Self-Monitoring Query Execution for Adaptive Query Processing

Anastasios Gounaris[a,*] Norman W. Paton[a]
Alvaro A. A. Fernandes[a] Rizos Sakellariou[a]

[a]*Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK*

Abstract

Adaptive query processing generally involves a feedback loop comprising monitoring, assessment and response. So far, individual proposals have tended to group together an approach to monitoring, a means of assessment, and a form of response. However, there are many benefits in decoupling these three phases, and in constructing generic frameworks for each of them. To this end, this paper discusses monitoring of query plan execution as a topic in its own right, and advocates an approach based on self-monitoring algebraic operators. This approach is shown to be generic and independent of any specific adaptation mechanism, easily implementable and portable, sufficiently comprehensive, appropriate for heterogeneous distributed environments, and more importantly, capable of driving on-the-fly adaptations of query plan execution. An experimental evaluation of the overheads and of the quality of the results obtained by monitoring is also presented.

Key words: query monitoring, adaptive query processing, query execution, operators

1 Introduction

Adaptive query processing (AQP) is particularly relevant to settings in which query planning must take place in the presence of limited or potentially inaccurate statistics for use by the query optimiser, and where queries are evaluated

* Corresponding author.
Email addresses: `gounaris@cs.man.ac.uk` (Anastasios Gounaris), `norm@cs.man.ac.uk` (Norman W. Paton), `alvaro@cs.man.ac.uk` (Alvaro A. A. Fernandes), `rizos@cs.man.ac.uk` (Rizos Sakellariou).

in environments with rapidly changing computational properties, such as loads or available memory (12). As such, the relevance of AQP is growing with the prevalence of computing environments that are characterized by a lack of centralised control, such as the web and the Grid (8). Such environments are not only inherently more complex to model, but it is often the case that runtime conditions are sufficiently volatile to compromise the validity of predictions. As a result, robust cost models are harder to come by, thereby reducing the likelihood that the optimiser will select a sufficiently efficient execution plan (17; 22). Useful statistics, like selectivities and histograms (e.g., (4; 7; 13)), may be inaccurate, incomplete or unavailable. Adaptivity is likely to prove crucial, and, for that purpose, precise, up-to-date, efficiently obtainable data about runtime behaviour is essential.

AQP generally involves a feedback loop in which there is *monitoring*, *assessment* and *response*. The execution of a plan and the execution environment itself are monitored, an assessment is made relating to the progress of the execution, and a response may be taken that affects the continuing evaluation of the query. The response may be fine grained (e.g., directing the next tuple to a particular node) or coarse grained (e.g., rerunning the optimiser over some or all of the query). In AQP, monitoring is not normally addressed as a topic in its own right. Rather, individual proposals either tend to group together an approach to monitoring, a means of assessment, and a form of response (e.g., (30)); or just take the existence of monitoring for granted (e.g., (27)). Simply assuming that the monitoring information that drives adaptation is in place justifies both the necessity and the pertinence of dealing with monitoring separately. On the other hand, grouping all the phases of adaptivity together has led to many interesting techniques for AQP, but to date no general framework has been constructed for identifying or composing generic techniques for monitoring, assessment or response. For example, one could envisage a particular approach to monitoring being used with different forms of assessment and response, or different categories of response being made in the light of a single approach to monitoring and assessment.

This paper discusses the monitoring of query execution as a topic in its own right. Monitoring the execution of a query can provide evolving estimates for properties of the query, such as its completion time and the number of values in its result. Such information can be useful for providing feedback to users, refining cost models, and suggesting circumstances in which runtime adaptation of a query plan is likely to yield improved performance, which is the most challenging task. Three different approaches to monitoring can be identified in the literature: use of an independent and centralised component within the query processor for monitoring (e.g., (3; 19)); construction of new physical query operators dedicated to statistics collection (e.g., (16)); and transformation of traditional operators to self-monitoring ones (e.g., (11)). A centralised component, apart from requiring significant changes in the architecture of query engines, does not scale well in parallel or distributed settings, due to the communication overhead incurred. Dedicated operators require modifications in the query optimisers, which are responsible for deciding which monitoring operators are employed for each query and where. Both centralised components and dedicated operators suffer from limitations in the scope of the monitoring information that can be gathered. For example, a dedicated monitoring operator can collect useful information about the value distribution of intermediate results, but cannot provide any information about the time cost of other operators in the query plan, as it can only monitor the data it processes. On the other hand, a centralised component can observe the behaviour of algebraic operators and their cost, but cannot monitor data properties like value distribution. Our approach is based on self-monitoring operators that capture metrics in the form of counters, timings (i.e., placing two timestamps and computing their difference), and computations of tuple sizes. In this way, modifications are required neither in the engine architecture nor in the optimiser, but only in the implementation of the operators themselves, which is, in our opinion, less disruptive to the query compilation and evaluation architecture. This feature, combined with the simple nature of the metrics (i.e., counters, timings, and size computations), makes the approach portable and implementable. Moreover, it enables the communication of monitoring information between nodes in the same way as the data items manipulated by operators are exchanged (e.g., through the *exchange* operator (9) in the operator model of parallel execution), and is thus more appropriate for multi-node environments like the Grid (25).

Additional features of the approach presented in this paper are:

(1) It covers a broad range of query execution aspects, although it is based only on counters, timings, and size computations, as discussed in Section 2. Moreover, it provides monitoring information that is sufficient to support most AQP proposals to date, as demonstrated in Section 6.
(2) It is able to collect information that is directly relevant to the assessment process of adaptivity by establishing where a plan is deviating from its anticipated behaviour. In other words, it can provide the necessary background for on-the-fly adaptation, as discussed in Section 3.
(3) It accommodates different levels of detail in the monitoring information, monitoring frequency and data movement. In particular, the paper discusses instantiations of the approach in which (i) no monitoring data is passed between the operators of the algebra, (ii) monitoring data is passed between operators of the algebra only within a single computational node, and (iii) information is passed between computational nodes in a distributed plan. Thus the approach is able to trade-off monitoring quality against monitoring overhead, as discussed in Section 4.
(4) As the *cost* of monitoring and the *quality* of the results obtained by monitoring are important, experiments have been conducted on both these

features, which are presented in Section 5. We cannot describe this cost as low or high as there is no general consensus on these terms, but we feel that the overheads incurred are reasonable and our results encouraging.

2 The scope of the approach: what can be captured

In many query processors, a declarative query is transformed, after being parsed, into an operator tree that is also referred to as a query plan. Usually, the query is mapped into a logical algebra and then into a physical one. This section identifies measurements that can be taken from physical operators in order to support monitoring tasks, such as accounting, adaptation, and calibration of the cost model employed. Although the measurements should be able to be expressed as counters, timings, or sizes, this is not a very restricting limitation as they can cover, as shown below, a broad range of operator properties. There are two kinds of measurements, corresponding to two different levels of monitoring: generic measurements that can be applied to any physical operator, e.g., index-scan, hash join and so on; and operator-specific measurements that decompose the operator's functionality into simpler parts, and that are essential for monitoring at a finer granularity.

2.1 Operator-independent information

Table 1 presents quantifiable properties that are common to all physical operators. Such properties are not related to specific implementations or functionalities of the operators, and cover the following distinct aspects of their behaviour:

(1) Operator workload and selectivity: the measurements that are useful for that are the number of tuples consumed n_{inp}^{j}, which is equal to the number of tuples processed, and the number of tuples produced n.
(2) Operator cost: for monitoring the cost of the operator, various timings can be captured. The time elapsed since the operator's instantiation t reflects the evaluation time of that operator in systems where all tuples are first processed by one operator before being sent to another. In systems that follow a different approach (e.g., the iterator model of query execution (10)), in the absence of blocking operators, this time may converge for all the operators in a query plan, and approximate the query execution time. In such systems the time the operator is active t_{real} is not the same as the time elapsed since initialisation. At a finer granularity, t_{tuple} gives the time cost for each data item processed.
(3) Resource requirements: when an operator needs to maintain certain state,

Table 1
General measurements on a physical query operator.

Symbol	**Description**
n	number of tuples produced so far
n_{inp}^{j}	number of tuples received from the jth input
t	time elapsed since the operator was created
t_{real}	time the operator is active
t_{tuple}	time to process a tuple, i.e. time to evaluate the next() function in the iterator model (10)
s	size of an output tuple
mem	memory used
t_{wait}^{j}	time waiting since last tuple from the jth input

it is important to monitor its memory requirements mem, along with the size s of intermediate results produced, especially in the case when these have to be kept in main memory or in secondary storage, as such resources are not always abundant.

(4) Connections with other operators and data stores: as well as obtaining basic measurements, characteristics of the execution of the part of the query plan below the relevant operator can be inferred, such as the delivery rate of data sources, and in a distributed setting, potential points of network failure, by monitoring the time the operator waits for its inputs to deliver data t_{wait}^{j}.

However, more useful and easily exploited monitoring information is aggregate statistics, e.g., averages, sums, counts, minimums and maximums. Aggregates can be taken in two ways. In one approach, a window is assumed and only the measurements that belong to that window are used for computing the aggregate. Windows can be either overlapping or disjoint, and their widths can be defined in either time units or the number of most recent tuples. In the second approach, the aggregate is computed over all the values seen. For each of the metrics in Table 1, additional information can be derived by performing aggregate functions on them. For example, the average number of result tuples $avg(n)$ over a period of time gives the output rate for that period; the sum of the sizes of each output tuple $sum(s)$ equals the size of that intermediate result; and the minimum time waiting for new tuples from a remote data source $min(t_{wait}^{j})$ can provide an upper bound on the data delivery rate.

Orthogonally to the nature of the measurements, there are numerous potential policies with regard to the frequency of monitoring. Some metrics need to be computed only once during the lifetime of a particular instance of a physical operator (e.g., the time elapsed since the operator's instantiation).

Table 2
The signatures of the physical operators examined. Each operator except scan has either one or two child operators as input.

Name	**Signature**
sequential scan	*seq_scan(table name, predicate)*
hash join	*hash_join(left input, right input, predicate)*
project	*project(input, list of fields)*
unnest	*unnest(input, collection attribute, new field)*
operation call	*operation_call(input, parameters, predicate)*
exchange	*exchange(input, list of consumers, list of producers, data distribution policy)*

Other information is inferred from observing each of the tuples that comprise the operator's input separately, or just some of them (e.g., by sampling).

2.2 Operator-specific information

In Section 2.1, the information collected was generic to all operators and independent of their role in the query plan. However, monitoring at a finer level of granularity may require specific data from distinct operator instances, according to their functionality. By drawing such distinctions, the set of measurements in Table 1 can be further extended. An important detail is that operator-specific monitoring cannot be performed using the two alternative approaches to monitoring, i.e., dedicated monitoring operators or centralised components. Also, note that, as the functionality of different operators is standardised, monitoring the inner basic functions of each operator is still generic and implementation-independent. Such monitoring can be crucial for understanding in depth implementation specific properties like execution time. For instance, we may experience significant variances in the performance of a hash join that is evaluated completely in main memory due to the existence of skew in the sizes of the buckets in the hash table. If the operator is considered to be a black box, such a cause of performance degradation is harder to identify.

Operator-specific monitoring can be applied to any kind of operator. As the complete set of operators from the database literature cannot be presented for brevity reasons, therefore a representative set of physical operators is chosen to demonstrate this approach as shown in Table 2. These operators are sufficient for evaluating SQL and OQL queries of the *Select-From-Where* form in a parallel or distributed environment. *Operation_call* (25) is used for method

Table 3
Measurements for operators that evaluate predicates.

Symbol	**Description**
n_{cond}	number of conditions evaluated per predicate
t_{pred}	time to evaluate a predicate

Table 4
Measurements for operators that touch the store.

Symbol	**Description**
t_{conn}	time to connect to source
n_{pages}	number of pages read
t_{page}	time to read a page
t_{map}	time to map store format into evaluation format

invocation, i.e., it encapsulates a call to a user-defined function.

An example of operator functionality that is not present in all operators is the predicate evaluation (see Table 2). A predicate consists of one or more conditions. Table 3 summarises monitoring information with regard to the evaluation of predicates.

Operators that touch the store include scans and some joins in object-oriented environments. Because the store format is usually different from the tuple format required by the query processor, a mapping between the two formats needs to take place. Monitoring information that is relevant to this kind of operators is shown in Table 4.

A hash join is executed in two phases. In the first phase, the left input is consumed and partitioned into buckets by hashing on the join attribute of each tuple in it. In the second phase, the same hash function is used to hash the tuples in the right input. The tuples of the right input are concatenated with the corresponding tuples of the left input by probing the hash table. Subsequently, the predicate is applied over the resulting tuple. The optimiser needs to ensure that the smallest input is placed as the left input. Table 5 presents metrics that are particular to hash joins.

The unnest operator takes as input a tuple with a n-valued attribute (or relationship), and produces n single-valued tuples. The cardinality of the collection attribute or relationship $Card_{col}$ can be monitored (Table 6).

The exchange operator encapsulates parallelism in multi-node environments. It performs two functions concurrently. It packs tuples into buffers and sends these buffers to consumer processors, while receiving packed tuples from buffers

Table 5
Hash-Join-specific measurements.

Symbol	**Description**
s_i	size of a tuple in the ith input
S_i	size of the ith bucket
N_i	cardinality of the ith bucket
M_i	number of tuples in the right input that correspond to the ith bucket
t_{hash}	time to hash a tuple
t_{conc}	time to concatenate two tuples

Table 6
Unnest-specific measurements.

Symbol	**Description**
$Card_{col}$	cardinality of multi-valued attribute

Table 7
Exchange-specific measurements.

Symbol	**Description**
s_i	size of input tuple
$n_{buffers\_sent\_i}$	number of buffers sent to the ith consumer
$n_{buffers\_received\_i}$	number of buffers received from the ith producer
t_{pack}	time to pack a tuple
t_{unpack}	time to unpack a tuple

sent by producers and unpacking them. The monitoring information for exchanges is given in Table 7.

From the above, it is evident that operator-specific measurements for a specific operator are defined solely on the basis of the distinctive functions that this operator performs. This ensures that the measurements are common in any of its implementations, and provides the criterion for defining the measurements of operators not included in Table 2.

3 Enabling adaptations through local operator monitoring

Traditionally, database systems use optimisers that rank candidate query plans on their predicted cost and, typically, select a plan on the basis of its low predicted cost. If the cost of the selected plan is substantially different from that predicted by the cost model, this may indicate that the chosen plan is not in fact the most suitable. Thus there needs to be an association between the information collected during monitoring and the cost model for the algebra. The cost metrics can be indirect (e.g., size of intermediate results), or direct (e.g., execution time). It is often the case that not only the complete query plan, but also the operators that comprise it can be annotated with performance predictions. Monitoring the cost of the operators can thus inform the calibration of the cost model used in estimation based on a post-mortem analysis. However, identifying erroneous estimates that refer to the final state of the operator at runtime, which is a monitoring task directly related to dynamic query execution, may be non-trivial. To this end, the monitor mechanism should be enhanced (i) with the capability to predict the final cost of query plan, or subplan, based on monitoring information that has become available up to that point; and/or (ii) with the capability to identify operation states that will prevent the system from reaching the expected performance.

In this section, the monitoring framework is applied to the operators in Table 2, which include some of the main physical operators evaluated by both parallel and non-parallel query processors. More specifically, it is verified that a deviation from initial expectations can be not simply detected, but also predicted on-the-fly. It is first examined if this can be achieved through *local* monitoring, i.e., without passing monitoring information between operators, then, in Section 4, this constraint is relaxed. The reasons why one would choose to perform local monitoring are threefold: firstly, one might not want any extra communication overhead regardless of the potential benefits; secondly, the query plan could be executed using blocking operators or materialisation points, which means that, effectively, only one operator is active at any time; and thirdly, initial estimates may only be available for particular operators or particular properties of operators, as is commonly the case for heuristic-based optimisation.

Regarding the predictions, this work does not seek to propose accurate formulas for all the possible cases, implementations, system configurations, value distributions, etc, but rather to demonstrate that such a generic monitoring approach is suitable as a basis for prediction mechanisms. For this reason, the signatures of the prediction formulas are more important than the formulas themselves, as they depict more explicitly the monitoring information required to predict whether there will finally be a deviation from the expected performance or not. It is important to notice that this section is complemented by Section 6. There, it is demonstrated how the monitoring framework presented here can be applied to other adaptive query processing techniques and support different approaches to feedback assessment and response, some of which may not use prediction mechanisms at all.

Table 8
Symbols denoting additional operator properties.

Symbol	**Description**
σ	monitored selectivity
S	monitored size of result set
S^j_{inp}	monitored size of the jth input
T	monitored completion time
$\widehat{\sigma}$	selectivity as known at compile time
$\widehat{S}$	size of result set as known at compile time
$\widehat{S^j_{inp}}$	size of the jth input as known at compile time
$\widehat{T}$	completion time as known at compile time

The cost of operators is estimated according to the detailed cost model described in in time units (23). Here, the focus will be on three aspects of operator execution: the selectivity σ, as it determines the workload for the remainder of the query plan and is hard to predict accurately at compile time when no statistics are available; the size of the result S; and the completion time T, which defines the operator's cost.

In the rest of the paper the following additional notations are used: For each property x being monitored at runtime, $\widehat{x}$ is its static value, either known or estimated at compile time. Each operator is annotated at compile time with expected selectivity $\widehat{\sigma}$, result size $\widehat{S}$, input cardinality $\widehat{n^j_{inp}}$, input size $\widehat{S^j_{inp}}$, and time cost $\widehat{T}$. Table 8 summarises the additional notations.

3.1 Detecting deviations

Spotting deviations from the expected selectivity $\widehat{\sigma}$, result size $\widehat{S}$ and completion time $\widehat{T}$ is supported by the framework in a straightforward manner. From Table 1 we have:

$$\sigma = \begin{cases} \frac{n}{n^1_{inp} \cdot n^2_{inp}} \; for\ binary\ operators \\ \frac{n}{n^1_{inp}} \; otherwise \end{cases} \tag{1}$$

$$S = sum(s) \tag{2}$$

$$T = \begin{cases} sum(t_{tuple})\ or \\ t_{real} \end{cases}$$

After the operator has finished its execution, these values can then be compared against the initials estimates, i.e., $\widehat{\sigma}$, $\widehat{S}$ and $\widehat{T}$, respectively, in order to assess their accuracy.

3.2 Predicting Deviations

If the overall goal is to predict, rather than simply detect deviations, the monitoring framework should provide the necessary input to the prediction mechanism. Table 9 gives examples of prediction formulas that use the monitoring information and can be applied for that purpose. The focus here, as explained earlier, is on the nature of parameters used, rather than on the validity of the formulas; ascertaining the latter is out of the scope of this paper.

The prediction formulas about the final output size belong to two categories: firstly, when the operator does not change the size of the tuple (i.e., the average size of the input tuples is equal to the average output size) and the initial estimate of the input size is correct; and, secondly, when the size does change or the initial estimate is inaccurate. For the final time cost, we have considered three approaches: firstly, to decompose the operator function into subfunctions, such as those in the cost model used (if this is possible), and to use cost information about these subfunctions obtained up to that point, which implies the most detailed measurements; secondly, to build the prediction on the cost of the operator up to that point assuming that the elapsed time is proportional to the number of input tuples, which, intuitively, cannot perform well when system parameters change; and thirdly, to base the prediction on the cost of the last tuple (or of the n last tuples) processed, which, again intuitively, can adapt better to load fluctuations, but may be unduly affected by temporary load changes.

3.2.1 Monitoring Sequential Scans

Based on the measured cardinalities of the input and output at a given point in the execution, the final selectivity can be estimated, e.g., as in Table 9. A new estimate for the final cardinality of the result can be obtained by multiplying the monitored selectivity by the known cardinality of the stored extent $\widehat{n^1_{inp}}$. The total size of result can be predicted in both ways mentioned in the previous paragraph. For the estimation of the total execution time, all three ways considered in the previous paragraph can be applied. In the first one, which requires the identification of simpler operator subfunctions, we can follow the approach of (23), where the cost can be divided into the cost for transforming the format of the tuples (if necessary), evaluating the predicates, and reading the pages $T = t_{page} \cdot n_{pages} + (t_{map} + t_{pred}) \cdot n^1_{inp}$. All these parameters can be monitored as shown in Tables 3 and 4. Different implementations are expected to vary significantly as to their cost, and the contribution of each of

Operator	**Selectivity** σ	**Result Size** S	**Completion Time** T
seq. scan	$\sigma(n, n^1_{inp}) = \frac{n}{n^1_{inp}}$	$S(\sigma, \widehat{S^1_{inp}}) = \sigma \cdot \widehat{S^1_{inp}}$, or $S(\sigma, \widehat{n^1_{inp}}, s) = \sigma \cdot \widehat{n^1_{inp}} \cdot avg(s)$	$T(t_{map}, t_{pred}, t_{page}, \widehat{n^1_{inp}}, \widehat{n_{pages}}) = (avg(t_{map}) + avg(t_{pred})) \cdot \widehat{n^1_{inp}} + avg(t_{page}) \cdot \widehat{n_{pages}}$, or $T(t_{tuple}, \widehat{n^1_{inp}}) = avg(t_{tuple}) \cdot \widehat{n^1_{inp}}$, or $T(t_{real}, t_{lasttuple}, n^1_{inp}, \widehat{n^1_{inp}}) = t_{real} + t_{lasttuple} \cdot (\widehat{n^1_{inp}} - n^1_{inp})$
hash join	$\sigma(n, n^1_{inp}, n^2_{inp}) = \frac{n}{n^1_{inp} \cdot n^2_{inp}}$	$S(\sigma, \widehat{n^1_{inp}}, \widehat{n^2_{inp}}, \widehat{S^1_{inp}}, \widehat{S^2_{inp}}) = \sigma \cdot \widehat{n^1_{inp}} \cdot \widehat{n^2_{inp}} \cdot (\widehat{S^1_{inp}} + \widehat{S^2_{inp}})$, or $S(\sigma, s, \widehat{n^1_{inp}}, \widehat{n^2_{inp}}) = \sigma \cdot avg(s) \cdot \widehat{n^1_{inp}} \cdot \widehat{n^2_{inp}}$	$T(t_{hash}, t_{conc}, t_{pred}, n_{pairs}, \widehat{n^1_{inp}}, \widehat{n^2_{inp}}) = avg(t_{hash}) \cdot (\widehat{n^1_{inp}} + \widehat{n^2_{inp}}) + (avg(t_{pred}) + avg(t_{conc})) \cdot n_{pairs}$, or $T(t_{real}, t_{lasttuple}, n^2_{inp}, \widehat{n^2_{inp}}) = t_{real} + t_{lasttuple} \cdot (\widehat{n^2_{inp}} - n^2_{inp})$
project / op. call	$\sigma() = 1$	$S(\widehat{n^1_{inp}}, s) = \widehat{n^1_{inp}} \cdot avg(s)$	$T(t_{tuple}, \widehat{n^1_{inp}}) = avg(t_{tuple}) \cdot \widehat{n^1_i}$, or $T(t_{real}, t_{lasttuple}, n^1_{inp}, \widehat{n^1_{inp}}) = t_{real} + t_{lasttuple} \cdot (\widehat{n^1_{inp}} - n^1_{inp})$
unnest	$\sigma(n, n^1_{inp}) = \frac{n}{n^1_{inp}}$	$S(\widehat{n^1_{inp}}, s) = \widehat{n^1_{inp}} \cdot avg(s)$	$T(t_{tuple}, \widehat{n^1_{inp}}) = avg(t_{tuple}) \cdot \widehat{n^1_{inp}}$, or $T(t_{real}, t_{lasttuple}, n^1_i, \widehat{n^1_{inp}}) = t_{real} + t_{lasttuple} \cdot (\widehat{n^1_{inp}} - n^1_{inp})$

Table 9. Prediction formulas exemplifying how the monitoring information can support predictions in AQP.

the three common subcosts to the total execution time. Thus, monitoring at a finer level may be important in order to identify and quantify such differences.

3.2.2 Monitoring Hash Joins

The approach for predicting the final values of the selectivity and the size of the output of a hash join is similar to the one used for scans. The main difference between scans and joins is that the cardinalities of the inputs are more likely to be estimated rather than measured and building estimates upon previous estimates may result in compound errors (14). The total execution time of a hash join consists of the time to hash the tuples for both inputs, the time to concatenate all the relevant pairs of tuples and the time to evaluate the join predicate, i.e., $T = t_{hash} \cdot (n_{inp}^1 + n_{inp}^2) + (t_{pred} + t_{conc}) \cdot n_{pairs}$. The initial estimate of the cost of a hash join at compile time uses a constant value for the time required to hash a tuple. This constant can be monitored (Table 5) and thus can be corrected in case it is not accurate. Another constant is used for the time to concatenate two tuples, which is also error prone. The number of pairs of tuples concatenated is more difficult to estimate. The optimiser can make a simple assumption that there is a uniform distribution of tuples across the hash table buckets. A more realistic formula that captures the potential skew in the partition of tuples into buckets and can be used to estimate the number of pairs is $n_{pairs} = (\sum_{i=1}^{m} N_i \cdot M_i) \cdot \frac{\widehat{n_{inp}^2}}{n_{inp}^2}$, where m is the total number of buckets, and the remaining arguments are defined in Table 5. Another option, if the left input has already been consumed, is to use the time elapsed along with the time taken to evaluate the last tuple, as shown in the second formula in the relevant field of Table 9.

3.2.3 Monitoring Projects, Unnests and Op. Calls

For projections and operation calls, the cardinality of the output is the cardinality of the input, as their selectivity is always equal to 1. The size and time prediction formulas resemble those for scan (Table 9). Unnests differ in that they may have a selectivity greater than 1.

3.2.4 Monitoring Exchanges

The time cost of an instance of exchange is the sum of the costs to receive packed tuples from remote nodes, unpack them, pack tuples into a buffer, and send the packed tuples to other nodes. The communication cost is the dominant cost.

The cardinality, the size of the output, and the time cost of the operator can be monitored. However, no more accurate estimates for the number of tuples to be produced can be made at runtime without communication, other than the estimations made by the optimiser at query compile time. This is because this metric depends on the number of buffers that other instances of exchange send to that node, and in order to get this information, data transmission is required.

On-the-fly updating of the predictions for the output size and the time cost can occur in a limited range of situations. A better estimate of the size can be made if the observed average tuple size is different than the estimated one, but again the information about the total number of result tuples is missing.

More accurate estimates of the time cost are produced by adding together more accurate estimates of the component costs. The time cost to transmit data depends on the input cardinality, the average size of a tuple, and the network speed between the two nodes involved. It also depends on system parameters that are not expected to vary for a given system, such as the size of a buffer and the space overhead for each buffer. From the above three variables, more accurate values can be obtained for the average tuple size. If this size remains the same, the system cannot detect deviations from expected cost caused by fluctuations in the network bandwidth, or from the expected number of incoming tuples. Changes in the expected cardinality of the input and the output are tracked, but cannot be predicted on the fly.

3.3 Discussion

Section 2 discussed what may be useful to monitor. This section has indicated how the approach for monitoring individual query operators can be used for detecting and predicting deviations from expected performance presenting an example that explained how monitoring can be useful for adaptation when no monitoring data are transmitted. Notice, however, that other useful monitoring tasks, such as cost model refinement, were not examined as the focus is on applications that may modify the query execution on-the-fly. Under some assumptions, predictions on the performance of the system for the remainder of the query can be made on the fly. The only exception to this is for parallel query plans that include exchanges. In that case, predictions can only be made for a limited range of cases and passing data between operators is necessary for improved estimates. Other examples of how the monitoring framework presented can be used to support adaptivity are given in Section 6.

The prediction formulas for scans and unnests that only use notation from Tables 1 and 8 can be generalised for any operator with selectivity different from 1. The formulas for projections can be generalised for any operator with

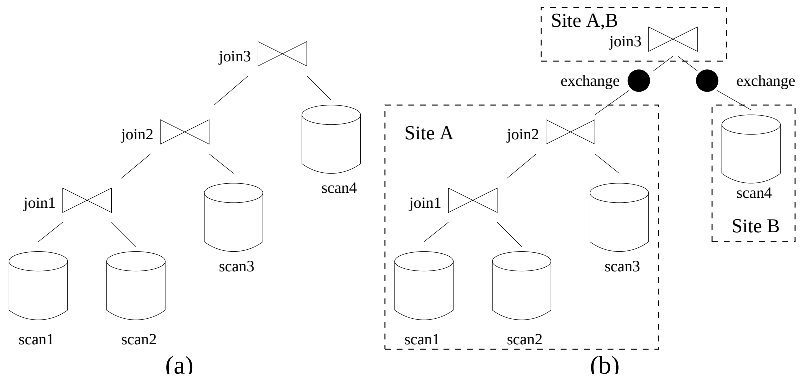

Fig. 1. Example query plans executed over (a) a single machine, and (b) two machines

selectivity equal to 1. The formulas for hash joins that do not use notation from Table 5 can be applied to any binary operator. In general, the formulas used here are simple and may not be appropriate in all usage scenarios. It is not the aim of this paper to explore their validity over more diverse usage scenarios. The role of such formulas in the monitoring task is to provide feedback for adaptive query processors. Although the performance criteria were defined to be the selectivity of operators, the size of (intermediate) results, and the time cost, there is no fundamental reason why this set cannot be extended and tailored to different system characteristics.

4 Propagating monitoring information

Section 3 examined the case of monitoring without communication overhead. This section shows how relaxing this constraint can enhance monitoring precision. Firstly, the case in which data is not transmitted to remote nodes is discussed, then the case in which monitoring data is shared among different nodes. In the first case, the communication overhead can remain low, as the information does not have to be conveyed through the network. Actually, it may not need be passed between operators physically at all, but simply recorded for access by later operators.

4.1 Sharing information among different operations in a node

When lower operators in the query plan propagate more accurate estimates to operators that lie above them, estimates for the latter become more accurate. The formulas in Table 9 allow on-the-fly predictions of the final number of tuples, the final size of the result and the final time cost. These formulas depend on initial estimates of the input cardinality ($\widehat{n^1_{inp}}$ and/or $\widehat{n^2_{inp}}$) and size ($\widehat{S^1_{inp}}$ and/or $\widehat{S^2_{inp}}$). Monitoring allows more accurate estimates of these properties. The input size and the input cardinality of an operator are the output size and the output cardinality of its children, respectively. All physical operators, except exchange, are able to produce more accurate predictions for these two metrics. This function operates in a recursive way that results in the propagation of better estimates from the lowermost to the topmost operator provided that an exchange operator does not break that chain.

Consider the query plan in Fig. 1(a). For each join, the expected cardinalities of the inputs are computed at compile time. Even if the selectivities of the three joins are estimated with the same accuracy, the estimate for the output of the third join can be much worse than the estimate for the second join and even worse than for the first one. (14) explains how propagation of errors affects the quality of these estimates. All operators can continuously update their expected output cardinalities and selectivities if monitoring is in place. The propagation of these measurements results in the third join having an up-to-date estimate for its inputs. These inputs also have up-to-date estimates for their inputs and so on. In that way, the effect of potentially inaccurate initial estimates can be ameliorated.

The type of the formulas of Table 9 remain the same. However, for each operator the values $\widehat{n^j_{inp}}$ and $\widehat{S^j_{inp}}$ are replaced with the relevant predictions of its child.

4.2 Sharing information among different nodes

In the previous example, assume now that the fourth scan is placed on another node, and that the third join is evaluated through partitioned parallelism on both sites. In the operator model of parallelism, tuples are exchanged between nodes through the exchange operator (Fig. 1(b)). If no communication across sites is permitted for monitoring, exchanges cannot give up-to-date estimates. In this case, the third join can only use the initial estimates computed at query compile time. However, if there is no zero-communication constraint, the monitored information can be transmitted to and across exchanges. In this way, each instance of exchange can predict on the fly the total number of buffers and the number of tuples that will be sent to each consumer. New estimates of the output cardinalities can be produced by gathering this information from all the exchanges.

Allowing monitored information to be transmitted over the network has additional benefits. The relative workload of the nodes can be monitored by tracking and comparing the number of tuples each instance of an operator receives. Moreover, the connection speed between two nodes can be monitored by recording the time when a buffer is sent from a node and the time it arrives at its destination. Finally, the relative load between nodes can be monitored by tracking and comparing the average times to process a tuple on different sites. Hence, communication overhead can be traded for such benefits.

This approach to propagating the monitoring information through the query plan allows for adaptive schemes where operators adapt autonomously (e.g., (31)) as well as approaches that co-ordinate the query re-optimisation centrally (e.g., (3)).

5 Evaluation

The presentation of the experimental results in this section serves two purposes. Firstly, to provide insights into how large the overhead of monitoring and predicting can be, and, secondly, to assess the accuracy of the predictions based on monitoring. The data used in the experiments are from the OO7 benchmark (5). The measurements are taken on a dedicated PC with 1.13GHz AMD Athlon CPU and 512 MB memory (of which 330 - 370 MB were available at the time of the measurements), running Redhat Linux 7.1. The query engine used is part of the Polar* Grid-enabled distributed query processor (25). The operators are implemented in C++ according to the iterator model (10) and following the standard algorithms as these appear in the literature, and all are single-pass, i.e., all intermediate data sets are stored in main memory, although the data starts off on disk. The granularity of the system's timer is one microsecond.

5.1 Overhead of Monitoring

The measurements fall in three categories: firstly, those that involve counters (e.g., cardinalities of input, output and hash table buckets); secondly, those that require timings, i.e., two timestamps are taken and their difference is computed (e.g., time to evaluate a tuple, time to change the tuple format from the storage format to the evaluator format), and thirdly, those that compute the size of a tuple. The size of the tuple is not statically known in two cases. Firstly, when the tuple has one or more tuple fields with string type of undefined length; and, secondly, when there is a collection attribute of undefined collection size. Measuring the size of a collection requires a counter. Measuring the size of a string of characters involves identifying the tuple fields in the tuple that are string-valued and computing the length of each.

Inserting a counter in an operator has a very small overhead, measured at

Table 10
The operators used in the experiments for monitoring overheads.

Operator	**Characteristics**
Scan A	average size of tuples is 155bytes
Scan B	average size of tuples is 727bytes
Scan C	average size of tuples is 2Kbytes
Scan D	average size of tuples is 20Kbytes
Hash-Join A	1 tuple per hash table bucket
Hash-Join B	10 tuples per hash table bucket
Hash-Join C	20 tuples per hash table bucket
Hash-Join D	200 tuples per hash table bucket
Project	project one tuple field out of 10
Unnest	fan-out is set to 3, average size of initial tuples is 155bytes

Table 11
The overhead of taking measurements compared to the cost of the operators for each tuple processed.

Operator	**time (in $\mu secs$)**	**counter (%)**	**timing (%)**	**100-byte string (%)**
Scan A	16.82	0.18	6.60	70.63
Scan B	25.50	0.12	4.35	46.60
Scan C	48.81	0.06	2.27	24.34
Scan D	350.57	0.01	0.32	3.39
Hash-Join A	8.22	0.36	13.50	144.52
Hash-Join B	13.02	0.23	8.52	91.22
Hash-Join C	16.25	0.18	6.83	73.11
Hash-Join D	62.86	0.05	1.77	18.90
Project	0.89	3.39	125.27	1340.71
Unnest	10.16	0.30	10.92	116.88

0.03 $\mu secs$. The overhead of measuring timings is of the order of microseconds (1.11 $\mu secs$). The time cost of measuring the size of a string of characters depends on the size of the string. For small strings, the overhead is small but for larger strings it can become several milliseconds (e.g., for a 1MByte string this takes 0.0044 secs).

The operators that were used in the experiments for monitoring overheads are shown in Table 10. All the joins are on a key/foreign key condition. Table 11 depicts more clearly the magnitude and relative importance of the overheads, as it shows the percentage increase in the cost of evaluating a tuple due to monitoring. For each of the operators in Table 10, the time cost is given (2nd column in Table 11). The last three columns show the increase in the cost when a counter, a timing and a character counter for a 100-byte string are applied to each tuple processed, respectively. As expected, the relative overheads are higher for computationally-inexpensive operators, like project, and significantly lower for the computationally-expensive ones, like hash join. The overhead of a counter is negligible for all the operators. Placing two timestamps is more costly than projecting an attribute, but the percentage overhead is relatively low for other operators (between 0.32% and 13.5%). Measuring the size of a string has essentially no cost if the string is a few bytes long. If the length is 100 characters or more, the performance may degrade significantly. For instance, it may increase the cost of a hash join by up to 144%, when the size is monitored for each tuple processed by the operator. If the size is computed for one tuple in ten, the increase is only 14.4%, and if the frequency is 5% (one in twenty tuples is monitored), the increase falls to 7.2%. However, it is not usually necessary to compute this in a database setting, as often, the length of a string is stored explicitly. The values in Table 11 can inform the choice of monitoring frequency by indicating broadly the overhead that can be anticipated.

5.2 Overhead of Predictions

Here, only the scan operator is analysed, but a similar approach can be followed for the remaining operators. The results for all operators are shown in Table 12.

It is assumed that the system holds information about the size and cardinality of the stored collections. The selectivity of an operator is given by $\sigma = \frac{n}{n^1_{inp}}$, where n and n^1_{inp} are the monitored cardinality of the output and the input up to that point of execution, respectively (Section 3.2.1). The output cardinality is predicted by multiplying the monitored selectivity with the known input cardinality. It requires two counters that are updated for each tuple and the evaluation of one formula. The formula may be evaluated at various frequencies, but it is processed in time significantly less than a microsecond. The cost of the two counters is of the order of nanoseconds (0.03 $\mu secs$ each). So, the overhead of predicting the final number of tuples produced is some fraction of a microsecond. If the output tuples do not contain strings with variable length, the final output size is predicted by multiplying the monitored selectivity with the known size of the stored collection, and the overhead of this prediction

Table 12
The percentage increase in the operator cost when predictions are made.

Operator	**Overhead computing output cardinality (%)**	**Overhead computing operator time (%)**
Scan A	0.36	$(6.6/\mathit{freq} + 0.18)$
Scan B	0.24	$(4.35/\mathit{freq} + 0.12)$
Scan C	0.12	$(2.27/\mathit{freq} + 0.06)$
Scan D	0.02	$(0.32/\mathit{freq} + 0.01)$
Hash-Join A	0.72	$(13.5/\mathit{freq} + 0.36)$
Hash-Join B	0.46	$(8.52/\mathit{freq} + 0.23)$
Hash-Join C	0.36	$(6.83/\mathit{freq} + 0.18)$
Hash-Join D	0.1	$(1.77/\mathit{freq} + 0.05)$
Project	6.78	$(125.27/\mathit{freq} + 3.39)$
Unnest	0.60	$(10.92/\mathit{freq} + 0.30)$

is the overhead incurred by two counters as well. If the tuple produced does contain strings of undefined length, the total size is given by

$$S = \sigma \cdot \widehat{n^1_{inp}} \cdot avg(s)$$

where $avg(s) = \frac{sum(s) \cdot freq}{n^1_{inp}}$ and $freq$ specifies every how many tuples the tuple size s is monitored. The cost of making these predictions is essentially dominated by the cost of measuring the length of the strings.

Predicting the total time for completion of the operator involves one timing t_{tuple} being captured for each monitored tuple as follows:

$$T_{total} = avg(t_{tuple}) \cdot \widehat{n^1_{inp}}$$

The average overhead is 1.11 $\mu secs$ for each monitored tuple, which is the cost of a single timing, plus the cost of updating a counter.

Table 12 shows the relative overhead of making predictions. As the prediction of the output size depends on the size of the variable length strings (if any) and is not generic, it is not shown in the table. If there are no collection attributes or variable-length strings, then the cost is the same as the cost to compute the output cardinality.

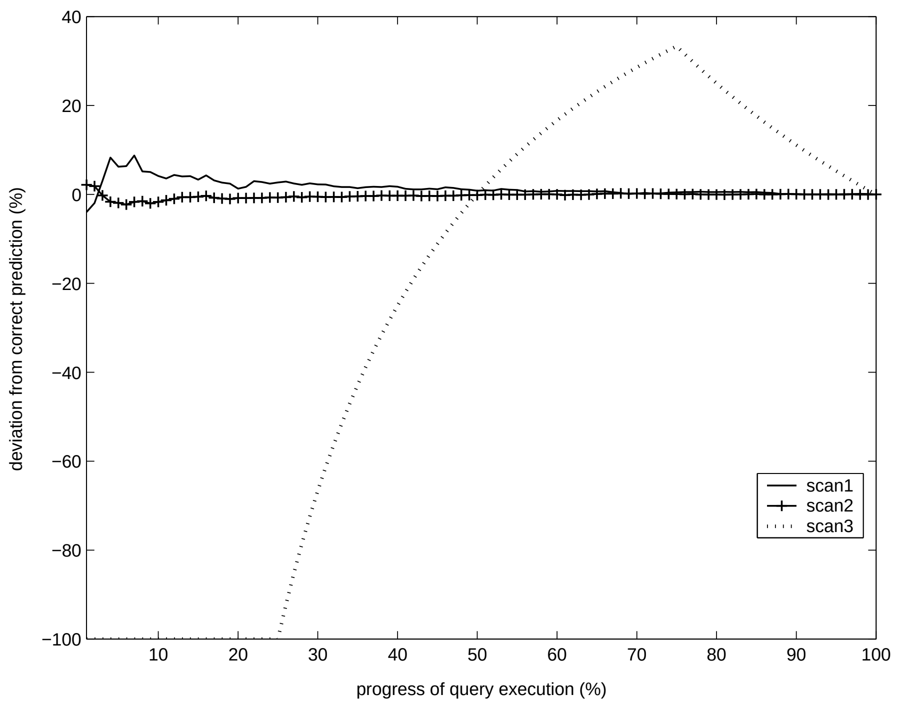

Fig. 2. The accuracy of the predictions for the output cardinality of the three scans at different stages of the operator execution.

5.3 Accuracy of predictions

The formulas introduced in Section 3 are rather straightforward and may be expected to give better results when the system is uniform in terms of load, attribute value distribution, operator workload, etc. However, it is interesting to examine how large the deviations are when the formulas are applied to skewed data. Since all operators require initial estimates for their input cardinality, an error in that cardinality compromises the accuracy at exactly the same magnitude. Consequently, it is important that an operator not only is able to make accurate predictions about the cardinality of its result set, but also that it is able to pass that information on to its parent operator in the query tree, as described in Section 4.

Consider three scans. The first, scan1, has selectivity 10% and the tuples that satisfy the scan condition are spread in a uniform manner across its extent. The second, scan2, also has a uniform distribution, but the selectivity is 50%. The third scan, scan3, has a selectivity of 50%, and is satisfied by all but the first 25% and the last 25% of the tuples. Figure 2 shows how accurate the predictions for the output cardinality are at each stage in the process of query execution. The formula used assumes that the final selectivity of the predicate is the same as the monitored selectivity at that point. If the tuples that satisfy the predicate are distributed across the dataset in a uniform manner (e.g., scan1 and scan2) the accuracy is very high and not dependent

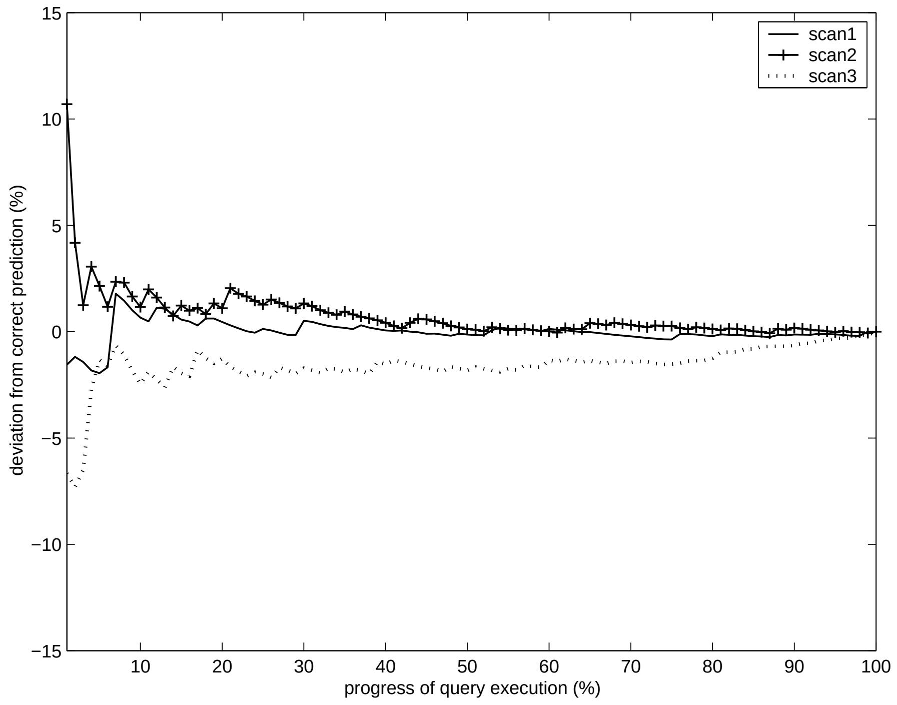

Fig. 3. The accuracy of the predictions for the response time of the three scans.

on the selectivity. However, for skewed distributions with unfavourable shapes (e.g., scan3), the predictions can be erratic over the course of execution. If the load of the system does not vary during execution and every tuple is monitored, the operator response time can be accurately predicted from the very early stages for all three scans (Figure 3). Lower monitoring frequencies result in worse accuracy, especially if the load varies.

5.4 General remarks on the evaluation

There are several lessons to be learned from the evaluation of the overheads related to monitoring:

(1) In our approach, there are three types of monitored information: counters, timings, and sizes of variable-length strings. The overhead of these three types is not dependent on the type of the query operator. The costs of counting and of computing a time interval are constant for a given system, whereas the cost of measuring the size of a string depends on its size.
(2) The cost of a counter is negligible for all the operators examined. However, this is not true for timings and string computations.
(3) The cost of computing the output cardinality is lower than 1% for all operators examined except project, for which it is 6.78%. So, it can be regarded as low. Additionally, the relative cost of predicting the final response time is lower than 13.5% for all operators except project, even

if the time cost of each tuple is measured separately. If the time cost is measured at a frequency lower than 10% (i.e., one in ten tuples is timed), the cost becomes lower than 1.5% for these operators.

In general, the relative overhead incurred by monitoring remains low if the monitoring frequency of inexpensive operators (and the monitoring frequency of all operators when the environment is stable) remains low. Notice that in multi-pass implementations of operator algorithms, where the data cannot fit entirely in main memory, the average cost of the operator is expected to be significantly higher, whereas the cost of monitoring is expected to remain the same. Consequently, in such systems the contribution of the monitoring cost to the total execution time is envisaged to be even smaller. Thus, the results presented here with respect to the proportional overhead of monitoring approximate the worst-case scenario, as other query processors are expected to behave either similarly or worse than the query processor used, in terms of the monitoring overhead. For accurate predictions, a good knowledge of the input sizes and cardinalities is always required, which means that the children also need to be able to make good predictions and pass on relevant information to their parent. If the load of the system does not vary, the total response time can be predicted accurately from the early stages of execution. When it varies, the predictions can still converge, but they require higher monitoring frequencies. Skewed distributions impose significant errors but, even in such cases, predictions can be better than direct usage of estimates produced at compile time.

6 Related Work

This work can be related to numerous activities in the area of performance analysis for databases and software systems, database cost models and performance prediction. However, the most relevant work is in the area of adaptive (or dynamic) query processing (AQP). Adaptive query engines receive information from their environment and determine their behaviour according to that information in an iterative manner (12). The most dynamic are those that capture specific aspects of the query processing, evaluate this feedback and react accordingly, during the execution of a single query. According to the feedback they collect from the query execution, they can be classified in three broad categories.

The adaptive systems that monitor the rate at which they receive their input belong to the first category. A typical example is the XJoin (28), a variant of pipelined hash joins that hides delays in the arrival of the input tuples by performing other operations when the inputs are blocked. In our approach, the input tuple rate and the time waiting since the last tuple was processed can be monitored for each operator. Consequently, it can be inferred whether an input is blocked by using a threshold. Ginga (20), Query Scrambling (30), and Bouganim *et al* (3) also deal with the problem of experiencing delays in the delivery of the first tuples from a remote source. (3) proposed an approach that generalised Query Scrambling to adapt not only to blocked connections, but to any changes in the data delivery rates as well. To monitor the delivery rates, they employ a new component, whereas in the approach proposed in this paper this could be easily achieved within the operator. Information about the data delivery rates can trigger adaptation also in the context of Rivers (1), a proposal for parallel I/O intensive applications, which monitors the bandwidth between data producers (e.g., disks) and consumers (e.g. scan operators).

Another group comprises systems that focus on the workload and the productivity of operators measured in tuples. For example, Eddies (2), a very dynamic technique, encapsulates a multi-join, and dynamically chooses the order of the individual joins for each incoming tuple. The basic routing policy observes the number of tuples received by each join so far, and the number of tuples produced. Both these metrics are covered by the proposed approach, not only for the joins, but for all the operators (Table 1). Also, Flux (24) extends the traditional exchange operator to adapt to fluctuations in resource availability (like resource and memory loads) while executing a query in a pipelining mode. It relies on the on-the-fly selection of simple statistics like the number of tuples processed and the time the operator is active. These systems can be combined together to form even more powerful mechanisms like the TelegraphCQ (6) and (18; 21). In (29), the Dynamic Pipeline Scheduler tries to reduce the initial response time of the query, basing its adaptive behaviour on the number of the tuples consumed so far by the operators and on their selectivities.

More generic systems, in terms of the information they collect from a query plan, fall in the third category. Kabra and DeWitt (16) use a separate monitoring operator for collecting statistics about data on the fly, provided that this is possible in one pass of the input. Such statistics include the cardinality of intermediate results, their average size, and certain histograms. However, it requires the monitoring points to be defined at compile time, it cannot operate in parts of the plan that are executed in a pipelined fashion, and, it cannot capture timings referring to other operators (e.g., the time taken for a lower operator to process a tuple). These limitations, which are essentially limitations of the approach to monitoring in which dedicated operators are employed, do not arise in our approach, since it is based on self-monitoring operators. The Tukwila system (15) integrates adaptive techniques proposed in (16; 30). A special operator is also used to switch to an alternative data source, when the initial source fails. The execution information that the system monitors for active operators is the number of tuples produced so far (to check whether the optimisers estimates were adequately accurate) and the time waiting since the last tuple was received (to identify slow or blocked connections). The Conquest query processing system resembles Tukwila in adopting a triggering approach to respond to runtime changes (19). Characteristics related to query execution that can trigger actions include updates to operator selectivities and sizes of intermediate results. Also, the system monitors the load of the resources. The framework we have presented can infer relative levels of load by comparing the time to process a tuple at different points of the execution. Operator selectivities and sizes are monitored explicitly.

Table 13 summarises the aspects of our monitoring proposal in Table 1 that are used by the AQP systems examined (referring to specific operators). It demonstrates that the proposal is generic enough to support many adaptive systems with different functionalities and requirements. The approach presented integrates and extends existing monitoring approaches with regard to data characteristics and execution cost. In essence, any of the above adaptive techniques can implement its assessment and response strategy on top of our monitoring framework. This cannot be achieved by operators dedicated to statistics collection or new components in the architecture of the query engine, as both these techniques can capture a significantly smaller amount of monitoring information. In contrast, updated information on computational resources (like available memory or new machines becoming available) is an important and complementary factor for deciding about adaptivity that is not covered by our approach, as such information cannot be inferred solely from the query execution.

The overhead of monitoring has not been explicitly considered in the literature above. Information about the overhead is included in LEO (26), which monitors the query plan but only collects information about operator and predicate selectivities, and about the cardinalities of the intermediate results. This additional information is stored so as to enable the adjustment of the query optimizer for the subsequent queries. The overhead is about 5% and has been regarded by the authors as small.

7 Conclusions

So far, adaptive query processors have tended to ignore the monitoring phase at all, or, to use potentially efficient, but *ad hoc*, ways of collecting feedback from the environment and the query plan itself, analysing that feedback and choosing a reaction, all grouped together. This paper argues that these three functions can be studied separately, in order to exploit the benefits of *divide-and-conquer* techniques and to gain generality, substitutability, and reusability. The main contribution of our work is the construction of a general technique for monitoring the execution of query plans, based on self-monitoring query

Table 13
Monitored information that can provide input to existing AQP systems.

Systems	**Operators**	n	n^j_{inp}	t_{real}	t_{tuple}	s	t^j_{wait}
Bouganim *et al*	scan & parent		√	√			√
Conquest	any	√	√		√	√	
Dyn. Pip. Scheduler	join	√	√				
Eddies	join	√	√				
Flux	exchange		√	√			
Ginga	scan & parent		√	√			√
Kabra & DeWitt	any	√				√	
Q. Scrambling	join & scan		√	√			√
River	scan		√	√			√
Tukwila	any	√	√	√		√	√
XJoin	join		√	√			√

operators. Our approach is generic in the sense that it is not dependent on any particular adaptive system or form of adaptation and can support most AQP proposals to date in terms of the monitoring information required. In addition, it is capable of identifying and predicting erroneous initial estimates on the fly. It can be easily implemented as it employs only counters, timestamps and tuple size computations. It can be easily integrated into existing query engines, as it does not require changes in the architecture or in the internal logic of the query optimiser. Moreover, our approach is not centralised and thus fits better to distributed environments with potentially large numbers of nodes. The simplicity does not compromise the comprehensiveness, as many properties of query execution can be captured in a systematic way. Also, it allows for monitoring at different levels of detail in the monitoring information and at different frequencies, and it examines the trade-offs between communication overhead and monitoring quality.

Finally, the monitoring approach was experimentally evaluated. The overheads and the increase in operator cost incurred by monitoring are reasonable enough for the approach to be incorporated in query systems that operate in volatile environments. In addition, the experimental results can provide strong insight into how the frequency and the intensity of monitoring impact on its cost.

Acknowledgements

This work was funded by the EPSRC Distributed Information Management Initiative (through Grant GR/R51797/01), whose support we are pleased to acknowledge. We are also grateful to Jim Smith and Paul Watson, for their valuable feedback.

References

[1] R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, 1999.

[2] R. Avnur and J. Hellerstein. Eddies: continuously adaptive query processing. In *Proc. of ACM SIGMOD 2000*, pages 261–272, 2000.

[3] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. Dynamic query scheduling in data integration systems. In *Proc. of ICDE 2000*, pages 425–434, 2000.

[4] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings of ACM SIGMOD 2002*, pages 263–274. ACM Press, 2002.

[5] M. Carey, D. DeWitt, and J. Naughton. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, 1993.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of 2003 CIDR*, 2003.

[7] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 34–43. ACM Press, 1998.

[8] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing.* Morgan Kaufmann Publishers Inc., 1999.

[9] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of ACM SIGMOD 1990*, pages 102–111, 1990.

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[11] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *Proc. of ACM SIGMOD 1999*, pages 287–298, 1999.

[12] J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing:

Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[13] Y. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[14] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of ACM SIGMOD 1991*, pages 268–277, 1991.

[15] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of ACM SIGMOD 1999*, pages 299–310, 1999.

[16] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of ACM SIGMOD 1998*, pages 106–117, 1998.

[17] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[18] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of ACM SIGMOD 2002*, pages 49–60. ACM Press, 2002.

[19] K. Ng, Z. Wang, R. Muntz, and S. Nittel. Dynamic query re-optimization. In *Proc. of 11th SSDBM*, pages 264–273. IEEE Computer Society, 1999.

[20] H. Paques, L. Liu, and C. Pu. Ginga: a self-adaptive query processing system. In *Proceedings of the 11th ACM CIKM*, pages 655–658, 2002.

[21] V. Raman and J. Hellerstein. Partial results for online query processing. In *Proceedings of ACM SIGMOD 2002*, pages 275–286. ACM Press, 2002.

[22] M. Roth, F. Ozcan, and L. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *Proc. of the 25th VLDB Conference*, pages 599–610. Morgan Kaufmann, 1999.

[23] S. Sampaio, N. W. Paton, J.Smith, and P. Watson. Validated cost models for parallel OQL query processing. In *OOIS*, LNCS, pages 60–75. Springer, 2002.

[24] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of ICDE 2003*, 2003.

[25] J. Smith, A. Gounaris, P. Watson, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou. Distributed query processing on the grid. In *Proceedings of the Third Workshop on Grid Computing, GRID2002*, pages 279–290. Springer, 2002.

[26] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of the 27th VLDB Conference*, pages 19–28. Morgan Kaufmann, 2001.

[27] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. of 29th VLDB Conference*, pages 333–344, 2003.

[28] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[29] T. Urhan and M. Franklin. Dynamic pipeline scheduling for improving interactive query performance. *The VLDB Journal*, pages 501–510, 2001.

[30] T. Urhan, M. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proc. of ACM SIGMOD 1998*, pages 130–141, 1998.

[31] W. Zhang and P. Larson. Dynamic memory adjustment for external mergesort. *The VLDB Journal*, pages 376–385, 1997.