Dynamic Scheduling of Complex Distributed Queries

Luc Bouganim, Françoise Fabret, C. Mohan, Patrick Valduriez

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Dynamic Scheduling of Complex Distributed Queries

Luc Bouganim - Françoise Fabret - C. Mohan
Patrick Valduriez

N° 3677
Avril 1999

THÈME 3

Dynamic Scheduling of Complex Distributed Queries

Luc Bouganim[*,**] **Françoise Fabret**[**] **C. Mohan**[**,***] **Patrick Valduriez**[**]

* PRISM Laboratory,
78035 Versailles,
France
Luc.Bouganim@prism.uvsq.fr

** INRIA Rocquencourt,
France
Francoise.Fabret@inria.fr
Patrick.Valduriez@inria.fr

IBM Almaden Research,
USA
mohan@almaden.ibm.com

Abstract: *Database queries over distributed data sources have become increasingly complex. Execution plans produced by traditional query optimizers for such queries may yield poor performance for several reasons: the cost estimates may be inaccurate; the memory available at run-time may be insufficient; and remote data may not be rapidly or predictably accessible. In this paper, we address the memory limitation and data accessibility problems. We propose to dynamically schedule complex distributed queries in order to gracefully adapt to the available memory and to deal with irregular data delivery. Our approach is both proactive by the careful step-by-step scheduling of several query fragments and reactive by the processing of these fragments based on data arrivals. We describe a performance evaluation that shows important performance gains (up to 70%) in several configurations.*

Keywords: *Query Execution, Scheduling, Data Accessibility, Memory Consumption*

Résumé: *Les requêtes effectuées sur des sources de données distribuées deviennent de plus en plus complexes. Les plans d'exécution produit par les optimiseurs de requêtes traditionnels peuvent être peu performants pour plusieurs raisons : Les estimations de coûts peuvent être inexactes; la mémoire disponible lors de l'exécution peut s'avérer insuffisante; et les données distantes peuvent ne pas être disponibles immédiatement, lorsqu'elles sont demandées. Dans cet article, nous nous intéressons aux problèmes de limitation mémoire et de disponibilité des données. Nous proposons de changer dynamiquement l'ordre d'exécution des fragments de requête complexes afin de s'adapter à la mémoire disponible et aux arrivées irrégulières des données distantes. Notre approche est à la fois prévisionnelle, en produisant un ordonnancement pas à pas de plusieurs fragments de requêtes, et réactionnelle en exécutant ces fragments en fonction de l'arrivée des données distantes. Nous décrivons une évaluation de performances montrant des gains importants (jusqu'à 70 %) dans plusieurs configurations.*

Mots clés : *Exécution de requêtes, Ordonnancement, Disponibilité de données, Consommation mémoire*

1 Introduction

Classical query processing is based on a well-known distinction between compile-time and runtime. The query, written in a declarative language (e.g., SQL), is optimized at compile time, thus resulting in a complete query execution plan. At runtime, the query engine executes the query, following strictly the decisions of the query optimizer.

However, database applications have become increasingly complex in terms of architecture (e.g., distributed), query complexity and data volumes. Typical examples are decision support applications (e.g., TPC-D) executing in a distributed environment or queries for bulk loading data warehouses that access several operational sites. With such complex database applications, the execution of a query plan as produced by the optimizer could result in poor performance for three main reasons:

- **Accuracy of estimates:** Sizes of intermediate results used to assess the costs of query execution plans can be inaccurate even when detailed database statistics are used [GI97, KD98]. Typically, poor performance is the consequence of a sub-optimal execution plan, especially for complex queries involving large databases.
- **Memory limitation:** The amount of available memory at runtime for processing a query may be much less than what was assumed at compile time. Executing the query "as is" might cause thrashing of the system because of *paging* [BKV98, CG96, ND98]. Poor performance thus results from an uncontrolled use of memory.
- **Data accessibility:** With respect to remote data access, it is possible that data needed during execution may be temporarily inaccessible [AFT98, AFTU96, UFA98]. This may happen because (i) the remote data is the result of a complex sub-query; (ii) the data is produced by a remote site which is overloaded and/or is not powerful enough; (iii) the network is slow or not reliable. In these cases, the query engine may be stalled, waiting for some data which is not yet accessible, leading to a dramatic increase in response time.

In this paper, we address the performance problem due to memory limitation and data accessibility issues. Inaccurate estimates may require run-time re-optimization [KD98], while memory limitation and data accessibility problems can be addressed using adaptive execution techniques (e.g., memory reallocation [BKV98, ND98], dynamic scheduling [AFTU96, ONK+97], and adaptive relational operators [IFF+99]). Run-time re-optimization and adaptive execution are, in fact, complementary and may be used together in order to provide good performance [IFF+99]. The problem of inaccurate estimates has been widely addressed by others. Hence, in this paper, we do not focus on estimation errors. We assume that an efficient execution plan has been produced at compile-time and that it may have been readjusted at run-time, just before query execution starts.

Now, consider the execution of a query which is complex in terms of the number of operators and data volumes, and which accesses remote data. Our goal is to develop an execution strategy that can dynamically adapt to limited memory and data unavailability situations for a given query execution plan that is assumed to be efficient. A classical solution to the data accessibility problem is to reschedule the query plan (e.g., changing the originally-decided order of execution of the query operators) when the query engine is stalled. However, this can increase memory consumption and thus interfere with decisions regarding memory allocation. This is the reason behind our decision of addressing these two problems jointly.

The strategy that we propose in this paper differs significantly from previous ones. In the following, we briefly compare our work to the most related approach called *query scrambling* [AFT98, AFTU96, UFA98]. We provide further discussion and comparison with other related work in Section 7.

The basic strategy of query scrambling is to dynamically modify the query execution plan in reaction to unexpected delays in data access. [AFTU96] defines three types of delays. (1) *Initial delay:* when a delay occurs for the first tuple only. (2) *Bursty arrival:* when data arrives in bursts followed by long periods of no

arrival. (3) *Slow delivery:* when the arrival rate is regular but slower than normal. Depending on the type of delay, several strategies have been proposed. To handle initial delay, the authors propose, in a first phase, to reschedule the query plan. If the latter is not sufficient, a second phase creates a new execution plan using heuristics [AFTU96] or a query optimizer [UFA98]. In [AFT98], bursty arrival is considered and dynamic scheduling is used to hide short repetitive delays. The authors have not provided any solution to the problem of slow delivery.

The different scrambling techniques are all based on the same concept: react to a significant event (e.g., a timeout occurring while waiting for remote data to arrive). Thus, scrambling is a *reactive* approach. Reactive methods can fail for two reasons. First, the event condition that triggers the reaction, e.g., an appropriate timeout value, may be difficult to choose and/or tune. Second, the event may be detected too late to enable a timely reaction. In contrast, *proactive* approaches (e.g., [CG94]) try to predict the behavior of query execution and plan ahead (possibly at start-up time of a query execution) for likely contingencies. Proactive methods, in turn, may fail in their prediction because of inaccuracy. In our approach, we interleave proactive phases, where we plan for the near future in order to avoid inaccuracy, and reactive phases in the spirit of [IFF+99].

In our approach, the query scheduler (QS) and the query processor (QP) are responsible for planning and executing, respectively. The QS has all the knowledge about the execution of the current query and it uses that information to control the QP. When the query execution begins, the QS chooses a set of independent query fragments which can be processed concurrently, and orders the fragments using a *priority concept*. To compute the priority of a given fragment, the QS uses heuristics based on its knowledge of the query execution plan, the arrival rate of the data, the memory context and the I/O overheads generated. These heuristics allow the QS to avoid scheduling fragments that may incur more overheads than the gains realized, and to determine the order of processing the fragments that might be scheduled. Selected fragments are sent to the QP in order to be processed concurrently. A fragment with a certain priority is considered for processing a batch of data only if none of the higher priority fragments has any data to process (i.e., those fragments are temporarily blocked because of inaccessibility of data). Thus, the QP as a whole is blocked only if there is no accessible data for all the fragments that are scheduled concurrently. When a fragment ends, the QS produces a new ordered list of fragments and continues until the query ends.

Thus, our query scheduler acts proactively by choosing and ordering a set of fragments. Our query processor acts reactively by processing all these fragments concurrently using their priorities. The triggering event, here, is the fact that no data is available for one fragment. However, no timeout mechanism is necessary. In fact, the QP can switch from one fragment to another with negligible overhead and without negative consequences as the QS ensures that these fragments can be processed concurrently.

By using this strategy, we can address both the data accessibility and memory limitation problems. Moreover, this solution applies to any of the three previously defined types of data accessibility problems. As our approach is independent of any timeout mechanism, it is able to hide repetitive short delays, which makes it particularly suited to slow delivery cases, e.g., when the remote sites are overloaded.

The remainder of the paper is organized as follows. Section 2 states the problem more precisely. Section 3 describes the architecture of our reactive-proactive query execution engine. Section 4 presents the heuristics used by the query scheduler in order to dynamically optimize the query execution. In Section 5, we discuss certain choice of heuristics and present implementation considerations. Section 6 presents the results of a performance evaluation which highlight the value of our contributions. Section 7 discusses related work. Finally, we summarize and provide some concluding remarks in Section 8.

2 Problem Formulation

In this section, we state explicitly our assumptions regarding the execution system and define more precisely the model of query processing. These help in making a clear statement of the problem we are addressing.

2.1 Context

We consider the execution of a query which is complex in terms of the number of operators and data volumes, and which accesses remote data. The remote data may be located on a high-speed local area network (LAN) or a wide area network (WAN). However, since we consider large queries, it is more likely that the data will be on a LAN.

Figure 1(a) and 1(b) show a possible architecture and a simple query example. We consider the execution of a query on a uniprocessor node (e.g., node 4 in Figure 1). Some of the data being processed (e.g., A, B and C) is the result of sub-queries processed on other nodes (e.g., nodes 1, 2 and 3 in Figure 1). These nodes are considered as black boxes. We are concerned with the response time for the execution of the query when it is executed using a certain amount of shared resources (e.g., CPU, disk, memory).

2.2 Communication Protocol

When the query engine needs some data from a remote site, it sends a sub-query and creates a queue of a given size in order to buffer the received tuples. The *communication manager* (see Figure 1(c)) is responsible for receiving tuples from several remote nodes and transferring them to the appropriate queues. If the relevant destination queue is full, sub-query processing at the remote node is temporarily stopped as it cannot send more tuples, until tuples are consumed from that queue. Thus, the communication protocol used is a kind of "window protocol" [MPTW94]. The technique proposed in this paper is restricted to the case where the data delivery rate of each remote node is independent of the number of nodes triggered in parallel. Thus, the network must not be the bottleneck of the system. Moreover, we assume that the query engine has a way to know the approximate data delivery rate of each node of the system or at least a relative one. These assumptions are quite realistic in modern distributed environments.

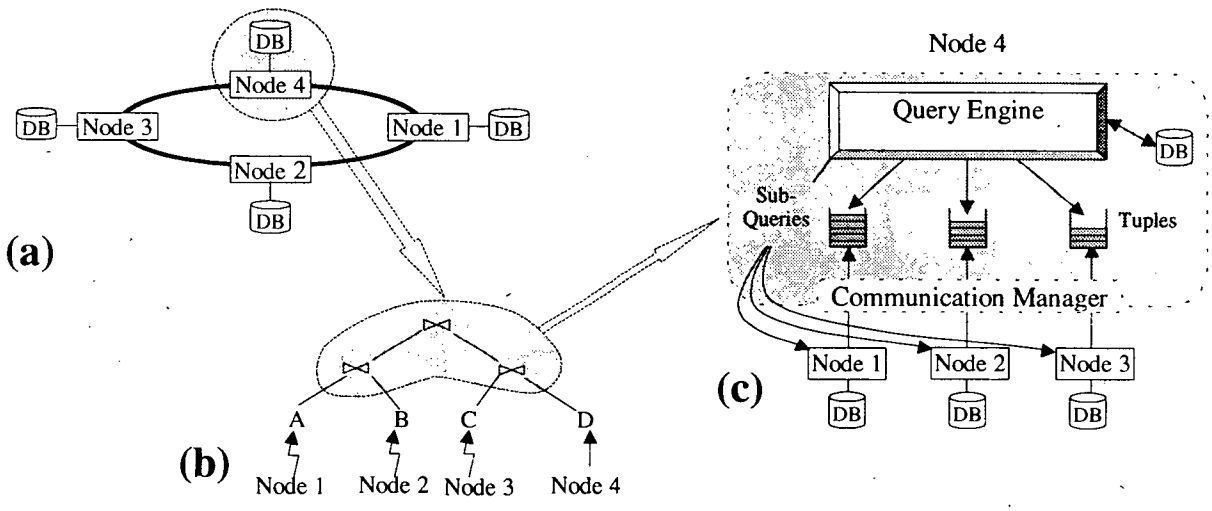


Figure 1: *(a) DBMS Architecture (b) a Query Example (c) Communication protocol*

2.3 Query Execution Plans

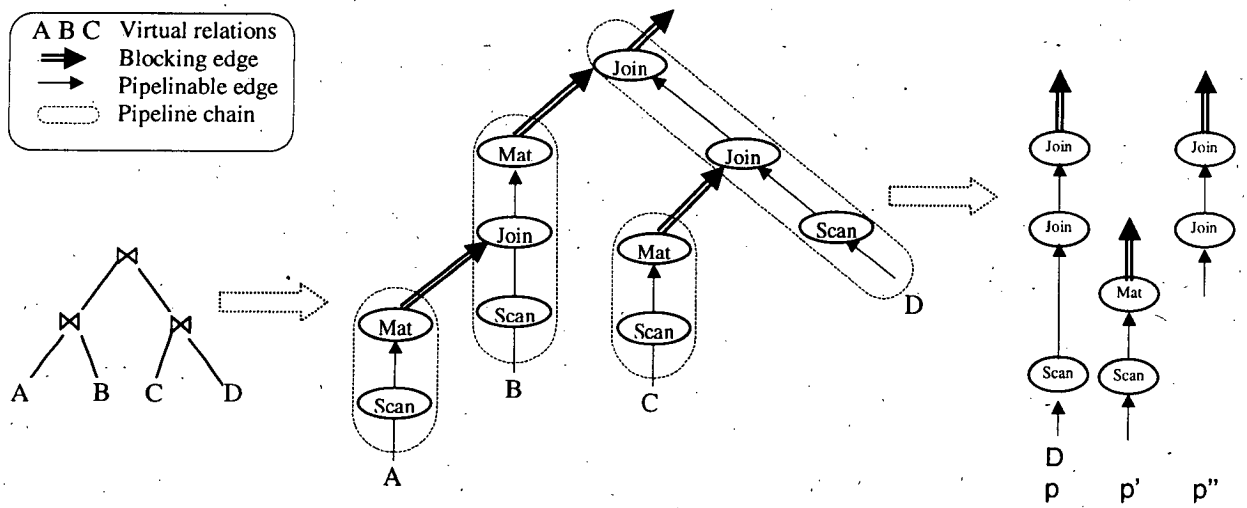
Query processing is classically done in two steps. The query optimizer first generates an "optimal" *query execution plan* (denoted by *QEP*) for a query. The *QEP* is then executed by the query engine which implements an *execution model* and uses a library of relational operators [Gra93]. The optimizer can consider different shapes of *QEP*: left deep, right deep, segmented right deep or bushy. Bushy plans are the most general and the most appealing because they offer the best opportunities to minimize the size of intermediate results [SYT93]. Hence, we consider bushy trees in this paper.

A *QEP* is represented as an *operator tree* and results from the "macro-expansion" of the join tree [HM94]. Nodes represent atomic physical operators and edges represent dataflow. Two kinds of edges are distinguished: blocking and pipelinable. A *blocking edge* indicates that the data is entirely produced before it can be consumed. Thus, an operator with a blocking input must wait for the entire operand to be materialized before it can start. A *pipelinable* edge indicates that data can be consumed "one-tuple-at-a-time". Therefore, the consumer can start as soon as one input tuple has been produced.

We consider classical query execution plans with binary, asymmetric relational operators (e.g., hash-join) that have one blocking input and one pipelinable input, and produce a pipelinable output. The *QEP* also contains unary operators, e.g., a scan. Finally, the unary operator *mat* will be introduced in the query plan before each blocking edge to indicate that materialization must occur at that point. Notice that such a materialization can occur in memory or on disk depending on the available resources.

2.4 Definitions and Notations

We introduce several definitions and notations that we illustrate with the examples related to the QEP of Figure 2.



(a) Logical execution plan

(b) Physical execution plan

(c) a PC p, C-fragment p', C-complement p''

Figure 2: *A QEP example*

Dependencies: A *pipeline chain* (PC for short) is the maximal set of physical operators linked by pipelinable edges. Blocking edges induce *dependency constraints* between PC's. Given two PC's p_1 and p_2, we say that p_1 *blocks* p_2 iff there exist two operators op_1 and op_2 such that op_1 is in p_1, op_2 is in p_2 and there is a blocking edge directly connecting op_1 and op_2. The ancestors of a PC p, noted *ancestors(p)*, are all the PC's that block p. The

ancestor relation may be extended to the *ancestors** relation in the classical way (transitive closure). A PC p is *C-schedulable* if p has no dependency constraints anymore, i.e., the execution of every PC in ancestors(p) has terminated. In Figure 2, ancestors(p_D) = { p_B, p_C} and ancestors*(p_D) = {p_A, p_B, p_C}.

Memory consumption: A physical operator can operate in a range of memory allocation situations between its minimum and maximum requirements with dissimilar performance results [YC93]. Let's define *mem(op)* to be the *maximum* memory requirement of a given operator *op*. *Mem(op)* does not include any memory for the result tuples[1]. For instance, if *op* is a hash-join operator, *mem(op)* is equal to the size of the hash table built using the left operand. Given a PC p such that $p = \{op_1, \ldots, op_n\}$, p is *M-schedulable* if the sum of mem(op_i) is less than the total amount of memory available for the query usage, which we assume will not change during its execution. In Figure 2, mem(p_B) is the size of the hash table for the input relation A (assuming hash-join).

Schedulability and C-fragmentation: A PC p is *schedulable* if it is C-schedulable and M-schedulable.

PC's can be further fragmented in order to enable more scheduling possibilities. If a PC cannot be scheduled because of memory or dependency constraints, a subset of the PC may be scheduled. Given a PC p that is not schedulable, we associate to p a pair of PC's p' and p''. The role of p' is to retrieve tuples from the input relation of p, apply the first scan operator of p (if any) and materialize the result in a temporary relation that will become the input of p'', which corresponds to the remaining operators of p. In what follows, we call p' *C-fragment(p)*, and p'' C-*complement*(p). The rationale for this decomposition is that p' has no ancestors and consumes a negligible amount of memory, and hence is always schedulable.

2.5 Motivating Example

To illustrate the potential gain brought by addressing the data accessibility problem, we now present a simple example which consists of the sub-tree (A join B) of the QEP presented in Figure 2. Suppose that relation A contains 10 K tuples and its delivery cost (delay) is 50 µs per tuple, and B contains 20 K tuples and its delivery cost is 60 µs per tuple. Processing the join using hashing will incur a relatively small cost (≈ 6 µs per tuple) compared to the data delivery rate. Thus, the response time is dominated by the data delivery time. A completely sequential execution will give a response time of 1.7 seconds (10 µs x 50 K + 20 µs x 60 K = 1.7s).

Now consider the decomposition of the QEP for A join B into PC's. A better execution strategy than the traditional iterator strategy is to decompose p_B into C-fragment(p_B) and C-complement(p_B) in order to schedule concurrently C-fragment(p_B) (which retrieves remote tuples and stores them locally) and p_A (which retrieves remote tuples and builds the hash table). When p_A finishes after 500ms, C-fragment(p_B) is stopped. p_B is then scheduled concurrently with C-complement(p_B) which consumes result tuples produced by C-fragment(p_B). This second phase of the execution is dominated by the retrieval time of the rest of p_B, approximately 700ms. Thus, the total elapsed time with this scheduling is 1.2 seconds, leading to a gain of 30% by overlapping delays between A and B.

However, scheduling a C-fragment induces I/O overhead due to the partial materialization of the retrieved tuples. Consequently, such a strategy may be employed only when this overhead is compensated by the gain obtained with an overlapped execution.

2.6 Problem Statement

We can now describe the problem we are addressing as follows. We have several producers (the remote sites) and one consumer (the query execution site). Each producer may have a different and variable delivery rate because of the complexity of the sub-query it is evaluating or because the producer's site is overloaded. A

[1] The amount of memory needed by the input and output buffers is neglected to simplify the presentation.

sequential execution, i.e., consuming entirely the data produced by one producer before accessing another one, leads to a response time with a lower bound equal to the sum of the time needed to retrieve the data produced by each producer. Thus, if the time taken for data retrieval by some producer is larger than the processing time of the data produced by this producer, then the query engine will stall.

We claim that there exists an execution strategy that interleaves the processing of data from several producers in order to keep the query engine busy doing some useful work, thus reducing the total response time.

Our objective is therefore to reduce the impact of data accessibility problems by scheduling several PC's concurrently. To decrease the risk of stalling the query engine, we may schedule as many PC's as possible. Hence, the C-fragmentation of a PC p into C-fragment(p) and C-complement(p) allows for more flexibility. However, this may cause two problems. First, scheduling C-fragment(p) can incur a high overhead since we break the pipeline chain, forcing materialization of an intermediate result. Second, scheduling too many PC's can result in disastrous performance because it can cause contention at several levels in the system.

The problem is, then, to select, schedule and execute concurrently several PC's (i.e., initial PC's and/or their decompositions) of a given query execution plan in order to minimize the response time. A crucial point is that the data delivery rate is typically unpredictable and may vary during the query processing. Thus, any algorithm that will fix ahead a given scheduling for the duration of the entire query execution will be unsatisfactory.

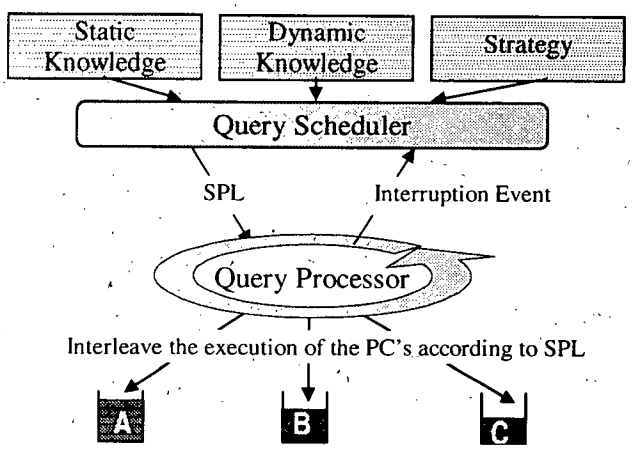
3 Dynamic Query Processing

This section presents the architecture of our query engine, the mode of operation of the query processor and the query scheduler requirements.

3.1 Query Engine Architecture

Our dynamic query processing strategy is both proactive and reactive. The query execution is divided into several steps in order to adapt to the dynamic variability of the data delivery rate. Each step contains a proactive and a reactive phase. Figure 3 presents the general architecture of our query engine.

The *query scheduler* (QS) implements the proactive component of the system. It uses *static knowledge* (e.g., the QEP), and *dynamic knowledge* (e.g., the *current state* of each scheduled PC) to compute a *scheduling priority list* (SPL), i.e., an ordering of scheduled PC's. The SPL is sent to the *query processor (QP)*. The QP processes concurrently the PC's of the SPL, reacting to predefined *interruption events*, thus implementing the reactive component of our execution strategy. Figure 3(b) shows how the QP and the QS cooperate.



Query Scheduler algorithm:

- *On receive interruption from QP*
  1. *Compute SPL*
  2. *Send SPL to QP*

Query Processor algorithm:

- *On receive SPL from QS*
  1. *Execute SPL until an interruption event arises*
  2. *Send interruption event to QS*

Figure 3: *(a) Query Engine Architecture and (b) Query Execution Algorithm*

3.2 A Reactive Query Processor

At each execution step, the QP takes as input the *scheduling priority list SPL* = {p1, ..., pn} produced by the QS. The task of the QP is to interleave the execution of these PC's in order to maximize the processor utilization with respect to the priorities defined in SPL. To do so, the query processor scans the queue[2] associated with the PC which has the highest priority. If no tuple is available, then the QP scans the second queue in the list and so on. When the QP finds tuples, it processes a certain amount of tuples called a *batch*. After each batch processing, the QP returns to the highest priority queue. Thus, the QP always processes the first PC which has some available batches, following the priorities indicated by the SPL. The rationale behind considering batches of tuples rather than individual ones is to reduce the potential overheads due to frequent switches between scheduled PC's. Thus, the query processor is a reactive process, reacting to variations in data accessibility. However, the reaction of the QP is immediate (i.e., it considers another queue) and there is no timeout protocol.

Moreover, our reactive query processor can react to two other events: *EndOfPC* and *RateChange* which may change the scheduling decisions. The former signals that some PC of the SPL has terminated, and the latter, that a significant change has occurred in the data delivery rate of some PC of the QEP. This information is delivered by the *communication manager* which monitors the data delivery rate of each queue. The algorithm of the query processor is shown below:

1. *Loop*
2. *Process a batch of tuples following priorities given by SPL*
3. *If a PC p has terminated then return (EndOf PC(p))*
4. *If the data delivery rate of PC p changes significantly then return (RateChange (p))*
5. *End Loop*

3.3 Query Scheduler Requirements

Considering our query processor algorithm, the QS must produce an SPL which contains a sufficient number of PC's in order to prevent the QP from stalling. However, scheduling too many PC's can cause thrashing because the interleaved execution of concurrent PC's can lead to paging [BKV98].

As the QS computes repetitively the SPL, at each step of the execution, there is a clear tradeoff between the gain brought by an "optimal" schedule and the time to find such a schedule. So, the challenge is to produce a reasonable schedule in a time interval that is short compared to the average processing time of one execution step.

4 Query Scheduler Strategy

Suppose that we want to produce an optimal execution schedule. In the following, we identify the different parameters that may impact this schedule and derive the complexity of the problem in a simplified case.

Obviously, the ***data delivery rates*** and the ***query plan*** (i.e., the query plan shape, the dependency constraints, the PC processing time, etc.) are crucial factors. Less evident parameters are: the ***intermediate result sizes*** which may impact the overhead induced by the scheduling of a C-fragmented PC and the ***available memory*** at run-time that impacts the number of PC's which can be scheduled concurrently. Finally, a more subtle parameter is the ***memory allocation point*** (i.e., the amount of memory assigned to an operator) [BKV98, YC93] of each operator of the scheduled PC's. In fact, giving more memory to an operator can speed up its

[2] Note that some PCs can have local inputs. In this case, the query processor will read tuples from the local disk. Conceptually, the local disk can be considered to be a *stored queue*.

processing, reducing the amount of I/Os. However, it can prevent other PC's from being scheduled, leading to more I/Os. This tradeoff is discussed in the following.

Assuming a simplified case, where all these parameters are known or decided at the beginning of the execution, and do not change during the execution, it can be shown[3] that finding the best schedule for such a case has a complexity of $O(n!)^n$ where n is the number of input relations of the query at the execution site. Note that this complexity does not take into account the choice of the optimal memory allocation point of each operator, considering this parameter leads to a search space of not finitely countable choices.

Thus, the problem is too complex to search exhaustively for an optimal solution, even when not taking into account the fact that the data delivery rate is typically unpredictable.

In the rest of this section, we describe the strategy we use to produce efficiently at each proactive step a reasonably "good" SPL based on heuristics. By "good" SPL, we mean an SPL which is as close as possible to the "ideal" SPL: the one which "*keeps the query processor busy doing only useful work*". We first present some cost functions to quantify the data accessibility problem. We then present the strategy we use to compute the SPL, based on the previous considerations.

4.1 Heuristic Cost Functions

Notations (n, w, c): Let p be a PC, and R the input relation of p. We associate to p three parameters noted respectively n_p, w_p, and c_p. These parameters represent respectively the number of tuples of R, the average time interval between the arrival of R tuples (called the *waiting time* of p), and the average processing time of one tuple of R.

Critical degree: *Let p* be a PC. The *critical degree* of p is given by the formula: $critical(p) = n_p \times (w_p - c_p)$. The PC p is said to be *critical* if critical(p) is greater than zero. Note that in a distributed environment, it is likely that any PC of the QEP, consuming remote data, will be critical as network times are generally predominant.

The critical degree represents the total CPU idle time when p is computed with no other concurrent PC's. Intuitively, a PC p with a high critical degree, can induce important performance loss if scheduled at the end of the query plan. Therefore, p has to be scheduled as soon as possible in order to have some other PC's to schedule concurrently. Nevertheless, it may happen that p is not schedulable because of dependency constraints or memory limitations. In order to avoid a late scheduling of p, an alternative solution is to apply a C-fragmentation to p and schedule C-fragment(p). However, such a strategy leads to the materialization of an originally-pipelined intermediate result, thus incurring I/O overheads. The potential gain of such an approach is given by the *benefit fragmentation ratio (bfr)* defined as follows:

Benefit fragmentation ratio: Let p be a PC, (p', p'') its C-fragmentation, $IO_{p'}$ the I/O cost for reading or writing a tuple produced by p', c' the average processing time associated to p', and σ the selectivity factor of the scan operator of p' if any (default value is 1). The benefit fragmentation ratio of p is given by the following formula:

$$bfr(p) = \frac{Economy\ achieved\ by\ scheduling\ p'}{Cost\ incured\ by\ scheduling\ p'} = \frac{w + c' - \sigma\ IO_{p'}}{c' + \sigma\ IO_{p'}}$$

Indeed, assuming that p' has been scheduled and that n_0 tuples of p' have been processed, then, the cost incurred by the execution of p' is $(n_0\, c' + \sigma\, n_0\, IO_{p'})$ for the scan processing and the result writing[4]. The

[3] Due to space limitations, we have not explained here this complexity computation.
[4] As p' is scheduled concurrently with others PCs, we do not include waiting times in the denominator of *bfr*

potential economy achieved is $n_0(w + c')$ as n_0 tuples of p have been processed. For these tuples, we will not have to pay neither waiting times (i.e., w), nor processing time (i.e., c'). In contrast, we must deduct from this value the cost of reading σn_0 tuples from the local disk which is incurred when executing p''.

Benefit thresholds: Obviously a negative *bfr* means that C-fragmentation should not be considered because there is no potential economy to be realized, i.e., scheduling a C-fragment may increase the response time (and the total work). High *bfr* means that the overhead induced by scheduling the C-fragment *is* negligible, compared to the response time improvement. Between these two extremes, the *bfr* is a continuous function which is an indicator of the importance of performing C-fragmentation.

Consequently, we define two constant threshold values of *bfr* named bfr_{min} and bfr_{ben} which are inputs for our query scheduler. bfr_{min} is the minimum value of *bfr(p)* (p is a given PC) below which C-fragment(p) should not be scheduled, while bfr_{ben} is the value of *bfr* beyond which scheduling C-fragment(p) is clearly beneficial (i.e., scheduling C-fragment(p) increases the total work while at the same time decreasing the response time significantly). Finally, a *bfr* between these two thresholds means that C-fragment(p) should be scheduled as long as it does not delay the execution of others PC's which are potentially more beneficial.

4.2 Strategy for Computing a Scheduling Priority List

The QS computes a SPL at each proactive step by using static and dynamic knowledge. The former consists of the QEP and some heuristic rules, while the latter consists of the current state of the query execution. The QS first computes the set of schedulable PC's (given the current execution state), then it establishes a priority order between these PC's and finally it uses this order, and memory considerations to extract an SPL from the set of schedulable PCs. We will now detail each phase, and, for each one of them, point out the heuristics used and the rationale for these heuristics with respect to our objectives of minimal QEP stalling and maximal usefulness of the work.

Phase 1: Schedulable set computation: During this phase, the QS derives a set, named *Sset,* of schedulable PC's by selecting the PC's that are currently schedulable, and also by fragmenting non-schedulable PCs. The heuristics used to decide which PC's ought to be fragmented are given by the following fragmentation rules:

Fragmentation rules: Let *p* be a PC.

F1: if *p* is not M-schedulable, then apply M-fragmentation (defined below) to p

F2: if *p* is critical, p is not C-schedulable, $bfr(p) > bfr_{min}$, then C-fragment p

Rule F1 applies as soon as the current state of the query execution lets the QS conclude[5] that p will never become M-schedulable, i.e. the memory needed by p is greater than the total amount of memory available for the query execution. In this case, following the heuristics devised in [BKV98, YC93], it is generally more efficient to break p than to execute it "as is". Breaking p changes the memory allocation point of the operators of p. Thus, we apply these heuristics and proceed with an M-fragmentation of p. M-fragmentation consists of modifying the QEP by replacing p by two fragments. This involves inserting a *mat* operator at the highest possible point in p, taking into account the memory requirements of the new fragments. A remarkable feature is that one of the created fragments is necessarily schedulable.

By creating schedulable fragments using M- or C-fragmentation, rules F1 and F2 allow more scheduling flexibility, tending to keep the QP busier doing useful work. With respect to the latter, executing an M-

[5] Such a decision may be taken based on the actual sizes of the operands which have been retrieved combined with estimates for the other operands.

fragment of p leads to some memory being freed, while the usefulness of the execution of the C-fragment of p depends on the value of bfr(p).

Phase 2: Priority computation: During this phase, the QS computes a total ordering of the elements of *SSet* by applying sequentially to *SSet* the following priority rules:

Priority rules: Let p_1 and p_2 be two schedulable PCs.

Pr1: if there exists some PC p such that p_1 is the C-fragment of p, $bfr(p) < bfr_{ben}$, p_2 is in ancestor*(p), then p_2 has priority over p_1

Pr2: if there exists some PC *p* that is not schedulable, and not C-fragmentable according to rule F2, if p_1 is in ancestor*(p), p_2 is not in ancestor*(p), and p is more critical than p_2, then p_1 has priority over p_2

Pr3: if rules Pr1 and Pr2 do not provide a priority order between p_1 and p_2, and if p_1 is more critical than p_2 then p_1 has priority over p_2

Pr4: if there is no priority order between p_1 and p_2 w.r.t. rules Pr1, Pr2, Pr3, then let c_1 (resp. c_2) be the processing time of p_1 (resp. p_2). If $Mem(p_1)/p_1 > Mem(p_2)/p_2$, then p_1 has priority over p_2.

For instance, consider the example of Figure 2. Suppose that P_B and P_C are both critical, P_A is not critical and P_B is not C-fragmentable because its benefit fragmentation ratio is too low. Then if P_B is more critical than P_C, rule Pr2 says that P_A has priority over P_C. On the other hand, if P_C is more critical than P_B, P_C has priority over P_A by rule Pr3. Suppose now that the benefit ratio of P_B is high, and P_B is more critical than P_C, then the C-fragment of P_B has priority over P_C, and P_C has priority over P_A.

Rule Pr1 tends to keep the QP busy: by giving higher priorities to the ancestors of a non-fragmentable critical PC, we can unblock more rapidly this PC, so we have more opportunities for the concurrent execution of the critical PC. The remaining rules tend to maximize the usefulness of the work.

Phase 3: Scheduling priority list computation: During this phase, the QS computes the SPL that is a subset of the ordered set produced in the previous phase. To do that, it considers memory usage with respect to the priority order established in phase 2. These heuristics are described by the following rules:

Order preserving: Let p_1, and p_2 be two elements of SPL. If p_1 has priority over p_2 then p_1 precedes p_2 in SPL

Memory limitation rule: The available memory is sufficient to execute concurrently the PC's of SPL, and SPL is *maximal*. More precisely, let p_1, ..., p_n be the sequence of PC's in SPL, $S = Mem(p_1) + ... + Mem(p_n)$), and M the available memory. Then (1) S is at most equal to M, and (2) for any PC p of SSet, either p is in SPL or it is not in SPL because S + Mem(p) is greater than M.

Stability rule: Let p be a PC occurring in the SPL computed during a certain proactive step, say S_i. Suppose that Mem(p) is not null. Then if p is schedulable at the proactive step S_{i+1}, the SLP computed at S_{i+1} must contain p.

The last two rules tend to maximize the usefulness of the work by avoiding paging.

5 Discussion

In this section, we discuss the rationale behind our approach to decomposing a sequence of operations and present some important implementation considerations. We also briefly outline the impact of considering executions of multiple queries. Due to space limitations, we are unable to cover these points in great detail.

5.1 Considering More Pipeline Chain Decomposition

In Section 2.4, we have defined a possible decomposition by C-fragmentation of a given PC p into p' and p''. For simplicity, we restricted ourselves to that decomposition which has no impact on memory consumption. However, other decompositions may bring important gains by reducing significantly the I/O overhead induced by the scheduling of a sub-chain of p. Handling several possible decompositions of a PC (i.e., potentially, after each binary operator) does not complicate too much the query engine. However, it may have some adverse effects. For a given PC $p = \{op_1, \ldots, op_n\}$, consider a decomposition $p^i = \{op_1, \ldots, op_i\}$ which consumes some memory. One of the operands needed by p to be schedulable (i.e., for $op_{i+1}, \ldots, op_n$) may be blocked by another PC p_2. $p2$ cannot be scheduled because there is insufficient memory to schedule concurrently p^i and p_2. Thus, p^i blocks p_2 because of memory, while p_2 blocks p because of dependency constraints. With such a configuration, p_2 will be scheduled only when p^i will have terminated, thus p^i has to terminate and will incur potentially a high I/O overhead. We have developed a strategy which allows scheduling only those PC decompositions which cannot conflict with the ancestors of the main PC because of memory considerations.

5.2 Multi-query Execution

Our strategy can reduce significantly the response time of a query (see next section), at the expense of a potential increase of total work. As soon as we consider a multi-query execution context, we must face the classical tradeoff between throughput and response time. A solution is to play with the value of bfr_{min} and bfr_{ben} in order to schedule only those PC's with a sufficiently high bfr. We can establish values for these thresholds by considering the load of the system, i.e., the more the system is loaded, the higher the values for those thresholds. If both thresholds are set to infinity, C-fragmentation will never occur and hence no overhead will be caused. Nevertheless, some gain is still possible by interleaving independent PC's of bushy trees (like p_A and p_C of Figure 2).

Anyway, our approach can be beneficial even if it increases the total work as, by reducing the response time, it also reduces resource contention (specifically, memory and transactional locks [Moh92]) and thus may increase throughput [YC93].

5.3 Implementation Considerations

In the remaining of this section, we briefly discuss implementation considerations relating to process/thread architecture and adaptation of a traditional query engine to support our approach to query processing.

5.3.1 Concurrent Execution and Thread Architecture

Notice that all of the query engine logic except for that of the communication manager is implemented in a single process. The messages exchanged between the QS and the QP are in fact procedure calls and hence are not expensive. The communication manager is asynchronous and is implemented in one or more processes.

5.3.2 Necessary Modifications to an Existing Query Engine

Modifying an existing query engine to handle dynamic scheduling can be a difficult task. Here, we show how this can be done on the commonly accepted iterator execution model [Gra93]. In the iterator model, each operator of the QEP is implemented as an iterator supporting three different calls: *open, next* and *close*. The iterator model is demand-driven, while the execution model assumed in this paper is data-driven. To adapt our execution strategy to the iterator model, we can create a thread for each scheduled PC and use thread priorities to produce a behavior similar to the one described in the previous section. Thus, our query scheduler remains unchanged, and controls the execution using thread priorities and system signals. While this incurs more overhead than the approach proposed here, it requires much less modifications to an existing query engine.

6 Performance Evaluation

Evaluating the performance of several execution strategies is a difficult task, requiring sound understanding of the influence of the parameters of (1) the experimentation platform (e.g., CPU speed, configuration), (2) the benchmark (queries, relations size, shape of the QEP), and (3) even the execution strategy itself (e.g. threshold values). The typical solution is to use simulation which eases the generation of queries and data, and allows testing with various configurations. In contrast, using implementation and benchmarking would restrict the number of queries and make data generation very hard.

We used a performance evaluation methodology similar to [BFV96]. We fully implemented our dynamic execution model and the compared strategies, and simulated the execution of the operators. With this approach, query execution does not depend on relation content and it can be simply studied by generating queries and setting relation parameters (cardinality and selectivity).

In the rest of this section, we describe our prototype and report on performance results focusing only on the data accessibility problem, i.e., assuming the existence of sufficient memory. We chose to present the results for a single, relatively simple, *QEP* in order to explain the behavior of the different strategies, and to analyze the behavior of our query scheduler.

In the first experiment, we slow down only one relation's arrival rate and measure the response time under three different query processing strategies. In the second experiment, we analyze the influence of changes in the minimum possible delay between the arrival of two tuples of a relation. The last experiment studies the performance gain of our approach when the number of slowed-down relations is varied.

6.1 Experimentation Platform

We first describe the QEP used in the experiments. Then, we present the strategies implemented in our prototype and the simulation parameters. Finally, we present the methodology applied in the experiments.

6.1.1 The Query Execution Plan

We chose a fairly simple query: a five-way join, with 4 medium size (i.e., 100K-200K tuples) input relations and 2 small ones (i.e., 10K-20K tuples). The query was generated using the algorithm of [SYT93] and optimized in a Volcano-style dynamic programming query optimizer [GM93]. The resulting QEP is shown in Figure 4. Other queries were tested in the same manner in order to check the validity of the results presented in the rest of this section.

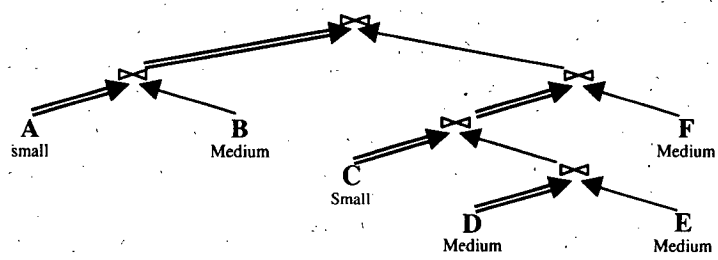


Figure 4: *QEP used for the experiments*

6.1.2 Experimental Prototype and Execution Strategies

We have implemented the classical iterator model, resulting in a sequential execution, denoted by *SEQ*, along with two other strategies *DSE* and *MA*. The sequential execution performance results are easy to predict

analytically. We use its performance as the baseline, i.e., the performance results when nothing is done to handle data accessibility.

We denoted our strategy *DSE,* Dynamic Scheduling Execution, and based the implementation on the architecture and the heuristics described in Sections 3 and 4. The different heuristics rules were implemented in an efficient algorithm which recursively computes the PCs' priorities, beginning with the most critical PC.

The last strategy is the fairly simple *Materialize All*, denoted by *MA,* proposed in [AFT98] which proceeds in two phases. In the first phase, *MA* materializes simultaneously on the disk of the query site all the remote relations. Then, in the second phase, it executes the query with local data stored on disk. Therefore, MA can overlap the delays of several input relations, however at a high I/O overhead.

Notice that we do not compare with the other policies presented in [AFT98]. In fact, the context and the problems considered are different. [AFT98] considers distributed queries on a WAN with potentially high delays between the arrival of two tuples (on the order of magnitude of a second), while we consider LAN architectures with very small delays (i.e., in all the experiments, the average delay between two tuples is at most 0.5 ms).

Finally, we also compute analytically a *lower bound* for the response time, denoted by *LWB.* For a given query Q, the lower bound for the response time is: $LWB(Q) = \max\left(\sum_{p \in Q} n_p c_p \ , \ \max_{p \in Q}(n_p w_p)\right)$

No execution strategy can obtain an execution time lower than LWB. Hence, LWB can be used as a conservative value for the optimal execution strategy's response time. Note however that LWB is generally not obtainable. We include the LWB curves in the figures to get an idea of the quality of our strategy.

The different execution strategies share the simulated operator library, simulated buffer management system and I/O system. Since the different strategies use the same lower-level code, the performance difference can only stem from the execution strategies.We used classical parameters [YC93] for the simulation. They are presented below. The prototype is written in C and runs on a Sun Ultra 1.

Parameter	Value
CPU Speed	100 Mips
Disk Latency	17 ms
Disk Seek Time	5 ms
Disk Transfer Rate	6 MB/s
I/O Cache Size	8 pages
Perform an I/O	3000 Instr.
Number of Local Disks	1

Parameter	Value
Tuple Size	40 bytes
Page Size	8 Kb
Move a Tuple	100 Inst.
Search for Match in Hash Table	100 Inst.
Produce a Result Tuple	50 Inst.
Network Bandwith	100 Mb/s
Send/Receive a Message	20000 Instr.

6.1.3 Experimentation Methodology

For all the experiments, we induce an average interval between the arrival of tuples of a given relation R, denoted $w_{avg}(R)$ by computing for each tuple delivered a w uniformly distributed in the interval $[0, 2w_{avg}(R)]$. This value represents the per tuple average remote processing time, plus the averaged time to send the tuple to the query site. We define the basic w_{avg} value denoted $basic\_w_{avg}$ as the minimum possible value of w_{avg}.

For the experiment, we fix the $basic\_w_{avg}$ as following: $basic\_w_{avg} = IO_{rem} + Send$ where IO_{rem} is the time to read a tuple from the disk on the remote site. In fact, the remote processing time to produce a tuple is generally more than IO_{rem} (e.g., if there is a remote selection). Nevertheless, in the experiment, a PC p can be *critical* even if we use a $basic\_w_{avg}$ for its input relation (depending on the processing time of p).

In the following experiments, we vary the w_{avg} value for one or more input relations. To simplify the presentation, we say in the following that we *slow down* these relations. Moreover, we denote by p_R the PC which takes as input the relation R. We fix the threshold values bfr_{min} and bfr_{ben} to respectively 0 and 1 as the experiment has been done considering a single query execution.

Given the random nature of the w_{avg} distribution, we repeat each measurement 3 times and compute an averaged value of the 3 response times.

6.2 Performance Comparison with one Slowed-down Relation

For this first experiment, we slow down only one relation and measure the response time of the three strategies. We perform this experiment slowing down successively each base relation of the QEP to observe the influence of the *position* of the slowed-down critical relation in the QEP. Figure 5 shows the performance results with A and F being the slowed-down relation. The X axis represents the total time taken to retrieve entirely the slowed-down relation.

As one can expect, SEQ strategy's response time increases linearly with the slowdown because the query processor stalls when tuples from the slowed-down relation are delayed. For SEQ, the performance difference between the two graphics stems from the overlap between processing time and waiting time.

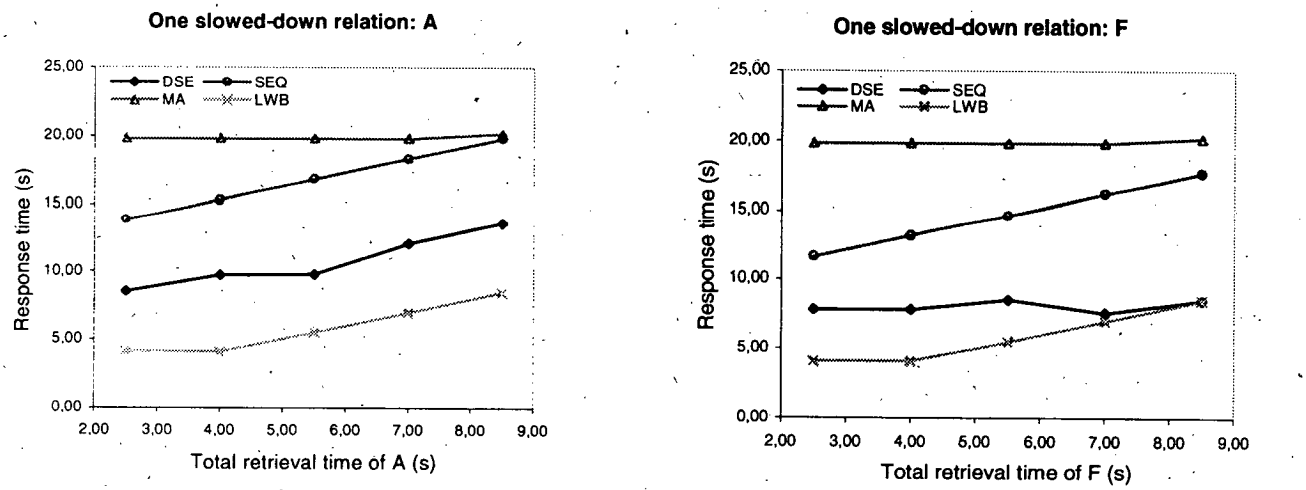


Figure 5: *One Slowed-down Relation Experiments*

MA's response time is always worse in these experiments and stays constant with a slight increase after 8s slowdown. The reason for such bad performance is that only one relation is slowed-down, and hence MA cannot overlap any delays. It should be noted that after 8s (not shown in the graphics), MA increases linearly with the slowdown, as the slowed-down relation becomes the bottleneck of the execution.

We can first remark that DSE achieves better performance improvement with F than with A, specifically when the slowdown is high because while p_A is not terminated, we cannot schedule p_B and p_F, which represent approximately one half of the query execution. This problems does not happen with p_F, which does not block any other PC.

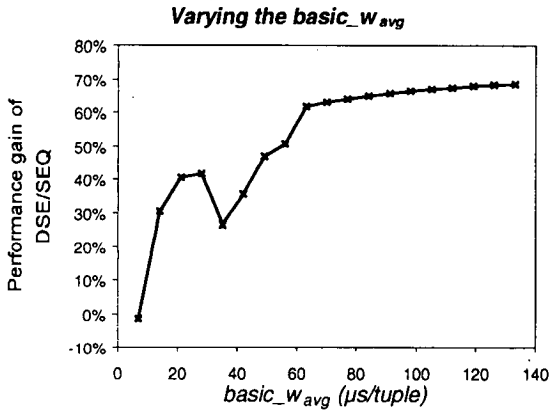
DSE shows better performance than MAT and SEQ, which is not surprising as it reacts to data inaccessibility by partially materializing A (or F) while executing other PC's in the background. One can be surprised by the important performance gain brought by DSE (around 40 %!) even when w_{avg} = $basic\_w_{avg}$. In fact, $basic\_w_{avg}$ is higher than the time to write a tuple on the local disk (i.e., remote accesses are costlier than local ones,

which is generally the case in the distributed environment). This important result shows the potential gains of our approach even when no specific problem occurs in the system. In essence, this gain is the same as the one that can be achieved with asynchronous I/Os in a uniprocessor machine with several disks [MPTW94].

6.3 Influence of the Basic_w_{avg} value

In this experiment, we want to analyze the influence of the *basic_w_{avg}* value. In the previous experiment, we used an underestimated value of *basic_w_{avg}* in order to isolate the effect of the slowed-down relation. Figure 6(a) shows the performance gain of DSE over SEQ with respect to the *basic_w_{avg}* value.

Basically, the performance gain increases with the *basic_w_{avg}* value and goes up to 70%. However, we observe an irregularity when *basic_w_{avg}* is around 35µs. Checking the execution traces, we have observed that this "bad" value is due to the heuristic nature of our QS, i.e., we obtain a better response time with *basic_w_{avg}* = 63µs than with 35µs because the QS has computed a wrong SPL for the latter case.



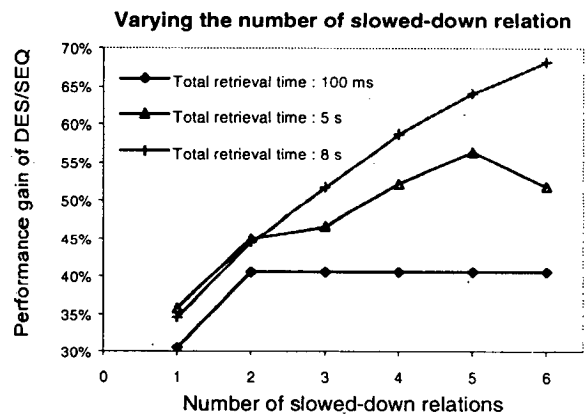


Figure 6: *(a) Varying the basic_w_{avg} value* *(b) Varying the number of slowed-down relation*

6.4 Performance Comparison with Several Slowed-down Relations

This last experiment shows the performance gain of DSE over SEQ with respect to the number of slowed-down relations. We have successively measured the performance by slowing-down 1 relation (A), 2 relations (A and B), and so on. The number of slowed-down relations is on the x-axis (Figure 6(b)). Each curve corresponds to a different total retrieval time for the slowed-down relation.

We observe that the performance gain increases with both the number of slowed-down relations and the total retrieval time of each relation.

6.5 Discussion

The lessons from these experiments are the following: (i) there is potentially an important gain even with a rather small query (e.g., 4 medium-size relations) and small slowdowns (e.g., around 20µs per tuple); (ii) the performance gain increases with the average slowdown of the relations up to a certain limit (e.g., 70%).

One could be surprised by the bad performance of MA, compared to those shown in [AFT98]. This stems from the extent of slowdown. In fact, the delays considered here are very small compared to the I/O overhead generated by MA (15s are necessary to materialize concurrently all the relations of the QEP). Moreover, MA can be beneficial only if there are delays to overlap, i.e., it does not attempt to interleave materialization and normal processing.

7 Comparison with Related Work

Three strategies described in the database literature relate to our work. (1) Re-optimizing at run-time or at start-up time, (2) changing some physical operators without changing the query plan, and (3) dynamically changing the query scheduling.

Re-optimizing the query during its execution can incur high overheads but can result in significant saving, principally if the join order is sub-optimal [KD98]. Thus, mid-query re-optimization should be invoked when estimates are highly inaccurate. Re-optimization may also be invoked at query start-up time if the available resources (specifically memory) differ greatly from what was assumed during query planning. However, taking into account resource availability in the optimizer is a hard optimization problem [GI97] and can make traditional optimization strategies like dynamic programming [HFLP89] impractical. Finally, we need to have somewhat reasonably accurate predictions on data accessibility if we want to take into account this factor during query optimization, but such predictions are rarely possible.

Changing some operators in the query plan is a more conservative approach, which allows adapting the resource needs to what is available [IFF+99, BKV98]. For instance, a hash join [KTM83] that was supposed to execute in one pass can be degraded to a GRACE [KNT89] or hybrid hash join [DKO+84], if available memory is insufficient. In the context of the Tukwila project, [IFF+99] proposes the use of specific non-blocking operators (e.g., the double-pipelined hash join [WFA95]) in order to hide data accessibility problems. As these operators consume more memory, they dynamically revert to blocking operators if sufficient memory is unavailable. The focus of Tukwila is on data integration, where volumes of data transferred across nodes are not likely to be very high, whereas we are concerned with classical distributed databases where such volumes might be high. In addition, the double-pipelined hash join can be used only with equi-joins.

Three related pieces of work that consider the memory allocation problem are [BKV98], [ND98] and [YC93]. In a multi-query environment, [YC93] defines the concept of *return on consumption* to study the overall reduction in response times due to additional memory allocation. The objective is to find a near-optimal way to distribute the available memory over the queries, in order to obtain the best overall reduction in response time. Unfortunately, the authors restrict themselves to single-join queries. [ND98] presents a static solution (just before execution) while [BKV98] presents static and dynamic solutions. The current paper is an extension of [BKV98] to the distributed context, with the additional consideration of the data accessibility problem.

[SS96] presents dynamic query scheduling strategies to take into consideration unpredictable response times associated with retrieving data resident on tertiary storage devices, and data access requirements of queries posed by multiple users. Reordering executions is shown to be beneficial when access latency to the data in various parts of the plan tree varies widely and dynamically.

A collection of related papers have taken into account the data accessibility problem via a technique called *scrambling*: [AFTU96], [UFA98], [AFT98]. Scrambling involves modifying query execution plans dynamically when delays are encountered during runtime. [AFTU96] considers two phases of scrambling, rescheduling and operator synthesis. When encountering delays, *rescheduling* changes the order of execution of the already-existing operations in the QEP. On the other hand, *operator synthesis* involves creating new operators that were not part of the original QEP. While the initial work was based on heuristics, [UFA98] relies on cost-based scrambling decision making that takes into account total cost or response time. Unlike our approach, the scrambling techniques rely on timeouts as a way of detecting delays. Even when only initial delays are considered, it is shown that, in the absence of good predictions of duration of expected delays, there are fundamental tradeoffs between risk aversion and effectiveness of scrambling. The authors have also mentioned that scheduling and memory management issues relating to bursty arrivals need to be considered as part of future work.

The approach taken in [ONK+97] to deal with uncertainties regarding processing times (and hence data accessibility) at the different autonomous nodes in a multidatabase context is to use a dynamic query optimization scheme. This method determines the sequence of internode operations, as partial results become available during runtime at the different nodes involved in processing subqueries of the global query. All the subqueries are initiated in parallel at the beginning. A statistical decision mechanism is used and takes into account the cost of an internode operation, its selectivity and the transmission costs involved. While this mechanism exploits inherent parallelism in the system as much as possible, unlike our approach, it does not deal with memory constraints.

The technique of using a single process for processing multiple data streams simultaneously, as data is brought into database buffers, has been successfully employed in a commercial product (DB2/MVS) to deal with uncertainties in data retrievals from different disks even in a centralized environment [MPTW94]. That technique is similar in spirit to our way of processing the message queues, as data becomes available. However, unlike [MPTW94], we associate scheduling priorities with the queues and process them in priority order as data arrives in the queues in order to minimize overheads.

8 Conclusion

In this paper, we addressed two important correlated problems that arise while processing complex queries in a distributed environment: data accessibility and memory limitation. With respect to the query processor, our goal is to avoid, whenever possible, blockage (i.e., waiting for some data that is not accessible) which leads to a dramatic increase in the query response time.

We proposed an execution strategy that reduces the query response time by concurrently executing several query fragments in order to overlap data delivery delays with the processing of these query fragments. Our approach is both proactive by the careful step-by-step scheduling of several query fragments and reactive by the processing of these fragments based on data arrivals. Pursuing this approach, we have made the following contributions:

(1) We proposed a general architecture where the proactive component, the query scheduler (QS), is responsible for planning one execution step. The reactive component, the query processor (QP), executes the orders of the QS and reacts immediately to data unavailability by following the contingency plan produced by the QS. The QP also detects situations that may invalidate the plan, thus concluding the execution step.

(2) We highlighted the important parameters that have to be taken into account to generate the plan and proposed heuristics to produce efficiently a plan which is always executable with the allocated resources and which is beneficial.

(3) We described experiments to validate the approach and the heuristics used, using a prototype implementation. The experiments show that our approach brings significant performance improvement (e.g., up to 70%) even when dealing with small data-delivery delays (e.g., 20µs per tuple). In contrast, more aggressive approaches like *materialize all* (MA) fail since MA may generate more overhead than gains.

In the near future, we plan to make more exhaustive experiments in order to tune the heuristics used when producing the plan, considering more complex situations like multi-query executions and/or parallel query executions in a multiprocessor system.

Acknowledgments: We would like to thank Patrick Valduriez, Olga Kapitskaia, Dennis Shasha, Philippe Pucheral, and Hubert Naacke for their comments, encouragement and help.

9 Bibliography

[AFT98] L. Amsaleg, M. J. Franklin, and A. Tomasic. Dynamic Query Operator Scheduling for Wide-Area Remote Access. *Journal of Distributed and Parallel Databases*, Volume 6, Number 3, July 1998.

[AFTU96] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling Query Plans to Cope With Unexpected Delays. *International Conference on Parallel and Distributed Information Systems*, 1996.

[BFV96] L. Bouganim, D. Florescu, P. Valduriez: Dynamic Load Balancing in Hierarchical Parallel Database Systems. *International Conference on Very Large Data Bases*, 1996.

[BKV98] L. Bouganim, O. Kapitskaia, and P. Valduriez. Memory-Adaptive Scheduling for Large Query Execution. *International Conference on Information and Knowledge Management*, 1998.

[CG94] R. L. Cole, and G. Graefe. Optimization of Dynamic Query Evaluation Plans. *ACM SIGMOD International Conference on Management of Data*, 1994.

[CG96] P. Corrigan, and M. Gurry. What Causes Performance Problems. In *Oracle Performance Tuning*, 1996.

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, D. A. Wood. Implementation Techniques for Main Memory Database Systems. *ACM SIGMOD International Conference on Management of Data*, 1984.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, Volume 25, Number 2, June 1993.

[GI97] M. N. Garofalakis, and Y. E. Ioannidis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. *International Conference on Very Large Data Bases*, 1997.

[HFLP89] L. M. Haas, J. Christoph Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. *ACM SIGMOD International Conference on Management of Data*, 1989.

[HM94] W. Hasan, and R. Motwani. Optimization Algorithms for Exploiting the Parallel Communication Tradeoff in Pipelined Parallelism. *International Conference on Very Large Data Bases*, 1994.

[IFF+99] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Wald. An Adaptive Query Execution System for Data Integration. To appear in *ACM SIGMOD International Conference on Management of Data*, 1999.

[KD98] N. Kabra, and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. *ACM SIGMOD International Conference on Management of Data*, 1998.

[KNT89] M. Kitsuregawa, M. Nakayama, M. Takagi: The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. *International Conference on Very Large Data Bases*, 1989

[KTM83] M. Kitsuregawa, H. Tanaka, T. Moto-Oka: Application of Hash to Data Base Machine and Its Architecture. *New Generation Computing*, Volume, Number 1, 1983

[Moh92] C. Mohan. Interactions Between Query Optimization and Concurrency Control. *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing (RIDE-TQP)*, 1992.

[MPTW94] C. Mohan, H. Pirahesh, W. G. Tang, and Y. Wang. Parallelism in Relational Database Management Systems. *IBM Systems Journal*, Volume 33, Number 2, 1994.

[ND98] B. Nag, and D. J. DeWitt. Memory Allocation Strategies for Complex Decision Support Queries. *International Conference on Information and Knowledge Management*, 1998.

[ONK+97] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac. Dynamic Query Optimization in Multidatabases. *Data Engineering Bulletin,* Volume 20, Number 3, 1997.

[SYT93] E. Shekita, H. Young, and K. L. Tan. Multi-Join Optimization for Symmetric Multiprocessors. *International Conference on Very Large Data Bases*, 1993.

[SS96] S. Sarawagi, and M. Stonebraker. Reordering Query Execution in Tertiary Memory Databases. *International Conference on Very Large Data Bases*, 1996.

[UFA98] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost Based Query Scrambling for Initial Delays. *ACM SIGMOD International Conference on Management of Data*, 1998.

[WFA95] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallel Evaluation of Multi-Join Queries. *ACM SIGMOD International Conference on Management of Data*, 1995.

[YC93] P. S. Yu, and D. W. Cornell. Buffer Management Based on Return on Consumption in a Multi-Query Environment. *VLDB Journal*, Volume 2, Number 1, 1993.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399