGMOD Record,* 19(4):104-112, 1990.

Ganguly, S., Hasan, W., and Krishnamurty, R. Query optimization for parallel execution. *Proceedings of the ACM SIGMOD,* San Diego, California, 1992.

Gardarin, G. and Valduriez, P. ESQL2: An extended SQL2 with f-logic semantics. *Proceedings of the International Conference on Data Engineering,* Tempe, Arizona, 1992.

Gardarin, G. and Valduriez, P. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems,* 9(1):133-161, 1984.

Graefe, G. Volcano: An extensible and parallel dataflow query processing system. Computer Science Technical Report, Oregon Graduate Center, Beaverton, Oregon, June, 1989.

Graefe, G. Encapsulation of parallelism in the Volcano query processing system. *Proceedings of the ACM SIGMOD,* Atlantic City, New Jersey, 1990.

Graefe, G. and Ward, K. Dynamic query evaluation plans. *Proceedings of the ACM SIGMOD,* Portland, Oregon, 1989.

Hong, W. Exploiting inter-operation parallelism in XPRS. *Proceedings of the ACM SIGMOD,* San Diego, California, 1992.

Hong, W. and Stonebraker, M. Optimization of parallelism query execution plans in XPRS. *Proceedings of the International Conference on Parallel and Distributed Information Systems,* Miami Beach, Florida, 1991.

Ioannidis, Y. and Cha Kang, Y. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. *Proceedings of the ACM SIGMOD,* Denver, Colorado, 1991.

Lanzelotte, R.S.G. and Valduriez, P. Extending the search strategy in a query optimizer. *Proceedings of the International Conference on Very Large Databases,* Barcelona, Spain, 1991.

Lanzelotte, R.S.G., Valduriez, P., and Zaït, M. Optimization of object-oriented recursive queries using cost-controlled strategies. *Proceedings of the ACM SIGMOD,* San Diego, California, 1992.

Lohman, G., Mohan, C., Haas, L., Daniels, D., Lindsay, B., Selinger, P., Wilms, P. Query processing in R*. In: Kim, W., Reiner, D.S., and Batory, D.S., eds. *Query Processing in Database Systems.* Berlin: Springer-Verlag, 1985, pp. 31-47.

Stonebraker, M., Katz, R., Patterson, D., and Ousterhout, J. The design of xprs. *Proceedings of the International Conference on Very Large Databases,* Los Angeles, California, 1988.

Schneider, D.A. Technical report #965. Ph.D. thesis, University of Wisconsin, 1990.

Schneider, D.A. and DeWitt, D.J. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *Proceedings of the ACM SIGMOD,* Portland, Oregon, 1989.

Schneider, D.A. and DeWitt, D.J. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. *Proceedings of the International Conference on VLDB,* Brisbane, Australia, 1990.

Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., and Price, T.G. Access path selection in a relational database management system. *Proceedings of the ACM SIGMOD,* Boston, Massachusetts, 1979.

Swami, A. Optimization of large join queries: Combining heuristics and combinational techniques. *Proceedings of the ACM SIGMOD,* Portland, Oregon, 1989.

Wilschut, A.N. and Apers, P.M.G. Dataflow query execution in a parallel main-memory environment. *Proceedings of the International Conference on Parallel and Distributed Information Systems,* Miami Beach, Florida, 1991.

Zaït, M. Access method selection in a parallel database system (in French). Master's thesis, Université Paris 6, 1990.

Ziane, M., Zaït, M., and Borla-Salamet, P. Parallel query processing in DBS3. *Proceedings of the International Conference on Parallel and Distributed Information Systems,* San Diego, California, 1993.