Adaptive Query Processing: A Survey

Anastasios Gounaris, Norman W. Paton, Alvaro A.A. Fernandes, and Rizos Sakellariou

Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK
{gounaris,norm,alvaro,rizos}@cs.man.ac.uk

Abstract. In wide-area database systems, which may be running on unpredictable and volatile environments (such as computational grids), it is difficult to produce efficient database query plans based on information available solely at compile time. A solution to this problem is to exploit information that becomes available at query runtime and adapt the query plan to changing conditions during execution. This paper presents a survey on adaptive query processing techniques, examining the opportunities they offer to modify a plan dynamically and classifying them into categories according to the problem they focus on, their objectives, the nature of feedback they collect from the environment, the frequency at which they can adapt, their implementation environment and which component is responsible for taking the adaptation decisions.

1 Introduction

In a distributed environment, statistical information about the available data sources may be minimal, and the availability or load of physical resources is prone to changes. Consequently, traditional query optimisation can lead to poor performance, especially in long running query evaluations, as the query optimiser may not, at compile time, have the necessary statistics, good selectivity estimates, knowledge of the runtime bindings in the query, or knowledge of the available system resources required to produce an optimal query plan (QP). In addition, traditional optimisers cannot predict the future availability of resources. *Adaptive* (or dynamic) *query processing (AQP)* addresses this, by adapting the QP to changing environmental conditions at runtime.

Current research on adaptive query processing follows two main directions. The first approach responds to changes in the evaluation environment by modifying the execution plan at runtime (e.g. by changing the operators used or the order in which they are evaluated). The other approach involves the development of operators that deal more flexibly with unpredictable conditions and adapt their behaviour by collecting and taking into consideration information that becomes available at query runtime about how query evaluation is proceeding and about changes in the wider execution environment.

A query processing system is defined to be adaptive in [11] if it receives information from its environment and determines its behaviour according to that information in an iterative manner, i.e. there is a feedback loop between the environment and the behaviour of the query processing system. In this survey, we assume a narrower definition, in the sense that we restrict our attention to cases where the feedback loop produces effects during the execution of the query. If the effects of the feedback loop are deferred, then only subsequent queries, rather than the one that is running, can benefit from the adaptive technique employed. This is less appealing for novel data management tasks where systems are processing potentially very long running queries over multiple, semi-autonomous sources on wide-area networks with unpredictable data transfer rates. Broadly speaking, the need for adaptive query processing is emerging as query engines are scaled and federated, so that they must cope with highly unpredictable and volatile environments. If databases are to be integrated into the Grid [8], this need will become even greater.

The remainder of the paper is structured as follows. Section 2 describes the classification criteria and the potential attributes of AQP. Sections 3 and 4 present existing adaptive operators and algorithms respectively, followed by a discussion of general observations in Section 5. Section 6 concludes the paper.

Related Work: [11] presents the most relevant work to this survey. As mentioned above, it examines systems that may produce different query plans at any time in the future, in response to changes in the environment, while the present survey focuses on query processing techniques that can adapt to changing conditions during execution time. Also, [11] uses a very limited set of classification criteria. [14] extends this set by proposing some dimensions over which AQP systems may differ. Other previous surveys on query optimisation, like [10, 12], include techniques that have a dynamic flavour, in the sense that the QP they produce at compile time is not completely fixed, but such techniques are not covered by the definition of adaptivity given earlier.

2 The Scope of the Survey

The query evaluator of a database management system (DBMS) comprises two main sub-components: the query optimiser (QO) and the execution engine. The query optimiser generates the input for the execution engine. After a query is submitted, the optimiser chooses one from many, logically equivalent, transformations of the query in the form of algebraic expressions. These logical query plans are often represented as logical operator trees. The most interesting characteristics of logical plans, with regard to query optimisation, are the location of project and select operators, the operator order and the tree shape.

A physical operator plan is derived from the logical one through the selection of algorithms to implement each of the operators of the logical plan. In order to produce the physical plan, decisions on whether to use indices or not, and whether to employ pipelining instead of blocking operators, are also taken. In general, each logical plan corresponds to many physical plans. Either cost models or heuristics are used to predict the cheapest plan for execution.

In parallel and distributed DBMSs, partitioning and scheduling issues arise during optimisation. A query plan can be divided into different sets of operators, called partitions, in order to harness, to the greatest possible extent, the benefits of parallelism. The query optimiser is responsible for deciding the number, the location, the kind, and how partitions will be scheduled.

Existing AQP techniques differ over many dimensions, as they have impacts of different kinds on the QP, they do not always share common goals and focus, and they are applicable to different environments. Moreover, they may collect different feedback and adapt with different frequency. These dimensions are discussed in the following sections and provide the basis for the survey, which is summarised in Table 1.

2.1 Modifications in the Query Plan

Modifications to the query plan may occur at two levels: logical and physical. Modifications at the logical level (*LL* field in Table 1) may fall into one of the following categories:

Reformulation of the remainder of the query plan, denoted in Table 1 by *rem*, when the AQP techniques can produce a totally different logical subplan for the remainder of the query, introducing new logical operators, changing the operator order or altering the tree shape.

Operator Reordering, denoted in Table 1 by *op_or*, when techniques can only change the order of pre-defined logical operators. The two categories, *rem* and *op_or*, are not mutually disjoint, as the extent of the effects of the techniques in the first category is a superset of those in the second.

No effects, denoted in Table 1 by *no*, when the AQP methods, although they can modify a query during runtime, they do not modify their logical plan.

In this survey, three aspects of modifications to the query plan at the physical level, which may be brought about by AQP techniques, are considered: effects on the physical query plan, re-partitioning and re-scheduling. The impact of the dynamic techniques on the physical query plan (*PP* field in Table 1) is of three kinds:

Usage of adaptive operators, in order to drive the adaptivity. In many cases, extended variants of sort (*sort*), hash (*hash*), pipelining joins (PJ) and parallel operators (*parl*) are used in order to enhance the query evaluator with the capability to adapt dynamically to certain changes in the environment. For example, special physical operators could be used to adapt to changes in memory availability, when these changes happen during operator execution.

Operator Replacement, takes place when a particular physical operator can be replaced with a logically equivalent one at runtime in order to produce an optimised QP. For example, a hash join may be replaced by an index join, if an index on the joining attribute becomes available during runtime. As there are no restrictions on the choices that can be made, the relevant category is marked as *any* in the relevant field of Table 1.

No effects, if an AQP technique does not affect the physical plan at all. This is denoted by *no*.

Although most of the techniques could be applied to parallel and distributed environments, a few of them deal with query re-partitioning (*Par*) and re-scheduling of these partitions (*Sch*). An example of effects on the partitioning is when AQP allows partitions to be joined together or split into many sub-partitions at runtime. If AQP is designed in such a way that it can explicitly modify the partitioning of a QP running in a parallel or distributed environment, then the relevant value in Table 1 is *yes*. Moreover, some techniques invoke the query optimiser during query execution and, in that way, they may modify the partitioning, provided that they are applied to a parallel or distributed setting (denoted as *maybe* in Table 1).

The scheduling of the query plan is modified when an operation in execution is moved to another node, the number of execution nodes changes during execution, or the relative input rates for these nodes is adjusted in order to achieve optimal performance. In wide-area systems, it is also desirable to have the capability to replace a source if it fails, and to postpone the decisions on which node to execute some operations until runtime. Finally the amount of data allocated to each participating node, can be controlled in order to achieve load balance. In all the above cases the scheduling is affected. The semantics of the relevant field in Table 1 is the same as for *partitioning*.

2.2 Characteristics of Adaptive Query Processing Techniques

AQP methods vary significantly in what they attempt to adapt to. There are six main different areas of focus, as shown in the *Focus* field:

Fluctuations in memory (*mem_fl*). Systems in this category try to adapt to memory shortages and to the availability of excess memory. Memory shortages can happen due to competition from higher-priority transactions, for example. In that case, running query plans may be forced to release some or all of the resources they hold. On the other hand, executing transactions may be given additional resources as they become available, when, e.g., other queries complete and free their buffers.

User preferences (*user_pr*). Adapting to user preferences includes cases where users are interested in obtaining some partial results of the query quickly. In order to meet such needs, the system produces results incrementally, as they become available, and the user can tradeoff the time between successive updates of the running queries and the amount by which the partial and the actual results differ at each update. The user can also classify the elements of the output in terms of importance, and in that case the query evaluator adapts its behaviour in order to produce more important results earlier.

Data arrival rates (*dar*). Techniques that adapt to data arrival rates apply to parallel and distributed systems, where the response times of remote data sources are less predictable.

Actual statistics (*ac_stats*) about the data sources which are not available initially. In some cases, it is not possible at compile time to gather accurate statistics about the data sources. A solution to this problem is to collect such statistical information at runtime, thereby ensuring that they are valid for the current circumstances, and to adapt query execution based on it. Techniques that adopt this policy may change the query plan when the actual statistics have become available.

Fluctuations in performance (*per_fl*). Problems in performance fluctuation arise more often in parallel systems. In these systems, performance can degrade when even a single node experiences poor data layout on disks, high memory and CPU load, or competing data streams.

Any changes in the environment (*any*) combine elements of all the other categories. Some techniques are comprehensive, as they can adapt to many kinds of changes in their environment, i.e. to computer resources, like memory and processor availability, and to data characteristics, like operator costs, selectivities and data arrival rates.

The aim of adaptive techniques may also differ. More specifically, AQP may aim at:

Minimising the total response time (*trt*) for a single evaluation. It is important to note that many AQP techniques use specific operators in order to achieve adaptivity. Adaptive operators perform better than static ones when there are changes in the environment. However, if all the parameters of the query are known at compile time and do not change during execution, a static QP may result in better response times.

Minimising the initial response time (*irt*). Some systems are mostly interested in minimising the time until useful information is returned to the user, according to his or her preferences. These systems are optimised to produce more, or more important, results during the early stages of the execution.

Maximising the throughput (*thr*) of the system. Some AQP techniques aim at minimising the total response time for the evaluation of a set of independent operations. Apparently, there is an interplay between the techniques of the last two categories. E.g., if there is only one operation to be evaluated, they share exactly the same objective. This is only one of the many possible ways that techniques described in the survey can interfere with each other.

Another characteristic of AQP techniques is the nature of the feedback (*FN*) they collect from the environment in order to examine whether there is scope for adaptation of the QP. Different kinds of feedback include:

- memory availability (*mem_av*),
- potential input from the user (*user_in*), such as priority ratings for different parts of the result or for the rate of updates of partial results,
- input availability (*in_av*), which is relevant to operators that may have one or more of their inputs blocked and need to check whether delayed inputs have become available,

– workload (l),
– data rates (dr), i.e. the rates at which tuples are produced, and
– statistical information ($stats$), other than those covered by the above categories, e.g. the size of each relation, the approximate number and frequency of different values for an attribute, the availability of indices, the layout of the data on disk, lists of replicated databases, etc.

For the frequency of feedback collection (FFC), there are two possible values:

– inter-operator ($inter$), and
– intra-operator ($intra$) frequency.

Techniques in the first category collect and may act on feedback between the execution of different physical operators in the QP. Collection can be triggered after certain physical operators have been evaluated or after special events, like the arrival of partial results from local data sources in multidatabase environments. In techniques adapting with intra-operator frequency, feedback is collected during the evaluation of physical operators. Check-points are added to the operator execution for this purpose. In general, feedback is collected after a block of tuples has been processed. In the limit, this block consists of a single tuple, resulting in a potentially different plan for each tuple.

Techniques differ in whether they have been designed for

– uniprocessor (U),
– parallel (P), or
– distributed (D) environments,

as shown in the *Env* field of Table 1. This does not mean that a technique initially developed on a single processor platform cannot be applied to a parallel or distributed setting, rather it is used to indicate that the developers may not have taken into consideration some of the specific problems that arise in such settings. For example, for a QP to be optimal in a distributed environment, statistical information about network cost, replicated data sources and the available nodes is required, along with commonly available statistics which underpin good models at least for uniprocessor systems. Such statistics typically include selectivities, histograms and indices among others. The main difference between distributed and parallel DBMSs is that the former are constructed by a collection of independent, semi-autonomous processing sites that are connected via a network that could be spread over a large geographic area, whereas the latter are tightly coupled systems controlling multiple processors that are in the same location, usually in the same machine room [12].

The responsibility for adaptivity decisions (AD) may also be assigned differently. Such decisions can be taken:

by the physical operators (O), when physical operators are implemented in such a way that they can adapt their behaviour during runtime without being invoked by any other DBMS component or optimiser;

locally (L), if the query optimiser, or any other DBMS component, takes action during runtime to ensure that the remainder of the query plan will be evaluated in an optimal manner;

globally (G), which applies to parallel and distributed systems when a global view of the state of many participating nodes may be required, in order to adapt.

The last criterion used in this survey is whether a technique is realised as a physical operator (O), a concrete algorithm (A), or a system (S). In the first case, the whole technique can be represented in the physical query plan as a single physical operator. A technique is classified as an algorithm, if it implies extension and specific manipulation of the operators of a QP, in order to achieve adaptivity. A system comprises many autonomous techniques in an integrated computational entity.

3 Adaptive Query Operators

This section describes query operators which have the capability to adjust their behaviour according to changing conditions and the information available at runtime. Adaptive operators continuously collect and evaluate the feedback from the environment. They act autonomously in order to adapt to changes but have limited effects on the executing QP, as they cannot perform radical changes to it. Additionally, as they are basically developed for uni-processor environments, they do not consider partitioning and scheduling issues.

3.1 Memory Adaptive Sorting and Hash Join

[19] introduces techniques that enable external sorting to adapt to fluctuations in memory availability. External sorting requires many buffers to run efficiently, and memory management significantly affects the overall performance. The memory-change adaptation strategy introduced is *dynamic splitting*, which adjusts the buffer usage of external sorts to reduce the performance penalty resulting from memory shortages and to take advantage of excess memory. It achieves that by splitting the merge step of external sorts into a number of sub-steps in case of memory shortage, and by combining sub-steps into larger steps when sufficient buffers are available. Alternative strategies, like *paging* and *suspension*, are non-adaptive and yield poor results [19]. Adopting memory-adaptive physical operators for sort operations avoids potential under-utilisation of memory resources (e.g. when memory reserved prior to execution based on estimates is in fact more than the memory required) or thrashing (e.g. when memory reserved prior to execution is in fact less than the memory required), and thus reduces the time needed for query evaluation. This technique is complementary to existing optimised strategies for the split phase of external sorts. In the same way, new memory-adaptive variants of merge-sort operators can also be developed.

For join execution, when the amount of memory changes during its lifetime, a family of memory adaptive hash joins, called *partially preemptible hash joins*

Technique	LL	PP	Par	Sch	Focus	Aim	FN	FFC	Env	AD	real.
Memory Adaptive Sorting [19]	no	sort	no	no	mem_fl	trt	mem_av	intra	U	O	O
Memory Adaptive Merge-Sort [27]	no	sort	no	no	mem_fl	thr	mem_av	intra	U	O	A
PPHJ [20]	no	hash	no	no	mem_fl	trt	mem_av	intra	U	O	O
Ripple [9]	no	PJ	no	no	user_pr	irt	user_in	intra	U	O	O
XJoin [24]	no	PJ	no	no	dar/ user_pr	trt - irt	in_av	intra	U	O	O
Dynamic Reordering [23]	no	sort	no	no	user_pr	irt	user_in	intra	U	O	O
Mid-Query Re-optimisation [15]	rem	any	maybe	maybe	ac_stats	trt	stats	inter	U	L	A
Eddies [4, 5]	op_or	no	no	no	any	trt	stats	intra	U	L	A
Rivers [3]	no	parl	no	yes	per_fl	trt	l/dr	intra	P	O	A
MIND [18]	op_or	any	no	yes	ac_stats	trt	stats	inter	D	G	A
Query Scrambling [1, 2, 26]	rem	any	no	no	dar	trt	in_av	inter	D	G	A
Pipeline Scheduler [25]	op_or	PJ	no	no	user_pr	irt	dr/ user_in	intra	D	O	A
Bouganim *et al* [6, 7]	rem	PJ	maybe	maybe	dar/ mem_fl	trt	dr/ mem_av	intra	D	O/G	A
Conquest [16, 17]	rem	any	yes	yes	any	trt	stats	inter	P	G	A
Tukwila [13, 14]	rem	PJ	maybe	yes	any	trt - irt	stats	inter	D	O/G	S
Telegraph [11]	op_or	parl	no	yes	any	trt	stats/ l/dr	intra	D	O/L	S
dQUOB [21, 22]	op_or	no	no	no	ac_stats	trt	stats	inter	D	L	A

Table 1. Properties of existing adaptive query processing techniques

(PPHJs), is proposed in [20]. Initially, source relations are split and held in memory. When memory is insufficient, one partition held in memory flushes its hash table to disk and deallocates all but one of its buffer pages. The most efficient variant of PPHJs for utilising additional memory is when partitions of the inner relation are fetched in memory while the outer relation is being scanned and partitioned. This method reduces I/O and, consequently, the total response time of the query. Although it cannot adapt very well to rapid memory fluctuations, it outperforms non-memory-adaptive hybrid hash joins, which employ strategies, such as *paging* and *suspension*.

3.2 Operators for Producing Partial Results Quickly

In many applications, such as online aggregation, it is useful for the user to have the most important results produced earlier. To this end, pipelining algorithms are used for the implementation of join and sort operations. The other kind of operators are block ones, which produce output after they have received and processed the whole of their inputs.

Ripple joins [9] are a family of physical pipelining join operators that maximise the flow of statistical information during processing. They generalise block nested loops (in the sense that the roles of inner and outer relation are continually interchanged during processing) and hash joins. Ripple joins adapt their behaviour during processing according to statistical properties of the data, and user preferences about the accuracy of the partial result and the time between updates of the running aggregate. Given these user preferences, they adaptively set the rate by which they retrieve tuples from each input of the ripple join. The ratio of the two rates is reevaluated after a block of tuples has been processed.

XJoin [24] is a variant of Ripple joins. It is an operator with low memory requirements (due to partitioning of the inputs), so many such operators can be active in parallel. Apart from producing initial results quickly, XJoin is also optimised to hide intermittent delays in data arrival from slow and bursty remote sources by reactively scheduling background processing. XJoin executes in three stages: Initially, it builds two hash tables, one for each source. In the first stage, a tuple, which may reside on disk or memory, is inserted into the hash table for that input upon arrival, and then is immediately used to probe the hash table of the other input. A result tuple will be produced as soon as a match is found. The second stage is activated when the first stage blocks and it is used for producing tuples during delays. Tuples from the disk are then used to produce some part of the result, if the expected amount of tuples generated is above a certain activation threshold. The last stage is a clean-up stage as the first two stages may only partially produce the final result. In order to prevent the creation of duplicates, special lists storing specific time-stamps are used.

To obtain and process the most important data earlier, a sort-like re-ordering operator is proposed in [23]. It is a pipelining dynamic user-controllable reorder operator that takes an unordered set of data and produces a nearly sorted result according to user preferences (which can change during runtime) in an attempt to ensure that interesting items are processed first; it is a best-effort operator.

The mechanism tries to allocate as many interesting items as possible in main memory buffers. When consumers request an item, it decides which item to process (i.e. it uses a pull model). The operator uses the two-phase Prefetch & Spool technique: in the first phase tuples are scanned, and uninteresting data are spoooled to an auxiliary space until input is fully consumed. In the second phase the data from the auxiliary space are read.

4 Algorithms and Systems for Adaptive Query Processing

4.1 Extensions to Adaptive Operators

A work on memory-adaptive sorting, which is complementary to [19] as discussed in Section 3.1, is presented in [27]. This method focuses on improving throughput by allowing many sorts to run concurrently, while the former focuses on improving the query response time. Because it depends on the sort size distribution, in general, limiting the number of sorts running concurrently can improve both the throughput and the response time. The method proposed enables sorts to adapt to the actual input size and changes in available memory space. For the purpose of memory adjustment, all the possible stages of a sort operation regarding memory usage are identified and prioritised. The operations may be placed in a wait queue because of insufficient memory in the system. Priority ratings are assigned to the wait queues as well. When memory becomes available, the sorts in the queue with the highest priority are processed first. A detailed memory adjustment policy is responsible for deciding whether a sort should wait or proceed with its current workspace. If it is decided to wait, further decisions on which queue to enter, and whether to stay in the current queue or to move to another, are made. Like [19], [27] has impact only on the physical query plan and adapts with intra-operator frequency.

The method in [25] for quicker delivery of initial results in pipelined query plans extends Ripple joins [9] and XJoin [24] with the capability to re-order join operators of the QP. Instead of scheduling operators, the *Dynamic Pipeline Scheduler* splits them into independent units of execution, called *streams*, and then schedules these streams. This reduces the initial response time of the query but may result in increases in the total execution time.

The Dynamic Pipeline Scheduler consists of two algorithms. In the *rate-based pipeline scheduling algorithm*, all result tuples have the same importance. The algorithm maximises the result output rate by prioritising and scheduling the flow of data across pipelined operators. Characteristics of the operators, such as cost and productivity, can be dynamically changed (due to blocked inputs, finished operators, etc). Productivity differs from selectivity in that selectivity describes how many tuples will eventually be produced, whereas productivity describes how many tuples will be produced at that point in the execution [25]. In the *importance-based tuple regulation algorithm*, the result tuples have different degrees of importance (according to the user) and the algorithm allocates more resources to the more important tuples. For the importance-rated algorithm, dynamic collection of query statistics is necessary. So, this algorithm needs to be combined with another technique that enables the collection of necessary statistics at runtime. The scheduling policy is modified on the fly in light of changes in system behaviour. Ideally, the output rates should be recomputed every time a tuple is processed by a stream, but, in fact, the scheduler is invoked less often in order to avoid large overheads.

4.2 Algorithms That Adapt to Data Arrival Rates

In the light of data arrival delays, a common approach is to minimise idle time by performing other useful operations, thus attenuating the effect of such delays. Query Scrambling [1, 2, 26] and a generic AQP architecture discussed in [6, 7] are two representative examples in this area.

Query scrambling focuses on problems incurred by delays in receiving the first tuples from a remote data source. The system performs other useful work in the hope that the problem will eventually be resolved and the requested data will arrive at or near the expected rate from then on. In the first phase, it changes the execution order to avoid idling (which is always beneficial) following a specific procedure, and, in the second, it introduces new operations in the QP, which is risky. In query scrambling, there is a trade-off between the potential benefit of modifying the QP on the fly and the risk of increasing the total response time instead of reducing it.

[6, 7] deal also with the problem of unpredictable data arrival rates, and, additionally, with the problem of memory limitation in the context of data integration systems. A general hierarchical dynamic query processing architecture is proposed. Planning and execution phases are interleaved in order to respond in a timely manner to delays. The query plan is adjusted to the data arrival rate and memory consumption. In a sub-optimal query plan, the operator ordering can be modified or the whole plan can be re-optimised, by introducing new operators and/or altering the tree shape. In general, operators that may incur large CPU idle times are pushed down the tree. Scheduling such operators as soon as possible increases the possibility that some other work is available to schedule concurrently if they experience delays. Bushy trees are also preferred because they offer the best opportunities to minimise the size of intermediate results. Thus, in case of partial materialisation, the overhead remains low. The query response time is reduced by running concurrently several query fragments (with selection and ordering being based on heuristics) and partial materialisation is used, as mentioned above. Because materialisation may increase the total response time, a *benefit materialisation indicator (bmi)* and a *benefit materialisation threshold (bmt)* are used. A *bmi* gives an approximate indication of the profitability of materialisation and the *bmt* is its minimum acceptable value.

4.3 Algorithms that Defer some Optimisation Decisions until Runtime

To ensure optimality in query evaluation, algorithms may wait until they have collected statistics derived from the generation of intermediate results thus far.

Kabra and deWitt [15] introduced an algorithm that detects sub-optimality in QPs at runtime, through on-the-fly collection of query statistics, and improves performance by either reallocating resources such as memory or by modifying the QP. This method aims to combat the problem of constructing QPs based on inaccurate initial statistical estimates about the data sources. To this end, a new operator, called the *statistics collector operator*, is used. The re-optimisation algorithm is heuristics-based and relies heavily on intermediate data materialisation. The *statistics collector insertion algorithm* inserts *statistics collector operators*, ensuring that these do not slow down the query by more than a specific fraction, and also assigns a potential inaccuracy level of low, medium, or high to the various estimates. So, overhead is taken into consideration and only statistics that can affect the remainder of the QP are collected.

The MIND system [18] uses a similar approach to address the problem of where to perform combination operations on partial results of a global query returned by local DBMSs in a multidatabase system. Due to the fact that useful statistics about these results are difficult for a static optimiser to estimate, decisions on where to execute operations on data that reside at remote sources are deferred until the arrival of the actual partial results. The decisions are based on both the selectivities and the cost involved.

dQUOB conceptualises streaming data with a relational data model, thus allowing the stream to be manipulated through SQL queries [21, 22]. For query execution, runtime components embedded into data streams are employed. Such components are called *quoblets*. Detection of changes in data stream behaviour is accomplished by a statistical sampling algorithm that runs periodically and gathers statistical information about the selectivities of the operators into an equi-depth histogram. Based on this information, the system can reorder the select operators on the fly.

4.4 The Telegraph Project

The Telegraph project [11] combines River [3] with Eddies [4, 5] to yield an adaptive dataflow system. Thus, unpredictable dataflows can be routed through computing resources in a network, resulting in a steady manageable stream of useful information. A dataflow engine is an execution environment that moves large amounts of data through a number of operators running on an arbitrary number of machines. Such an engine is a database query processing system. Eddies can reshape dataflow graphs for each tuple they receive to maximise performance, while rivers are responsible for load balance across multiple nodes.

More specifically, Eddies is a query processing mechanism that continuously (i.e., for each tuple) reorders operators on the fly in order to adapt dynamically to changes in computing resources (e.g., memory) and data characteristics (operator costs, operator selectivities, and rates at which tuples arrive from the inputs), provided that these operators are pipelined [4, 5]. Although current work focuses on uniprocessor environments, Eddies can be exploited in large federated databases. In order to insert Eddies in a query plan, joins should allow frequent and efficient reoptimisation. Such joins have frequent moments of symmetry, adaptive or non-existent synchronisation barriers, and minimal ordering constraints. A moment of symmetry is a state in which the orders of the inputs to a binary operator can be changed without modifying any state in this operator. A synchronisation barrier occurs when an input of a binary operator waits until the other input produces a specific tuple. Joins that are appropriate for Eddies include Ripple joins, pipelined hash join, Xjoin, and index joins, with the Ripple join family being the most efficient.

A River [3] is a dataflow programming environment and I/O substrate for clusters of computers to provide maximum performance even in the face of performance heterogeneity. In intra-operator parallelism, data is partitioned to be processed among system nodes. The two main innovations in a River are *distributed queues (DQ)* and *graduated declustering (GD)*, which enable the system to provide data partitioning that adapts to changes in production and consumption nodes. DQs are responsible for naturally balancing load across consumers running at different rates. GD is used for full-bandwidth balanced production, with the effect that all available bandwidth is utilised at all times, and all producers of a given data set complete near-simultaneously. While GD is transparent to programmers resulting in new ready-to-use physical operators, DQs need to be added to the operator interface manually.

4.5 Adaptive Query Processing in High-Performance Systems

AQP engines are an integral part of high-performance data mining and data integration systems, like Conquest [16, 17] and Tukwila [13, 14], respectively.

The Conquest query processing system enables dynamic query optimisation in a parallel environment for long-running queries following a triggering approach in response to runtime changes [16, 17]. Changes relate to system parameters (e.g., new processors become available, while others may be withdrawn), and data characteristics relevant to query statistics (e.g., a buffer queue becomes empty, or selectivity is higher than initially estimated). Such changes make the re-computation of cost estimates necessary. The capabilities of Conquest for modifying a QP are very strong, especially in terms of partition scheduling, but are limited to unary operators, such as scans. Systems that can modify the remainder of a QP use materialisation techniques in order to capture and restore the intermediate state of the operators and executions. One partial exception to this is the Conquest System, which may only require the buffered records to be materialised before reconfiguring the operators.

The Tukwila [13, 14] project is a data integration system in which queries are posed across multiple, autonomous and heterogeneous sources. Tukwila attempts to address the challenges of generating and executing plans efficiently with little knowledge and variable network conditions. The adaptivity of the system lies in the fact that it interleaves planning and execution and uses adaptive operators. Coordination is achieved by event-condition-action rules. Possible actions include operator reordering, operator replacement and re-optimisation of the remainder of a QP. The Tukwila system integrates adaptive techniques proposed in

[15, 26]. Re-optimisation is based on pipelined units of execution. At the boundaries of such units, the pipeline is broken and partial results are materialised. The main difference from [15] is that materialisation points can be dynamically chosen by the optimiser (e.g., when the system runs out of memory). Tukwila's adaptive operators adjust their behaviour to data transfer rates and memory requirements. A *collector operator* is also used, which is activated when a data source fails so as to switch to an alternative data source.

5 Discussion

In general and according to results from evaluation of prototypes, AQP techniques often attain their goal of decreasing the response time of a query in the presence of changes in the parameters they consider, or of delivering useful results as early as possible. However, in AQP, there is always a trade-off between the potential benefits from adapting to current conditions on the fly with very high frequency, and the risk of incurring large overheads.

The techniques that have more extensive capabilities in terms of the modifications they can induce in the running plan, are more expensive and risky than the others. Before applying such techniques, certain steps need to be taken (e.g., validation against a cost model) to ensure that adaptation is likely to be beneficial. These techniques are mostly applicable to parallel and distributed settings, taking the adaptivity decisions at a global level and relying heavily on the materialisation of intermediate results. This happens because in parallel and distributed settings, network costs and resource availability need to be taken into consideration, and not adapting to changes in data transfer rates or resources may have detrimental effects. On the other hand, less complex adaptive strategies (e.g. those that do not affect the logical plan of the query) are generally intended for single processor environments, where it is more common that satisfactorily accurate statistics can be obtained at compile-time.

Surprisingly, little attention has been paid to changes in the pool of available processors and data sources, which have a great impact on long-running queries in wide-area systems. As a result, most of the AQP techniques proposed so far for parallel and distributed DBMSs do not alter their partitioning and scheduling policies dynamically at runtime.

In summary, for memory fluctuations, the main strategy is to dynamically adjust the usage of memory buffers. In the light of data arrival delays, a common approach is to minimise idle time by performing other useful operations, and thus to hide such delays. The problem of inaccurate cost and statistical estimates is addressed by collecting and using the actual statistics on the fly and then comparing them with the estimates in order to decide whether there is an opportunity for query re-optimisation. Techniques that attempt to adapt to various changes in a wide area environment follow a triggered approach. Event-condition-action rules are used, where events can be raised both by operator execution (e.g. a block has completed its processing or there is insufficient memory) and by environmental changes (e.g. a new processor has become available).

In online aggregation queries where the time to provide the user with useful information quickly is more important than the time to query completion, adaptive pipelining operators need to be adopted. Non-traditional operators are also required in all the cases where the query needs to adapt with intra-operator frequency.

6 Conclusions

In this survey, existing adaptive query processing techniques have been classified and compared. The opportunities offered by such techniques to modify a query plan on the fly, have been identified. In addition, adaptive systems have been classified according to the problem they focus on, their objectives, the nature of feedback they collect from the environment and the frequency at which they can adapt. Other areas of interest include the implementation environment and which component is responsible for taking the adaptation decisions. The survey reveals the inadequacy of existing techniques to adapt to environments where the pool of resources is subject to changes. In particular, the paper provides evidence to the need for research into AQP in computing infrastructures where resource availability, allocation and costing are not, by definition, decidable at compile time. Consequently, a promising area for future work is the development of concrete cost models to evaluate whether the expected benefits from the improved query plan and decision quality compensate for the cost of collecting and evaluating feedback from the environment during execution time.

Acknowledgement: This work is supported by the Distributed Information Management Programme of the Engineering and Physical Science Research Council, through Grant GR/R51797/01. We are pleased to acknowledge their support.

References

1. L. Amsaleg, M. Franklin, and A. Tomasic. Dynamic query operator scheduling for wide-area remote access. *Distributed and Parallel Databases*, 6(3):217–246, 1998.
2. L. Amsaleg, M. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *Proc. of the Fourth International Conference on Parallel and Distributed Information Systems*, pages 208–219. IEEE Computer Society, 1996.
3. 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, Atlanta, GA, 1999. ACM Press.
4. R. Avnur and J. Hellerstein. Continuous query optimization. Technical Report CSD-99-1078, University of California, Berkeley, 1999.
5. R. Avnur and J. Hellerstein. Eddies: continuously adaptive query processing. In *Proc. of ACM SIGMOD 2000*, pages 261–272, 2000.
6. L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. A dynamic query processing architecture for data integration systems. *IEEE Data Engineering Bulletin*, 23(2):42–48, 2000.

7. 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.
8. I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 1999.
9. P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *Proc. of ACM SIGMOD 1999*, pages 287–298, 1999.
10. A. Hameurlain and F. Morvan. An overview of parallel query optimization in relational databases. In *11th International Workshop on Database and Expert Systems Application (DEXA'00)*. IEEE Computer Society, 2000.
11. 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.
12. Y. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.
13. Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of ACM SIGMOD*, pages 299–310, 1999.
14. Z. Ives, A. Levy, D. Weld, D. Florescu, and M. Friedman. Adaptive query processing for internet applications. *IEEE Data Engineering Bulletin*, 23(2):19–26, 2000.
15. 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.
16. K. Ng, Z. Wang, and R. Muntz. Dynamic reconfiguration of sub-optimal parallel query execution plans. Technical Report CSD-980033, UCLA, 1998.
17. K. Ng, Z. Wang, R. Muntz, and S. Nittel. Dynamic query re-optimization. In *Proc. of 11th International Conference on Statistical and Scientific Database Management*, pages 264–273. IEEE Computer Society, 1999.
18. F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac. Dynamic query optimization in multidatabases. *Data Engineering Bulletin*, 20(3):38–45, 1997.
19. H. Pang, M. Carey, and M. Livny. Memory-adaptive external sorting. *The VLDB Journal*, pages 618–629, 1993.
20. H. Pang, M. Carey, and M. Livny. Partially preemptible hash joins. In *Proc. of ACM SIGMOD 1993*, pages 59–68, 1993.
21. B. Plale and K. Schwan. dquob: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, pages 263–270, 2000.
22. B. Plale and K. Schwan. Optimizations enabled by a relational data model view to querying data streams. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2001.
23. V. Raman, B. Raman, and J. Hellerstein. Online dynamic reordering. *The VLDB Journal*, pages 247–260, 2000.
24. T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.
25. T. Urhan and M. Franklin. Dynamic pipeline scheduling for improving interactive query performance. *The VLDB Journal*, pages 501–510, 2001.
26. T. Urhan, M. Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. In *Proc. of ACM SIGMOD 1998*, pages 130–141, 1998.
27. W. Zhang and P. Larson. Dynamic memory adjustment for external mergesort. *The VLDB Journal*, pages 376–385, 1997.