Dynamic Query Scheduling in Data Integration Systems

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

Execution plans produced by traditional query optimizers for data integration queries may yield poor performance for several reasons. The cost estimates may be inaccurate, the memory available at run-time may be insufficient, or data delivery rate can be unpredictable. In this paper, we address the problem of unpredictable data arrival rate. We propose to dynamically schedule queries in order to deal with irregular data delivery rate and gracefully adapt to the available memory. Our approach performs careful step-by-step scheduling of several query fragments and processes these fragments based on data arrivals. We describe a performance evaluation that shows important performance gains in several configurations.

1 Introduction

Research in data integration systems has popularized the mediator/wrapper architecture whereby a mediator provides a uniform interface to query heterogeneous data sources while wrappers map the uniform interface into the data source interfaces [13]. In this context, processing a query consists in sending sub-queries to data source wrappers, and then integrating the sub-query results at the mediator level to produce the final response.

Classical query processing, based on the distinction between compile-time and runtime, could be used here. The query is optimized at compile time, thus resulting in a complete query execution plan (QEP). At runtime, the query engine executes the query, following strictly the decisions of the query optimizer. This approach has proven to be effective in centralized systems where the compiler can make good decisions. However, the execution of an integration query plan produced with this approach can result in bad performance because the mediator has a poor knowledge of the behavior of the remote sources.

First, the data arrival rate, at the mediator from a particular source, is typically difficult to predict and control because it depends on the complexity of the sub-query assigned to that source, the load of the remote source and the characteristics of the network. Delays in data delivery may stall the query engine, leading to a dramatic increase in response time.

Second, the characteristics of the sub-query results are difficult to assess, due to the autonomous nature of the data sources. The sizes of intermediate results used to estimate the costs of the integration query execution plan are then likely to be inaccurate. Poor performance is therefore the likely result of a sub-optimal execution plan, and, of a bad estimate of the memory needed to execute the plan. Indeed, the amount of available memory at runtime for processing the integration 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* [4,12].

All these problems have led database researchers and implementers to resort to dynamic strategies to correct or adapt the static query execution plan.

1.1. Unpredictable Data Arrival Rate

In this paper, we mainly address the performance problems due to unpredictable data delivery rates in data integration systems. Dynamic strategies are a logical response to this problem as they try to adapt dynamically the query execution plan to the execution context. This adaptation can be done at three different levels:

- *at the operator level*, using relational operators that are able to absorb delays in delivery. [8] has adapted the double-pipelined hash join [16], originally designed for parallel databases. However, such an approach is restricted to hash-based queries (i.e., equi-joins).
- *at the scheduling level*, by modifying on the fly the scheduling of the operators to avoid query engine stalling [1,2,15].
- *at the query execution plan level*, by partially re-optimizing the query plan in order to adapt to the data arrival rate using estimates of such rates [1,15].

As in [8,15], we think that partial re-optimization and dynamic scheduling are complementary and should be used together to provide good performance. A general algorithm for dynamic optimization can be sketched as follows:

Produce an initial QEP
Loop
  Process the current QEP using dynamic scheduling strategies
  If scheduling strategies fail or the plan appears to be sub-optimal, apply dynamic re-optimization
End Loop

0-7695-0506-6/00 $10.00 © 2000 IEEE

Partial re-optimization of the query plan is difficult to implement and tune [15]. Moreover, the possibility for re-optimization decreases as query execution reaches completion (because of the results previously computed). [1,15] describe approaches which combine dynamic scheduling with partial re-optimization. But many problems remain at the scheduling level. Solving them may alleviate the need for dynamic re-optimization.

In this paper, we describe a new dynamic scheduling strategy to address the performance problems due to unpredictable data delivery. But such a strategy may increase memory consumption and thus interfere with decisions regarding memory allocation. Thus, we must also take into account the problem of memory limitation.

Our strategy differs significantly from previous ones. In the following, we briefly compare it to the most related approach which is *query scrambling* [1,2,15]. Further discussion and comparison with other related work is provided in [6].

1.2. Query Scrambling

The basic strategy of query scrambling is to dynamically modify the query execution plan in reaction to unexpected delays in data access. [2] defines three types of delays. (i) *Initial delay:* when a delay occurs for the first tuple only; (ii) *Bursty arrival:* when data arrives in bursts followed by long periods of no arrival; and (iii) *Slow delivery:* when the arrival rate is regular but slower than normal. In [1], bursty arrival is considered while [1,15] focus on initial delays. However, the authors have not provided any solution to the problem of slow delivery.

To handle initial delays, 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 [2] or a query optimizer [15]. This second phase is a run-time re-optimization, and, as mentioned above, can be complementary to any dynamic scheduling strategy. In what follows, we only consider the first phase.

The different scrambling techniques are all based on the same concept: react to a timeout while waiting for remote data to arrive. When this timeout occurs, a *scrambling step* takes place: The operator currently in execution, say *O1*, is suspended (as it has no input data), and a new operator, say *O2*, is selected for execution. Depending on the strategy or on configuration parameters, *O1* resumes as soon as data arrives, or *O2* is executed until it ends or until a new time-out occurs. In this last case, a new scrambling step is triggered.

Scrambling techniques have two distinct problems. First, a delaying data source may be detected too late to enable a timely reaction. For instance, if all the execution occurs normally, and a single problem arises with the last accessed data source, scrambling will be ineffective since there is no more work to scramble [1].

Second, scrambling may be difficult to configure as it can incur important overheads (*e.g.,* materialization, memory consumption). The behavior of scrambling strongly depends on configuration parameters. Consider for instance the time-out value. If its value is too large, scrambling will never occur. On the contrary, a value that is too small may trigger too many scrambling steps while simply waiting for the delayed data might have been more effective.

1.3. Proposed Approach

Scrambling techniques assume that the execution plan will, *a priori,* execute without delays and then react when such delays occur. Our approach takes the *opposite direction.* We suppose that delays will happen during the execution, and include this factor in the execution strategy.

To do that, we constantly monitor the arrival rate of each participating data source and the available memory. We use this knowledge to elaborate a scheduling plan which is periodically revised when the delivery rates change significantly. Thus, we interleave planning phases, where we plan for the near future and execution phases where we react instantaneously to delays. This constant re-evaluation of the scheduling plan allows us to *adjust* to the data arrival rate and memory consumption.

The planning phase selects and orders independent query fragments which can be processed concurrently (*i.e.,* which fit together in the available memory). The selection and the ordering are based on heuristics which use two metrics: the *critical degree* which quantifies the critical path (in terms of total retrieval time) of the query execution plan, and the *benefit materialization indicator* which represents the profitability of scheduling a query fragment.

During the execution phase, query fragments are considered for execution depending on their order and on data availability. 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 data unavailability). The query engine can switch from one fragment to another with negligible overhead and without negative consequences as the scheduling plan ensures that these fragments can be processed concurrently. Thus, the query engine is stalled only if there is no available data for all the fragments that are scheduled concurrently.

Using this strategy, we can address both the delays in data delivery and memory limitation problems. Moreover, this solution applies to any of the three previously defined types of delays 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 presents the context and the query execution problems. Section 3 describes the architecture of our query execution engine. Section 4 presents the metrics and the general strategy used by the query scheduler to dynamically optimize the query execution (more details are given in [6]). Section 5 gives an experimental validation which shows the behavior of our query engine and the performance gains we can obtain. Finally, Section 6 concludes.

2 Context and Query Execution Problems

In this section, we make precise our integration system architecture and discuss query execution.

2.1. Data Integration System Architecture

We consider a distributed architecture of specialized components with a mediator and wrappers [13]. To process a user query, the *Mediator* sends sub-queries to the wrappers, and then integrates the sub-query results to produce the final response. A *Wrapper* gives a structured view of the associated data source. It translates mediator sub-queries for the particular data source, and reformats answers for the mediator.

In this paper, we focus on the execution of the integration query by the mediator and consider the wrappers as black boxes. Figure 1 illustrates the system architecture. When the query execution starts, the query engine sends a sub-query to each participating wrapper and creates a queue of a given size in order to buffer the received tuples. The *communication manager* is responsible for receiving tuples from several wrappers and transferring them to the appropriate queues. If the relevant destination queue is full, sub-query processing at the wrapper is suspended as it cannot send more tuples, until tuples are consumed from that queue. This communication protocol is a kind of "window protocol" that has been successfully used in DB2/MVS for parallel query executions [11]. This allows controlling the flow of tuples between the wrappers and the mediator.

In order to simplify the remainder of the presentation, we choose to restrict our context by making some assumptions. First, we consider only the execution of a single query on the mediator (discussed in [6]), and thus, our objective is to optimize the response time of this query. Second, the mediator is supposed to run on a monoprocessor machine, *i.e.,* we do not consider parallel execution at the mediator. Finally, we assume that the connection between the wrappers and the mediator is a network of reasonable speed (discussed in section 5.3).

Mediator
Query Engine
Integration Query
Tuples
Communication manager
Network
Sub-Queries
Wrappers W1 W2 W3 W4
Data Sources DS DS DS DS

Figure 1: System Architecture

2.2. 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 [7]. The optimizer can consider different shapes of *QEP*. Bushy plans are the most general and the most appealing because they offer the best opportunities to minimize the size of intermediate results [14]. Hence, we consider bushy trees in this paper.

A *QEP* is represented as an *operator tree* and results from the "macro-expansion" of the query tree. 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.

Figure 2 and 3 show a query tree and its corresponding QEP with a simple query integrating five data sources (named W_A, W_B, W_C, W_D and W_E). This example will be used to explain our approach.

W_A W_B W_C W_D W_E

Figure 2: An example of query tree

A QEP can be decomposed into a set of maximum pipeline chains. 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. For instance, if the PC rooted at W_A, denoted p_A, has not finished, p_B cannot begin, otherwise the scan and the following join in p_B will not be executed in pipeline.

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

Blocking edge
Pipelinable edge
Pipeline chain
Join Mat Scan
W_A W_B W_C W_D W_E

Figure 3: The corresponding query tree

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.

2.3. Query Execution Problems

The iterator model [7] is the most popular execution model for processing relational queries. It resembles closely those of commercial systems, *e.g.*, Ingres, Informix and Oracle. In the iterator model, each operator of the QEP is implemented as an iterator supporting three different calls: *open()* to prepare the operator for producing data, *next()* to produce one tuple and *close()* to perform a final clean-up. A QEP is activated starting at the QEP root and progressing towards the leaves. The iterator model allows for pipelined execution. Moreover, the shape of the QEP fixes the order of execution, generally recursively from left to right (*e.g.*, in Figure 3, the execution order would be $\{p_A, p_B, p_C, p_D, p_E\}$).

The iterator model is simple, powerful (see [7]) and efficient. It synchronizes naturally the whole QEP within a single process. However, it has some subtle drawbacks [2,5], because it produces a sequential execution. Such an execution, i.e., consuming entirely the data produced by one wrapper before accessing another one, leads to a response time with a lower bound equal to the sum of the times needed to retrieve the data produced by each wrapper. Thus, for a given wrapper, if the time to retrieve the data is larger than the time to process it, the query engine stalls.

A first solution to the problem raised by a sequential execution is to interleave the execution of several parts of the query, i.e., PC's. For instance, with the QEP of Figure 3, p_A and p_C can be triggered concurrently at the beginning of the execution. If delivery delays occur while retrieving data from W_A, the query engine can avoid stalling by executing p_C. However, this approach is limited by the number of PC's which can be executed concurrently (due to dependency constraints or memory limitation reasons). For instance, dependency constraints require p_E to be executed alone. Therefore, this method will not apply if delivery problems appear with W_E. A possible solution is to trigger the materialization of W_E's result concurrently with the execution of p_A and p_C.

We want to devise an execution strategy that interleaves the processing or materialization of data from several wrappers in order to have the query engine doing some useful work, thus reducing the total response time.

Our objective is therefore to reduce the impact of unpredictable data delivery rate problems by scheduling several query fragments concurrently. To decrease the risk of stalling the query engine, we may schedule as many query fragments as possible. Hence, partial materialization allows for more flexibility. However, this may cause two problems. First, scheduling a partial materialization can incur high I/O overheads. Second, scheduling too many query fragments can result in disastrous performance because it can cause contention in the system.

The problem is, then, to select, schedule and execute concurrently several query fragments (i.e., PC's and/or partial materializations) 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 a query's execution. Thus, any algorithm that will fix ahead a given scheduling for the duration of the entire query execution will be unsatisfactory.

3 Query Engine

In this section, we describe our query engine's architecture and dynamic query processing.

3.1. Query Engine Architecture

Our objective is to propose a dynamic scheduling strategy. However, such a strategy needs to be integrated with other dynamic strategies for execution as it is likely that execution problems appear together. Thus, we propose a general architecture which allows integrating dynamic re-optimization techniques. This general architecture is presented in Figure 4.

Our query engine divides the query execution into several phases: planning and execution. Planning phases adapt the QEP to the current execution context. We propose to divide the planning responsibility between the dynamic QEP optimizer and the dynamic query scheduler.

The *Dynamic QEP Optimizer* (DQO) implements dynamic re-optimization strategies such as the ones described in [4,8,9,15]. Each planning phase of the DQO can potentially modify the QEP, which is passed to the dynamic query scheduler. Although our focus is not on the DQO, we include it in the architecture in order to show how the latter interacts with the dynamic query scheduler.

The *Dynamic Query Scheduler* (DQS) takes as input the QEP and produces a *scheduling plan* (SP) *i.e.,* it takes **exclusively** scheduling decisions. The scheduling plan consists of a totally ordered set of query fragments (QF's).

The *Dynamic Query Processor* (DQP) implements the execution component of the system and processes concurrently the query fragments of the SP.

The *Communication Manager* (CM) implements the communicating component of the system. It receives data from the wrappers and makes it available to the DQP. The DQP and the CM run asynchronously in a producer-consumer mode by means of communication queues. Moreover, the CM is responsible for computing an estimate of the delivery rate and signaling any significant changes to the DQP.

The DQO, the DQS and the DQP interact synchronously, *i.e.,* they never run concurrently. The DQO calls the DQS passing the QEP as an argument. The DQS, in turn, calls the DQP passing the SP as an argument. The DQP, then

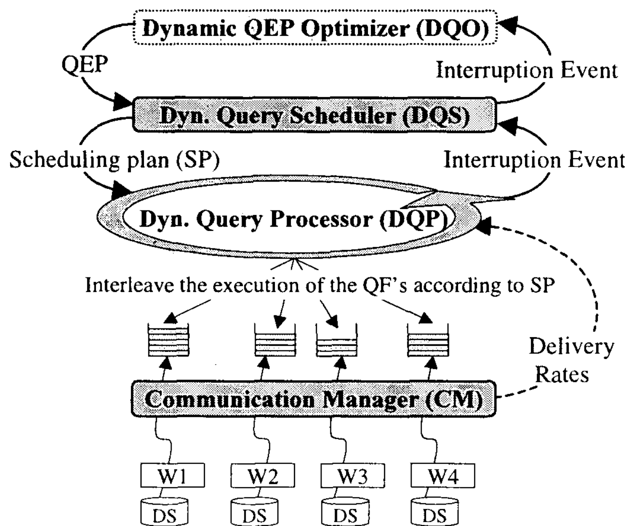

Figure 4: Query Engine Architecture

processes the query fragments of the SP and returns an interruption event informing the DQS of the reason why the execution phase must be terminated. The interruption event can be processed by the DQS, or returned to the DQO depending on its nature, thereby starting new planning phases.

Two kinds of interruption events can occur: *Normal interruptions*, signaling the end of a QF (for the DQS) or the end of the QEP (for the DQO); and *abnormal interruptions*, signaling any significant change in the system which may imply a revision of the SP (for the DQS) or even, of the QEP (for the DQO).

To include dynamic re-optimization strategies, we need to detect several events or collect information during the query processing. For the problem of inaccuracy of estimates, we must collect statistics during the query execution and transmit them to the DQO [9]. For severe data delivery problems, a timeout mechanism is necessary to detect when our dynamic strategy fails (see Section 3.2). Then the phase2 of scrambling [15] can be triggered to re-optimize the plan, taking into account the data delivery delays. Finally, the problem of memory overflow must be detected by the DQS. The strategy defined in [4] can then be implemented inside the DQO to solve such problem.

3.2. Dynamic Query Processor Algorithm

At each execution phase, the task of the DQP is to interleave the execution of the query fragments in order to maximize the processor utilization with respect to the priorities defined in the scheduling plan SP. To do so, the DQP scans the queue associated with the query fragment which has the highest priority and processes a certain amount of tuples called a *batch* (if any). If the queue does not contain a sufficient amount of tuple[1], the DQP scans the second queue in the list and so on. After each batch processing, the DQP returns to the highest priority queue. The rationale behind considering batches of tuples rather than individual tuples is to reduce the potential overheads due to frequent switches between scheduled query fragments.

Thus, the DQP is stalled only if there is no available data for all the fragments that are scheduled concurrently. In this extreme case and after a timeout has expired, the DQP returns a *TimeOut* interruption event to the DQS, which returns this event to the DQO. Query processing can be interrupted by two other events: *EndOfQF* and *RateChange*. The former happens when some query fragment has terminated, and the latter, when a significant change has occurred in the data delivery rate of some wrapper. These three events interrupt the query execution since they may change the scheduling decisions.

3.3. Dynamic Query Scheduler

The DQS must produce an SP which contains a sufficient number of query fragments in order to prevent query processor stalling. To decrease this risk and allow for more scheduling flexibility, we give the DQS the ability to trigger partial materialization of the results of wrappers which have a slow data delivery rate and are associated with PC's which are not yet schedulable. Thus, the query fragments of an SP can be PC's or partial materialization of wrappers results.

As the DQS computes repetitively the SP, at each scheduling phase 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 short time interval compared to the average processing time of one execution phase.

Three categories of parameters impact the computation of the scheduling plan at a given scheduling phase: (i) the annotated query execution plan [9] produced by the DQO; (ii) the total available memory for the query execution, which is assumed not to change during the query execution; and (iii) The current data delivery rate provided by the CM.

The *annotated query execution plan* contains the following parameters: (i) the QEP including blocking and pipelinable edges; (ii) the memory requirement of each operator *op*, denoted *mem(op)* - *i.e.*, size of the memory needed to process the operator; and (iii) the estimated size of each operator result. The estimated result size may impact the overhead induced by the introduction of a materialization, under the responsibility of the DQS. The memory needed is necessary to compute the number of query fragments which can be scheduled concurrently.

4 Dynamic Query Scheduler Strategy

In this section, we describe the way the query scheduler works, i.e., the scheduling conditions it uses, the way it handles memory limitations and critical PCs, its materialization strategy and the way it computes the SP.

[1] Notice that batch size can vary dynamically.

4.1. Schedulability Conditions

At each scheduling phase, the optimization potential of the DQS depends how many query fragments (QF's) may be schedulable concurrently. The schedulability conditions are the following:

Dependency constraints: As said before, 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 3, ancestors*(p_E) = $\{p_A, p_B, p_C, p_D\}$.

Memory constraints: 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.

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

4.2. Handling Memory Limitations

Dependency constraints imply that a PC p will never be scheduled until all its operands are ready. Thus, when p is C-schedulable, we know exactly the sizes of the operands of p, and thus, can deduce the M-schedulability property, based on these exact values (and not on estimates). Note that p can be discovered to be not M-schedulable earlier, when the sum of the sizes of the already computed operands is greater than the available memory. If p is not M-schedulable, the DQP cannot process p, even alone, in the available memory without generating paging, which may thrash the system [4].

As we wish to avoid such a situation, the only solution is to revise the query execution plan, which is the responsibility of the dynamic QEP optimizer. Therefore, the query scheduler suspends execution when a PC is discovered to be not M-schedulable and informs the dynamic optimizer which must change the query execution plan in order to resume execution.

As a consequence, the dynamic optimizer must, at least, include a module which deals with these memory problems. One simple solution is to use the technique devised in [4]. It 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 the first created fragment is necessarily M-schedulable.

4.3. Detecting Critical PC's

Now, given the input parameters of the DQS and a PC *p*, we want to estimate how critical the execution of *p* may be, by incurring query processor stalling. A critical situation occurs if the data delivery rate is less than the processing rate of incoming tuples of *p*. At any moment, the communication manager is aware of the instantaneous data arrival rate. Thus, it is able to compute dynamically an estimated value of the averaged data delivery rate.

Critical degree: Let *p* be a PC, and W the associated wrapper. Let n_p be the number of tuples produced by W. Let w_p be the averaged time interval between the arrival of tuples from W (called the *waiting time* of *p*). Notice that $w_p = 1/d_p$ if d_p is the averaged data delivery rate for W. Let c_p be the average processing time of one tuple from W. 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.

The critical degree represents the total CPU idle time when p is computed with no other concurrent PC's. 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. 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 work to schedule concurrently.

4.4. Materialization Strategy

It may happen that a critical PC *p* is not schedulable because of dependency constraints. In order to prevent query processor stalling due to a late scheduling of *p*, it is the responsibility of the DQS to trigger the materialization of the results of the wrapper *W* associated with *p*. We call such an operation a PC ***degradation***. It consists of splitting *p* into two query fragments. The ***Materialization Fragment(p)*** or MF*(p)* retrieves tuples from the associated wrapper, applies the first scan operator of p (if any) and materializes the result in a temporary relation. The ***Complement Fragment(p)*** or CF(*p*) takes as input this temporary relation and includes the remaining operators of *p*. As MF(*p*) has no ancestor and consumes a negligible amount of memory, it is always schedulable. This feature is of particular interest for non-schedulable PC's having high critical degree.

The problem of PC degradation is that it leads to the materialization of originally pipelined intermediate results, thus incurring possibly high I/O overheads. Thus, the use of PC degradation technique is conditioned by the definition of a profitability criterion. The best criterion for a given PC *p* is the ratio of the expected execution cost of the PC *p* versus the sum of the expected execution cost of MF(*p*) and of CF (*p*). However, such costs are impossible to estimate as they highly depend on the number of QF's scheduled when executing these QF's. For instance, suppose that we want to compute this ratio for p_B when the QEP of Figure 3 starts its execution. We must estimate the cost of executing MF(p_B) now, and of executing CF(p_B) later, and compare it to the cost of executing p_B later. As we cannot know what will be the amount of work done when executing CF(p_B) or p_B, we cannot predict the influence of the delivery delays on the response time, *i.e.*, we don't know if they will be absorbed by a concurrent execution of others QF's or not. Moreover,

the costs of materialization overheads depend on the disk activity at the time of execution. Therefore the principal components of the cost are impossible to estimate accurately. Thus, we use a rough approximation called *benefit materialization indicator* which gives an indication of the profitability of materialization.

***Benefit materialization indicator* (bmi):** Let p be a critical PC, IO_p, the I/O cost for reading or writing a tuple produced by MF(p) and w_p the averaged waiting time of p. Bmi is given by: $bmi = w_p / 2\ IO_p$ (we ignore the first scan selectivity for the ease of presentation).

To arrive at this simple formula, we suppose that MF(p) is executed concurrently with others QF's. Thus, we assume that no idle time occurs, however, we account for all the IO costs. For CF(p), we assume that the I/O and CPU operations for CF(p) are done concurrently (asynchronous I/O). Finally, for the PC p, we consider that the DQP processes a tuple while waiting for the next tuple.

Benefit materialization threshold (bmt): High *bmi* means that the overhead induced by the PC degradation *is* negligible, compared to the response time improvement while low *bmi* means that PC degradation may increase the response time (and the total work). *Bmi* is a continuous function which is an indicator of the importance of performing PC degradation. Consequently, we define a constant threshold value of *bmi* named *bmt* which is an input for our query scheduler; *bmt* is the minimum value of *bmi(p)* below which the PC p should not be degraded.

4.5. Strategy for Computing a Scheduling Plan

At each scheduling phase, the DQS computes an SP by using the annotated query execution plan, a set of heuristic rules, the current state of the query execution (*e.g.*, data arrival rates estimations and the available memory) and the benefit materialization threshold (bmt). The DQS first computes the set of schedulable PC's. It then selects non C-schedulable PC's for degradation when bmi is greater than bmt. Then it establishes a priority order between these PC's using the critical degree of the PC's. Finally the DQS uses this priority order, and memory constraints (*i.e.*, ensures that the scheduling plan fits in the available memory) to extract a scheduling plan. Due to space limitation, we do not detail each phase in this paper. The heuristics used and the rationale for these heuristics are detailed in [6].

5 Experimental Validation

The objective of our experiment is to assess the possible gain brought by our dynamic scheduling. Therefore, we compare our approach with a classical iterator based execution and with another strategy proposed in [1].

Understanding the behavior of several execution strategies is a difficult task, requiring sound comprehension of the influence of the parameters of (i) the experimentation platform (e.g., CPU speed, configuration), (ii) the benchmark (queries, relations size, shape of the QEP), and (iii) 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. However, simulation cannot produce real values for some overheads. For instance, it will be very hard to assess the overheads due to *context switching, i.e.,* when the DQP switches from one QF to another one.

For these reasons, we used a performance evaluation methodology similar to [3]. We fully implemented our dynamic execution model and all the compared strategies in order to capture exactly their behavior. We simulated the data sources and the relational operators. With this approach, query execution does not depend on relation content and it can be simply studied by setting relation parameters (cardinality and selectivity).

In the rest of this section, we describe our experimentation platform and report on performance results focussing only on the delays in data delivery problem, i.e., assuming the existence of sufficient memory for a classical iterator based execution. We have chosen 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 increasing the delay between the arrival of two tuples of all incoming relations.

5.1. 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.

5.1.1. 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 input relations are delivered by distinct wrappers. The query was generated using the algorithm of [14] and optimized in a classical dynamic programming query optimizer. The resulting QEP is shown in Figure 5.

Other queries, differing by their complexity, size and shape, were tested in the same manner. Even if these parameters influence the behavior of our query engine, we check that the results presented in the rest of this section are representative of the general behavior of the query engine.

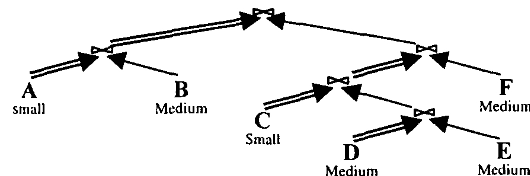

Figure 5: QEP used for the experiments

5.1.2. 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 unpredictable data delivery rates.

We denote our strategy *DSE*, Dynamic Scheduling Execution. The different heuristic rules presented in [6] were implemented in an efficient algorithm which recursively computes the QFs' priorities, beginning with the most critical PC.

The last strategy is the fairly simple *Materialize All*, denoted by *MA* and proposed in [1] which proceeds in two phases. In the first phase, *MA* materializes simultaneously on the disk of the mediator 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.

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:

$$LWB(Q) = \max\left(\sum_{p \in Q} n_p c_p \;,\; \max_{p \in Q}(n_p w_p) \right)$$

where c_p denotes the average per tuple CPU cost for a PC p.

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 [17] for the simulation. They are presented on Table 1. The prototype is written in C and runs on a Sun Ultra 1.

Table 1: Simulation parameters

Parameter	Value
CPU Speed	100 Mips
Disk : Latency - Seek Time - Transfer Rate	17 ms - 5ms - 6 MB/s
I/O Cache Size	8 pages
Perform an I/O	3000 Instr.
Number of Local Disks	1
Tuple Size - Page Size	40 bytes - 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.

5.1.3. Experimentation Methodology

For all the experiments, we simulate for each wrapper data delivery delays. To do so, we delay the production of each tuple by a delay uniformly distributed in [0, 2*w*], thus resulting in an average waiting time (as defined in section 4.3) of w. The value of w represents the per tuple average processing time, to produce the remote tuple, plus the averaged time to send the tuple to the mediator site.

We simulate wrappers which are producing tuples without any particular delays, by incurring a minimum waiting time called, w_{min}, between tuples, assuming that, at the source, the tuples are read sequentially and sent using a network at 100 Mb/s. We obtain a value of 20 µs. In fact, the remote processing time to produce a tuple is generally more than the time to read sequentially a tuple from a disk (e.g., if there is a selection predicate on the wrapper). Nevertheless, in the experiment, a PC p can be *critical* even if we use w_{min} for its input relation (depending on the processing time of p).

In the following experiments, we vary the w value for one or more input relations. To simplify the presentation, we say in the following that we *slow down* these relations. We fix the benefit materialization threshold (*bmt*) to 1 as the experiment has been done considering a single query execution.

We repeat each measurement 3 times and compute an averaged value of the 3 response times.

5.2. Performance Comparison with one Slowed-down Input Relation

For this first experiment, we slow down only one input relation and measure the response time of the three strategies. We perform this experiment slowing down successively each input relation of the QEP to observe the influence of the *position* of the slowed-down relation in the QEP. Figure 6 and 7 show 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.

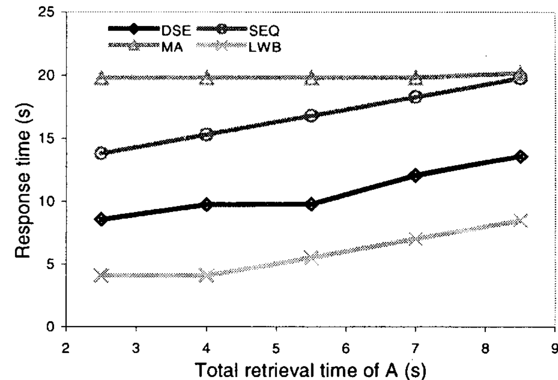

Figure 6: One Slowed–down Relation Experiments (A)

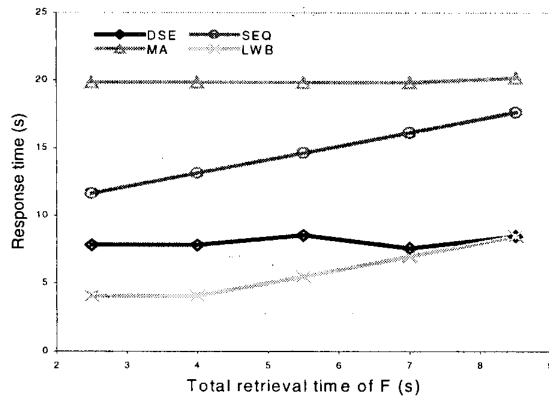

Figure 7: One Slowed–down Relation Experiments (F)

MA's response time is always worse in these experiments and stays constant with a slight increase after 8 seconds of 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 problem 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 delays in data delivery 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 $w=w_{min}$. In fact, w_{min} 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 [11].

5.3. Performance Comparison with Several Slowed-down Input Relations

In the previous experiment, we used an underestimated value of w_{min} in order to isolate the effect of the slowed-down relation. In this experiment, we did not slowdown any particular wrapper but induced an increasing w_{min} to all wrappers. Figure 8 shows the performance gain of DSE over SEQ with respect to the w_{min} value.

Basically, the performance gain increases with the w_{min} value and goes up to 70%. However, we observe an irregularity when w_{min} is around 35µs. Checking the execution traces, we have observed that this "bad" value is due to the heuristic nature of our DQS strategies, *i.e.*, the DQS has computed a wrong SP. This is due to the fact that a total ordering of QF is difficult to achieve, specifically when several PC's have quite the same critical degree.

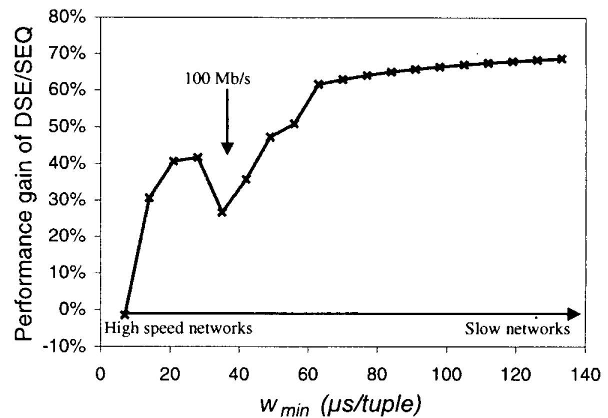

Figure 8: Varying the W_{min} value

This experiment also shows the behavior of our strategy depending on the network bandwidth. A high value of w_{min} is representative of slow networks while a low value corresponds to high speed networks. Remember that for a network of 100 Mb/s, the w_{min} value is of 20 µs (pointed in Figure 8).

5.4. Discussion

The lessons learnt 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.

One could be surprised by the bad performance of MA, compared to those shown in [1]. 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. Moreover, MA only overlaps delays between several incoming relations and does not attempt to overlap those delays with the processing of the incoming tuples.

In fact, the context and the problems considered are different than those studied in [2,15]. [15] only considers initial delays while [1] proposes a solution for bursty arrivals. In our experiments, we focus on the slow delivery problem and thus, cannot use these techniques. Our dynamic scheduling technique is more general as it can apply to any kind of delays.

6 Conclusion

In this paper, we addressed two important correlated problems that arise while processing queries in data integration systems: unpredictable delays in data delivery and memory limitation.

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 solution applies to any kind of delay (initial delay, bursty arrival and slow delivery). 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.

We have made the following contributions:

We handle unpredictable delays by dividing the responsibility for query execution between the dynamic query scheduler (DQS) and the dynamic query processor (DQP). DQP has the ability to interleave the execution of several query fragments in order to react immediately to delays. To do that, the DQP follows a scheduling plan provided by the DQS. Planning and execution phases are interleaved: A planning phase is triggered when the DQP detects situations possibly invalidating the current scheduling plan.

We highlighted the important parameters that have to be taken into account to generate the scheduling plan which is always executable with the allocated resources and which is beneficial.

We described experiments to validate our approach, using a prototype implementation. The experiments show that our approach brings significant performance improvement 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.

Finally, we integrate our dynamic scheduling approach in a more general dynamic query engine. Our dynamic query engine permits an easy integration of dynamic re-optimization strategies like the ones described in [4,9,15].

In the near future, we plan to make more exhaustive experiments in order to tune the heuristics used when producing the plan.

We also plan to study the behavior of our approach in the context of multi-query execution. As soon as we consider such context, we face the classical tradeoff between throughput and response time. Indeed, our strategy can reduce significantly the response time at the expense of a potential increase of total work. However, 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 [10], and thus may increase throughput [17]

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

7 Bibliography

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

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

[3] L. Bouganim, D. Florescu, P. Valduriez. Dynamic Load Balancing in Hierarchical Parallel Database Systems. *Int. Conf. on VLDB*, 1996.

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

[5] L. Bouganim, D. Florescu, P. Valduriez. Load Balancing for Parallel Query Execution on NUMA Multiprocessors, *Journal of Distributed and Parallel Databases*, 7 (1), January 1999.

[6] L. Bouganim, F. Fabret, C. Mohan, P. Valduriez. Dynamic Query Scheduling in Data Integration Systems (Extended Version) available on http://rodin.inria.fr/dataFiles/BFMV99e.ps

[7] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25 (2), June 1993.

[8] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Wald. An Adaptive Query Execution System for Data Integration. *ACM SIGMOD Int. Conf. on Management of Data*, 1999.

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

[10] C. Mohan. Interactions Between Query Optimization and Concurrency Control. 2nd Int. Workshop on Research Issues on Data Engineering: Transaction and Query Processing, 1992.

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

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

[13] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. 2nd Edition. Prentice Hall, 1999.

[14] E. Shekita, H. Young, and K. L. Tan. Multi-Join Optimization for Symmetric Multiprocessors. *Int. Conf. on VLDB*, 1993.

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

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

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