A Dynamic Query Processing Architecture for Data Integration Systems

Luc Bouganim†,⋆, Françoise Fabret⋆, C. Mohan⋆,‡, Patrick Valduriez⋆

†PRiSM
Versailles, France
Luc.Bouganim@prism.uvsq.fr

⋆INRIA
Rocquencourt, France
FirstName.LastName@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 the data delivery rate can be unpredictable. All these problems have led database researchers and implementors to resort to dynamic strategies to correct or adapt the static QEP. In this paper, we identify the different basic techniques that must be integrated in a dynamic query engine. Following on our recent work [6] on the problem of unpredictable data arrival rates, we propose a dynamic query processing architecture which includes three dynamic layers: the dynamic query optimizer, the scheduler and the query evaluator. Having a three-layer dynamic architecture allows reducing significantly the overheads of the dynamic strategies.

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 run-time, 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 poor performance because the mediator has limited 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. It depends on the complexity of the sub-query assigned to the source, the load of the source and the characteristics of the network. Delays in data delivery may stall the query engine, leading to a dramatic increase in response time.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

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 QEP are then likely to be inaccurate. As the amount of available memory at runtime for processing the integration query may be much less than that 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 implementors to resort to dynamic strategies to correct or adapt the static QEP. In this paper, we identify the different basic techniques that must be integrated in a dynamic query engine. This is based on our recent work [6] on the problem of unpredictable data arrival rates. We then propose a dynamic query processing architecture which includes three dynamic layers: the dynamic query optimizer, the scheduler and the query evaluator. Having a three-layer dynamic architecture allows reducing significantly the overheads of the dynamic strategies.

The remainder of the paper is organized as follows. Section 2 presents the context and the query execution problems. In Section 3, we derive from the experience of [6] the basic concepts of a dynamic architecture and describe the architecture of our dynamic query execution engine. In Section 4 we describe the specification of the query engine components which we exemplify with the solution given in [6] for unpredictable delays. Finally, Section 5 concludes.

2 Problem Formulation

An integration query is nothing else than a standard centralized query except that the data are collected in remote sources instead of being extracted from local storage units. In this section, we first present standard query processing techniques and show their problems. Query processing is classically done in two steps. The query optimizer first generates an "optimal" 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].

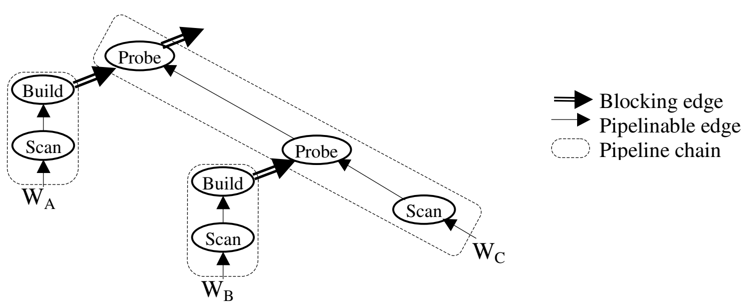

Figure 1: A simple Query Execution Plan (QEP)

A QEP is represented as an operator tree. Nodes represent atomic physical operators and edges represent data-flow. 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 1 shows a QEP for a query integrating three data sources represented by wrappers W_A, W_B and W_C, respectively. The three wrapper's results are joined using asymmetric join operators (e.g., hash-join) that have one blocking input and one pipelinable input, and produce a pipelinable output. Such QEP can be decomposed in three fragments called *pipeline fragments*[1] (PF's): p_A, p_B and p_C. A pipeline fragment (PF for short) is a maximal set of physical operators linked by pipelinable edges.

[1] With standard asymmetric join operators, pipeline fragments reduce to pipeline chains.

The most popular execution model for processing relational queries is the *iterator model*[7]. 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. For example, a query engine implementing the iterator model would execute the QEP of Figure 1 following the order p_A, p_B, p_C. So, standard query processing techniques plan completely the execution and thus, cannot react to unpredictable situations which may compromise planning decisions taken at several levels:

- at the QEP level, the actual values of parameters (cardinalities, selectivities, available memory) are far from the estimates made during planning, thus invalidating the QEP [8,9];
- at the scheduling level, when the query engine faces unpredictable delays while accessing remote data. The query engine then stalls, thereby increasing the response time [1,2,6,15].
- at the physical operator level, discovering, for instance, that the available memory for the operator execution is not sufficient [4,8].

Poor performance of integration query processing lies in the fact that the execution is fully specified before it starts, and is never revised until it finishes. The problem is therefore to define a query engine architecture which allows to dynamically adapt to unpredictable situations arising during the execution.

3 Dynamic Query Processing

The performance problems of integration queries can be solved by means of *dynamic strategies* that try to adapt dynamically to the execution context. This adaptation can be done at three different levels:

- at the QEP level, by partially re-optimizing the query plan in order to adapt to the actual values of cardinality, selectivity and available memory [8,9].
- at the scheduling level, by modifying on the fly the scheduling of the operators to avoid query engine stalling [1,2,6,15].
- at the operator level, using auto-adaptive relational operators [4, 8].

These techniques are complementary and should be used together to provide good performance [8,15]. Indeed, 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). Solving most problems at the operator or scheduling level can alleviate the need for dynamic re-optimization.

A general algorithm for dynamic processing can thus be sketched as follows:

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

Implementing dynamic strategies require to design a new query engine architecture. In this section we give an overview of our recent work [6], focusing on the problem of unpredictable data arrival rates. Based on that work, we identify the basic techniques, necessary to introduce dynamism into a query engine. We then present the dynamic query engine architecture.

3.1 Unpredictable Data Arrival Rate

The iterator model 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. Handling delays in data arrival is therefore crucial for obtaining good performance. A first solution to the problem raised by a sequential execution is to interleave the execution of several PF's. For instance, with the QEP of Figure 1, p_A and p_B 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_B. However, this approach is limited by the number of PF's which can be executed concurrently due to memory limitation or dependency constraints. For non-executable PF's (e.g., p_C which must be executed alone for dependency constraint reasons), the delays can be amortized by triggering a materialization of the associated wrapper's result (e.g., W_C). Such a materialization occurs while executing concurrently other executable PF's (e.g., p_A and p_B). However, scheduling materializations can incur high I/O overheads. Thus, these overheads must be estimated and compared with the potential benefit. Since the data delivery rate is typically unpredictable and may vary during the query's execution, we must monitor it along the execution and react to any important variation. Thus, the materialization of a given wrapper's result must stop as soon as the PF becomes executable or the benefit becomes too small (if for instance, the delivery rate increases).

Analyzing this problem and its intuitive solution, we can identify some basic techniques, necessary to implement such a dynamic strategy: (i) decomposition of the QEP into PF's; (ii) (partial) materialization of the wrapper's result; (iii) Concurrent execution of (partial) materializations and PF's; and (iv) execution monitoring in order to react with up-to-date meta-data.

3.2 Dynamic Query Engine Architecture

The main property of a dynamic query engine is to divide the query execution into several phases: planning and execution. Planning phases adapt the QEP to the current execution context. Execution phases stop when the execution context changes significantly. The architecture of a dynamic query engine must include planning components, execution components and define the interaction between these components.

We propose a dynamic query engine (see Figure 2) where the planning responsibility is divided between the dynamic query optimizer and the dynamic query scheduler.

The *Dynamic Query Optimizer* (DQO) may implement 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 on to the dynamic query scheduler.

The *Dynamic Query Scheduler* (DQS) takes as input the QEP and produces a *scheduling plan* (SP) i.e., it makes exclusively scheduling decisions. The scheduling plan consists of a set of *query fragments* (QF's) that can be evaluated concurrently, i.e., pipeline fragments which fits together in memory and have no dependency constraints and partial materializations. The SP also includes priority information for QF execution.

The *Query Fragment Evaluator* (QFE) implements the execution component of the system and evaluates concurrently the query fragments of the SP, following the specified priorities.

The DQO, the DQS and the QFE 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 QFE passing the SP as an argument. The QFE, then evaluates 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).

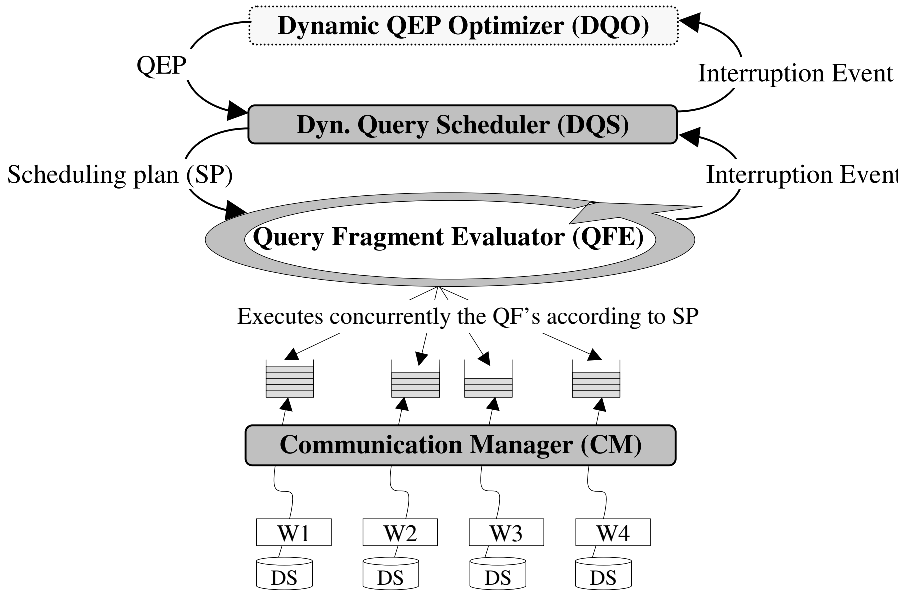

Figure 2: Dynamic Query Engine Architecture

Finally, the *Communication Manager* (CM) implements the communicating component of the system. It receives data from the wrappers and makes it available to the QFE. The QFE and the CM run asynchronously in a producer-consumer mode by means of communication queues.

The proposed architecture is hierarchical. The highest layers have large dynamic capabilities (e.g., the DQO may change completely the QEP), while the lowest layers have more restricted capabilities. As in any hierarchical architecture, each layer should process the interruption event when it is of its domain and refer to higher layer when it is not. However, an important concern is that, although there are more capabilities at the highest layers of the architecture, they have higher risk and cost.

This architecture allows one to implement dynamic strategies which favor low-layer, cheap, and secure reactions in order to minimize higher-layer, expensive, and risky reactions. For instance, when dealing with unpredictable delays, the strategy presented in [6] is based on dynamic scheduling techniques. It would invoke dynamic re-optimization only when there is no more possible dynamic scheduling reaction since, in that case, re-optimization is hazardous [15].

4 Implementing Dynamic Strategies

The architecture we propose provides a uniform way to implement dynamic strategies. In this architecture, a dynamic strategy is described by specifying the DQO, the DQS and the QFE. In this section, we specify each component which we exemplify with the solution given in [6] for unpredictable delays.

4.1 Query Fragment Evaluator

The Query Fragment Evaluator must obviously evaluate the QF's with respect to the scheduling plan. Additionally, it is responsible for detecting abnormal situations (e.g., delays, estimate inaccuracy) and producing interruption events. This requires to divide the query fragment execution into atomic execution steps followed by detection steps. Thus, when designing the QFE for a given strategy, we need to define (i) the granularity of the atomic execution steps; (ii) the situations to detect and (iii) the reaction to each situation. Depending on the situation, the reaction can be handled by the QFE itself or by the higher levels.

In [6], our objective was to design a QFE which interleaves the QF's execution in order to overlap delays in data delivery with respect to the priorities defined in the scheduling plan. During an atomic execution step, the QFE scans the queue associated with the QF which has the highest priority and processes a batch of tuples. If the queue does not contain a sufficient amount of tuples (abnormal situation), the QFE scans the second queue in the list and so on (reaction of the QFE). When a QF evaluation ends or a significant change has occurred in the data delivery rate, the QFE returns an *EndOfQF* or *RateChange* interruption event. Finally, if the QFE is stalled (i.e. there is no available data for all the QF's that are scheduled concurrently) the QFE returns a *TimeOut* interruption event. These events interrupt the QF's evaluation since they may change the scheduling decisions.

4.2 Dynamic Query Optimizer and Scheduler

Specifying the DQO and DQS requires to describe (i) their strategy; (ii) the events to which they react; and (iii) the corresponding reactions. Additionally, the DQS may produce events for the DQO.

In [6], our objective for the DQS was to produce an SP which contains a sufficient number of query fragments in order to prevent query evaluator stalling. The important parameters for computing a scheduling plan which is always executable with the allocated resources and which is beneficial are (i) the QEP; (ii) the memory requirement of each pipeline fragment; (iii) the total available memory for the query execution; and (iv) information which allows to estimate the gain brought by partial materialization. The details of the DQS strategy are given in [6].

The DQS reacts to three events sent by the QFE, namely EndOfQF, RateChange and TimeOut. Since these events affect at various degrees the scheduling decisions, the reaction is to recompute the scheduling plan . The DQS is also in charge to detect that a QF cannot be evaluated using the available memory. In this case, it sends a *MemoryOverfow* event to the DQO since a QEP modification is necessary.

The solution provided in [6] does not handle QEP sub-optimality. The unique dynamic feature of the DQO is to handle the MemoryOverflow event. In that case, the DQO applies a strategy similar to the one developed in [4], i.e., break a pipeline fragment in order to reduce memory consumption.

5 Conclusion

Static QEPs for data integration queries may yield poor performance because of inaccurate cost estimates, insufficient memory at run-time or unpredictable data delivery rate. A good solution is to resort to dynamic strategies to correct or adapt static QEPs. In this paper, we identified the different basic techniques that must be integrated in a dynamic query engine. We proposed a dynamic query processing architecture which is general enough to support a large spectrum of dynamic strategies. The architecture is hierarchical and includes three dynamic layers: the dynamic query optimizer, the scheduler and the query evaluator. The highest layers have larger dynamic capabilities than the lowest layers. This allows one to implement dynamic strategies which favor low-layer, cheap, and secure reactions in order to minimize higher-layer, expensive, and risky reactions.

Dynamic query processing for data integration systems is still in its infancy. Much more work is needed to better understand its potential. Research directions in this area should include : prototyping and benchmarking with performance comparisons with static query processing; devising guidelines or heuristics to decide when to avoid reoptimization; dealing with distributed mediator systems such as LeSelect [18] which provide more parallelism for data integration.

References

[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. Int. Conf. on Data Engineering, 2000.

[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. Weld. 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.

[18] A Mediator System Developed in the Caravel Project, Inria, France, *http://rodin.inria.fr/Eprototype_LeSelect.html*.