Adaptive Query Processing in the Looking Glass

Shivnath Babu*
Stanford University
shivnath@cs.stanford.edu

Pedro Bizarro
University of Wisconsin, Madison
pedro@cs.wisc.edu

Abstract

A great deal of work on adaptive query processing has been done over the last few years: Adaptive query processing has been used to detect and correct optimizer errors due to incorrect statistics or simplified cost metrics; it has been applied to long-running continuous queries over data streams whose characteristics vary over time; and routing-based adaptive query processing does away with the optimizer altogether. Despite this large body of interrelated work, no unifying comparison of adaptive query processing techniques or systems has been attempted; we tackle this problem. We identify three families of systems (*plan-based*, *CQ-based*, and *routing-based*), and compare them in detail with respect to the most important aspects of adaptive query processing: plan quality, statistics monitoring and re-optimization, plan migration, and scalability. We also suggest two new approaches to adaptive query processing that address some of the shortcomings revealed by our in-depth analysis: (1) *Proactive re-optimization*, where the optimizer chooses query plans with the expectation of re-optimization; and (2) *Plan logging*, where optimizer decisions under different conditions are logged over time, enabling plan reuse as well as analysis of relevant statistics and benefits of adaptivity.

1 Introduction

For more than twenty years most database systems have used the *plan-first execute-next* approach to query processing, where an optimizer first picks an efficient plan for a query, and then this plan is used to execute the query to completion [34]. One effect of this approach has been to make database systems critically dependent on the optimizer. Unfortunately, optimizers may make mistakes: they may use outdated statistics or invalid assumptions like independence of attributes, resulting in query plans that are suboptimal by orders of magnitude [14, 30]. Several factors have increased the severity of this problem: Users are now posing more complex queries over larger data sets than before; queries may now involve data sources, e.g., web services, for which no statistics are available; database systems are being deployed in highly unpredictable and changeable environments [10, 11, 31].

Adaptive query processing (*AQP*) is a promising approach to avoid the performance penalty caused by optimizer mistakes, unknown statistics, and changes in data and system conditions [28, 30]. With AQP, the optimization and execution stages of processing a query are interleaved, possibly multiple times over the running time of the query. AQP makes query processing more robust to optimizer mistakes, unknown statistics, and to changes in conditions over the running time of a query. AQP is part of the general trend toward *autonomic computing*, which aims to minimize human effort in running systems efficiently.

Numerous techniques have been proposed for AQP, and some have been prototyped in database systems. To the best of our knowledge, Table 1 lists all techniques that have been proposed for adaptive processing of a single query (as opposed to techniques that use the statistics collected during the execution of the current query change to better optimize future queries, e.g., [35]). Figure 1 shows a timeline listing the year in which the first paper describing each general technique was published. Some important observations can be made from Table 1 and Figure 1:

- Most of the work on AQP has been done in the past 5-6 years.
- The applicability of AQP has been demonstrated on real-life data and query workloads. Some of these techniques are being incorporated into commercial database systems.
- There are significant differences among the proposed systems and techniques for AQP: different query semantics, different types of data sources, different cost metrics and optimization frameworks, and different scopes of adaptivity including choice of access methods, plan shapes, pipelining, and memory allocation across operators.

Since there has been a flurry of activity recently in AQP systems, now is a good time to juxtapose these systems.

---

*Supported by NSF Grants IIS-0118173 and IIS-9817799.

Proceedings of the 2005 CIDR Conference

Name	Query Type; Data Type	Brief Description	Novel Features
Re-Opt [28]	SQL; D	Re-optimize remaining query if statistics of materialized subplan differs significantly from optimizer's estimates	First complete AQP prototype based on possible re-optimization during execution
POP [30]	SQL; D	Similar to Re-Opt. Avoids unnecessary re-optimization and supports re-optimization within pipelines	(i) Prototyped in a commercial optimizer, (ii) Shows significant benefits on real workloads
Tukwila [25]	SQL; N	Query optimization for data integration systems	(i) Adaptivity at both operator and plan levels, (ii) Event-condition-action rules for AQP
Corrective Proc. [24]	SQL; N	Query processed in stages by partitioning data, statistics from previous stages used to improve plan, final cleanup phase	Techniques to exploit properties of data arrival from remote sources, e.g., sortedness
Query Scrambling [38]	SQL; D, N	Dealing with startup delay and burstiness of remote data sources	Techniques to minimize idle time during query processing
Pipeline Sched. [37]	SQL; D, N	Scheduling operators in pipelined plans to reduce query response time	AQP for the response time metric
Conquest [32]	SQL; D	Adaptive distributed query processing	Extensive use of triggers for AQP
DQE [9]	SQL; D, N	Query optimization for data integration	Adaptivity at plan and operator-scheduling levels
DEC-Rdb [1]	SQL; D	Starts multiple access methods competitively, chooses best, and stops others	First to explore optimization based on competing plan choices against each other
Memory adap. [17]	SQL; D	Adaptive allocation of memory to operators	Prototyped in a commercial system
Redbrick	SQL; D	Computes intermediate results involving dimension tables before choosing access method for fact table	Perhaps the first commercial use of full-blown re-optimization
Ingres [39]	SQL; D	Tuples in joins routed to nested-loop join operators adaptively	Extremely adaptive join ordering
Eddies [4, 18, 29, 33]	SQL,CQ; D,N,S	Query processing by routing tuples through a pool of operators	(i) No optimizer, (ii) Routes chosen based on operator exploration, (iii) Fine-grained adaptivity
River [3]	Sort; D	Adaptive sorting using a machine cluster	Established a sorting record
NiagaraCQ [13]	CQ; D, N	Techniques to handle very large numbers of continuous queries	Adaptive sharing of common subexpressions
StreaMon [5, 6, 7]	CQ; S	Integrated statistics collection and re-optimization for data stream systems	(i) Sampling-based statistics tracking, (ii) Efficient approximation algorithms with provable guarantees
CAPE [41]	CQ; S	Adaptive operators, scheduling, and distributed processing	Adaptivity at many levels
Parametric opt. [16, 19, 20, 22, 23]	SQL; D	Handling parameters (e.g., memory size, user inputs) whose values are unknown during optimization	An optimization framework to generate plans optimal for partitions of the parameter domains, and defer plan choice until the actual parameter values are known at run-time
Expected-cost opt. [15]	SQL; D	Finds plan with least expected cost given distribution of values of parameter(s)	An optimization framework that can be used as an alternative to parametric optimization

Table 1: Systems and techniques for adaptive processing of a single query. Query type is conventional SQL and/or CQ (for Continuous Queries). Data type is D (local disk-resident relations), N (network-bound relations), and/or S (data streams).

The main contribution of this paper is a detailed qualitative comparison of the systems in Table 1 with respect to a wide range of issues related to AQP. To our knowledge, no previous attempt at comparing different AQP systems [21, 24, 27, 30] is as exhaustive as the one we present. Our purpose is not to "bake off" these systems against one another; instead, we want to bring out their characteristic features and algorithms.[1] For example, our comparison shows that, in spite of the numerous differences among these systems, there are many commonalities in the way they support AQP. To our knowledge, four broad comparisons of AQP systems have appeared previously in the lit-

---

[1] The title of this paper highlights the analogy between the goals of this paper and the act of looking in a mirror to find out how we (the AQP community) look, and how we can improve our appearance.

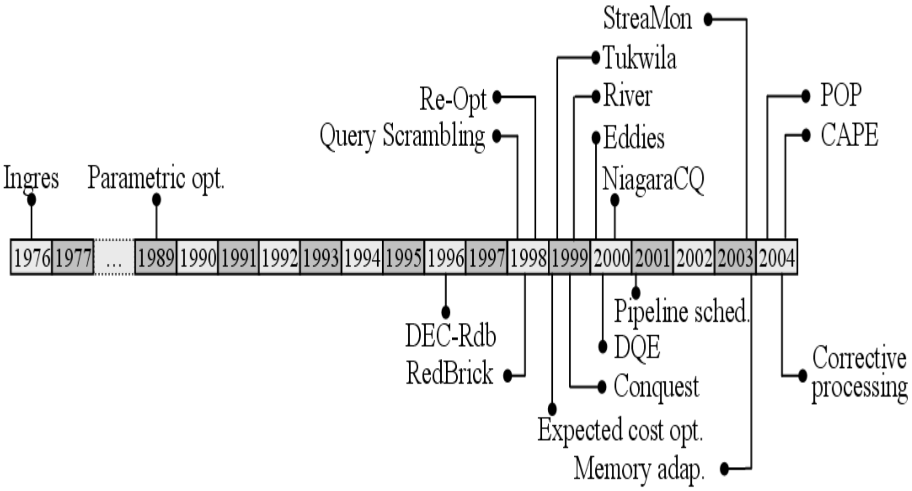

Figure 1: Publication timeline

erature [21, 24, 27, 30]. All of these comparisons take a higher level view of AQP than we do, which limits the depth of their comparisons. On the other hand, we focus on AQP systems that interleave optimization, execution, and statistics tracking, possibly multiple times, during the running time of a single query. This focus enables us to present a more in-depth and exhaustive comparison than in [21, 24, 27, 30]. Based on our exhaustive study, we also suggest two new techniques for AQP to address some of the remaining shortcomings of current AQP systems.

The rest of this paper is organized as follows. In Section 2, we detail the main motivations for current AQP systems. In Section 3, we identify the core tasks in AQP, partition current AQP systems into three families, and specify how each of these families handles the core tasks. In Section 4, we present a detailed qualitative comparison of the three system families with respect to a wide range of specific issues relevant to AQP. In Section 5, we suggest two new techniques for AQP. We conclude in Section 6.

2 Motivations for AQP

In this section we detail the main motivations for AQP.

1. **Correcting Optimizer Mistakes:** There are many reasons why an optimizer may make a mistake while costing physical plans for a query, picking a plan that performs significantly worse than the best plan. The usual culprit is the unavailability of statistics about attribute correlations and skewed attribute distributions. Since collecting statistics is an expensive operation in a database system, statistics may not be kept up-to-date, which also may contribute to optimizer mistakes. The severity of optimizer mistakes and the usefulness of AQP in solving this problem are demonstrated in [24, 25, 26, 28, 30] using synthetic and real-life data sets and query workloads. For example, in [30] AQP is shown to improve the performance of real-life complex queries by an order of magnitude.

2. **Coping with Unknown Statistics:** With the growing popularity of web services, it has become much easier to make any data source available online [40]. Now a database system may have to execute a query involving one or more sources for which no statistics are available. A plan-first execute-next approach is of limited use in this environment, and AQP may be the only reasonable approach. AQP is also important in data integration systems and other data management systems that support queries over autonomous remote sources, e.g., see [9, 24, 25, 26, 33, 38].

3. **Reacting to Changes in Input Characteristics and System Conditions:** *Continuous queries*, which are common in data stream systems [10, 11, 31], are long-running queries, so input characteristics may change during the running time of a continuous query. For example, stream arrival may be bursty, alternating unpredictably between periods of slow arrival and periods of extremely fast arrival [38]. Also, system conditions, e.g., the memory available to a single continuous query, may vary significantly over the query's running time. AQP is important to maintain good performance in these contexts as shown in [5, 29].

3 Adaptive Query Processing Families

In this section we identify the core tasks in AQP, then partition the systems in Table 1 into three families, and specify how each of them handles the core AQP tasks. Broadly speaking, the traditional plan-first execute-next approach to query processing involves three tasks as described next and also shown in Figure 2:

1. *Optimization*, in which an *optimizer* chooses a plan to execute a given query. To choose the plan, the optimizer uses statistics (e.g., table sizes, histograms) available on the input data.

2. *Execution*, in which an *executor* runs the plan chosen by the optimizer to completion to produce the query result.

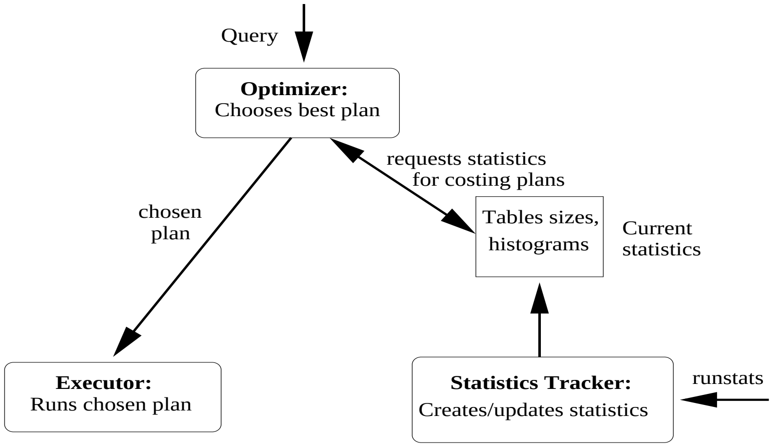

Figure 2: The traditional approach to query processing

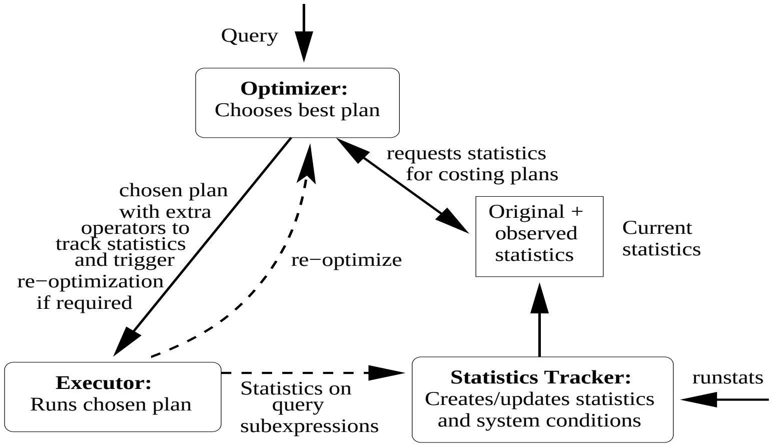
Query
Optimizer:
Chooses best plan
requests statistics for costing plans
chosen plan with extra operators to track statistics and trigger re−optimization if required
re−optimize
Original + observed statistics
Current statistics
Executor:
Runs chosen plan
Statistics on query subexpressions
Statistics Tracker:
Creates/updates statistics and system conditions
runstats

Figure 3: AQP in plan-based systems

3. *Statistics tracking*, in which a *statistics tracker* maintains the statistics used by the optimizer. Usually, a database administrator invokes the statistics tracker both to create new statistics and to bring existing ones up-to-date, e.g., using a "runstats" command as in Figure 2.

Recall from Section 1 that an AQP system may interleave optimization, execution, and statistics tracking during the processing of a query. That is, an AQP system may *re-optimize* a query, possibly multiple times, during the running time of the query. (We will hereafter use the terms AQP and re-optimization interchangeably.) Broadly, we can divide the AQP systems in Table 1 into three families, which we term *plan-based*, *routing-based*, and *CQ-based*, described next.

3.1 Plan-based Systems

Plan-based AQP systems extend the plan-first execute-next approach by monitoring the execution of the current plan and re-optimizing whenever observed plan properties (e.g., intermediate result sizes) or system conditions (e.g., available memory) differ significantly from the estimates made by the optimizer while choosing the plan. Roughly, as shown in Figure 3, this extension can be captured by extending statistics tracking to include plan properties and system conditions during query execution, along with some criteria to trigger re-optimization based on the values observed. The systems listed from Re-Opt to Redbrick in Table 1 are plan-based AQP systems.

3.2 Routing-based Systems

Routing-based AQP systems, represented by Eddies [4] and River [3] in Table 1, represent an innovative approach to AQP. They avoid a traditional optimizer, and in fact, eliminate query plans altogether. (Joins in the Ingres system were processed using a similar approach [39].) Instead, routing-based systems process queries by routing tuples through a pool of operators, ensuring that most tuples follow the most efficient routes. As shown in Figure 4, the tuple router integrates both optimization and statistics tracking using the single primitive of selective routing of tuples along alternate routes. Most tuples *exploit* the route that is most efficient currently, while the rest *explore* other routes. The Eddies architecture has been extended to support continuous queries [29], primitives for state management [18, 33], distributed processing [36], and routing based on tuple content [8].

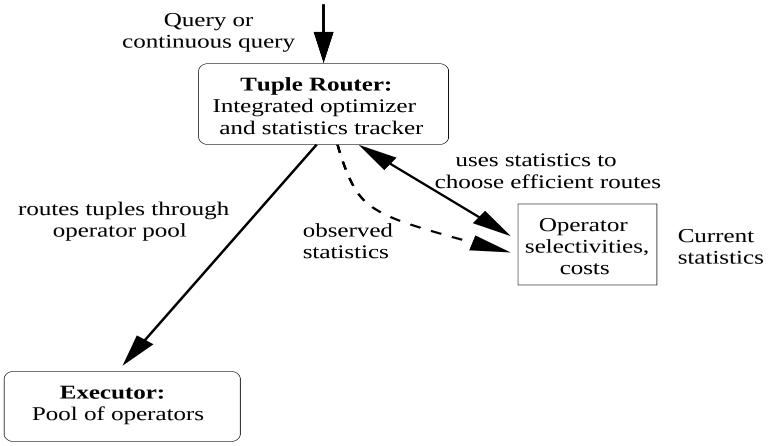

Figure 4: AQP in routing-based systems

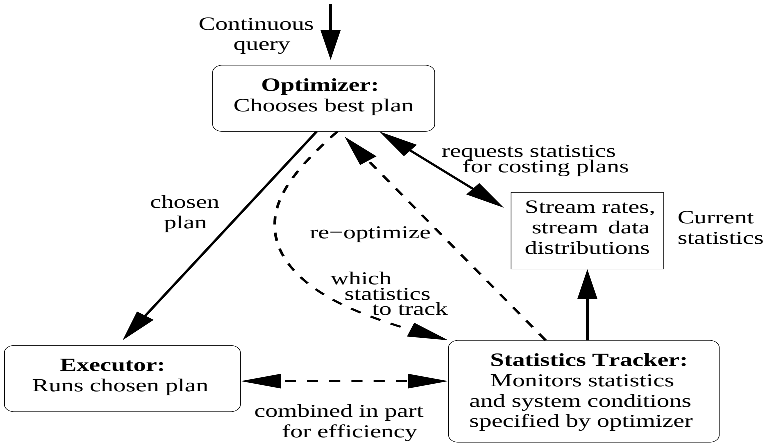

Figure 5: AQP in CQ-based systems

3.3 CQ-based Systems

Continuous-Query-based, or CQ-based, AQP systems support AQP for continuous queries, prevalent in data stream systems. (Routing-based systems also support continuous queries [12, 29].) CAPE [41], NiagaraCQ [13], and StreaMon [7] are the CQ-based systems in Table 1. AQP in CQ-based systems is shown in Figure 5. Like plan-based systems, CQ-based systems use query plans and an optimizer. However, their focus is on arbitrary changes in stream characteristics and system conditions during the running time of a continuous query, rather than mistakes in statistics for a given query plan. For example, the optimizer may inform the statistics tracker about all statistics that it used when considering alternative plans, and the tracker monitors for changes in these statistics [5]. To minimize the run-time overhead, CQ-based systems use sampling-based techniques for statistics tracking, and combine statistics tracking with query execution whenever possible [5, 6, 7].

3.4 Summary and Details

Table 2 provides a detailed summary of the three families in terms of the problem addressed, the objectives, the assumptions made, and the general approach to the core tasks of AQP.

4 Comparison of the System Families

We now present a comparison of the three AQP system families with respect to a wide range of specific issues relevant to AQP. This section is organized as follows:

- One distinctive feature of routing-based systems that sets it apart from plan-based and CQ-based systems is the absence of an optimizer. We begin by listing the pros and cons of using an optimizer in an adaptive environment (Section 4.1).
- We then discuss the techniques used by different systems to track statistics required for AQP (Section 4.2). While each system family currently uses a different technique, most techniques have broad applicability that translates across the system families.
- We then discuss techniques specific for re-optimization in the system families, specifically, knowing when to re-optimize, doing the re-optimization, and switching to a new plan if required (Section 4.3).
- We wrap up the comparison with a look at performance issues in AQP, namely, quality of re-

	Plan-based	CQ-based	Routing-based
Problem addressed	(i) Not all input statistics are known initially (ii) System conditions (e.g., memory availability) may change over time	(i) No input statistics known initially (ii) Input statistics and system conditions may change over time	(i) No input statistics known initially (ii) Input statistics and system conditions may change over time
Objectives	Detect and correct situations where optimizer picks an inefficient plan	Ensure that plan is efficient for current input statistics and system conditions	Ensure that tuples follow efficient routes
Assumptions	Given correct estimates of statistics and system conditions, optimizer can find an efficient plan	(i) Statistics required by optimizer can be estimated during execution (ii) Given correct estimates of statistics and system conditions, optimizer can find an efficient plan	Efficient routes can be found by exploring different routes
Executor	(i) Pipelined and non-pipelined plans (ii) Plans contain operators to track statistics and trigger re-optimization if required	Pipelined plans only (because operators in continuous query plans must be non-blocking [2])	Routes through operators simulate pipelined plans
Optimizer	Conventional optimizer that also adds additional operators to track statistics and trigger re-optimization if required	May use conventional optimizer or approximation algorithms that are guaranteed to find near-optimal plans	(i) No conventional optimizer (ii) Greedy optimization via selective tuple routing
Statistics Tracker	Conventional statistics along with statistics on query subexpressions collected during execution	Estimate all statistics (e.g., operator selectivities, costs) relevant to current optimization	Operator-level selectivities and costs maintained during execution

Table 2: Model, goals, assumptions, and approach of the three system families

Concern	Using an Optimizer (Plan- and CQ-based systems)	No Optimizer (Routing-based systems)
Effect on plan quality	(+) Can consider large complex plan spaces (-) Susceptible to estimation errors (optimizer mistakes)	(+) Routing is based on observed (accurate) statistics (-) Routing algorithms are usually greedy and designed for smaller, simpler plan spaces
Complexity	(-) Optimizers are very complex modules	(+) Simple primitive for re-optimization
Run-time overhead	(-) Context switching between optimizer and executor (-) Plan enumeration and costing may be invoked multiple times	(-) Overhead of routing (-) Have to ensure continuously that routes correspond to valid plans

Table 3: Advantages (“+”) and disadvantages (“-”) of using a conventional optimizer

optimization, run-time overhead, robustness to thrashing, and scalability (Section 4.4).

We use tables throughout this section to provide detailed comparisons for all the above issues.

4.1 Pros and Cons of Using an Optimizer

Recall from Section 3.2 that routing-based systems do not use conventional optimizers or query plans. Table 3 summarizes the advantages (indicated by “+”) and disadvantages (indicated by “-”) of using a full-blown optimizer in an adaptive environment.

4.2 Tracking Statistics

In Sections 3.1–3.3, we saw why different AQP systems need to track statistics during query execution. Statistics are tracked either to verify that the actual values observed during execution match optimizer estimates, or to obtain run-time values for use in future optimization steps. Some statistics can be observed directly from the execution of the plan in a plan-based or CQ-based system, or from the flow of tuples along the current best path in a routing-based system. For example, to track the cardinality of a query subexpression that happens to be materialized by the current plan, the system just counts the tuples in the materialized result.

However, we may want to track statistics that cannot be observed directly from the current plan. To do so, the system can process input data using one or more additional plans or routes apart from the current best one. This additional processing may be devoted solely to the purpose of tracking statistics, or it may be integrated into regular query processing so that no redundant work is done.

To illustrate the above points, we list four techniques used in current AQP systems for tracking statistics during query execution:

1. **Observation:** [28, 30] Observation is used commonly in plan-based systems where statistics (e.g., cardinali-

Technique	Computational Overhead	Accuracy of Estimation	Coverage of Statistics
Observation	Depends what statistics are collected	High because statistics collected are actual observations	Restricted to what can be observed from plan
Exploration	Depends on fraction of tuples sent along less efficient paths	Susceptible to bias in routing decisions and to correlations	Limited by the large number of alternative routes
Competition	Redundant processing of data	High because statistics collected are actual observations	Low, only limited number of competing plans can be run
Profiling	Extra work on a random sample of the data	(i) Depends on sampling fraction (ii) Unbiased estimates	Broad coverage of statistics based on a sample of the data (e.g., conditional selectivities can be estimated)

Table 4: Analysis of the four common techniques for tracking statistics, with respect to three key properties

ties, random samples, histograms) are collected on tuples that pass through selected points in a query plan.

2. **Exploration:** [4] Exploration is used in routing-based systems where a fraction of input tuples are routed along routes different from the current best route to track operator selectivities and costs along these routes.

3. **Competition:** [1] Competition is similar to exploration, but it routes the same set of tuples along multiple *competing* routes (or plans) in order to track operator selectivities and costs using the same input data.

4. **Profiling:** [5] Profiling differs from exploration and competition in that a fraction of tuples are processed by all operators solely to collect statistics.

Table 4 analyzes these techniques with respect to three key properties: Computational overhead, accuracy of estimation, and *coverage*. The coverage of a technique characterizes the number of different statistics that can be estimated simultaneously using that technique. While competition and profiling are intrusive techniques in the sense that they require changes to the operators to avoid generating false positives or false negatives in the query result, both observation and exploration are nonintrusive techniques. Furthermore, current implementations of exploration and profiling support pipelined plans only.

4.3 Re-optimization

Re-optimization involves three components: (i) choosing when to re-optimize, (ii) finding the new best plan or route, and (iii) switching to the new plan or route if it is different from the current one. We consider (i) and (ii) together in Section 4.3.1 and (iii) in Section 4.3.2.

4.3.1 When and How to Re-optimize

Re-optimization is invoked explicitly in plan-based and CQ-based systems, while it happens implicitly in routing-based systems. During execution, plan-based and CQ-based systems track some or all of the statistics that the optimizer estimated while picking the current plan. The optimizer is re-invoked whenever an observed value is found to be either significantly different from the optimizer's estimate or outside the range of values for which the current plan is optimal [30]. (Instead of comparing at the level of statistics, e.g., cardinalities, sometimes the comparison is done at the level of estimated plan costs, e.g., time to completion, computed from these statistics [28].) When invoked for re-optimization, the optimizer is given the new set of statistics which includes those tracked by the current plan. In a routing-based system, the scheme used to route tuples to operators is based on current statistics. For example, most tuples may be routed first to the operator with the current lowest selectivity. Thus, re-optimization happens automatically when statistics change [4].

Example 4.1 We use an example to illustrate some of the subtleties that arise in statistics tracking and re-optimization. Consider a query that contains a four-way join of relations R, S, T, and U. (Each relation could be a window over a stream in a data stream processing system, in which case the query semantics is similar to that of a materialized view [2].) Assume that $R \bowtie T$ has low selectivity, i.e., $R \bowtie T$ produces many joined tuples, and $R \bowtie U$ has high selectivity, but both these statistics are unknown. The optimal join order is $((R \bowtie U) \bowtie S) \bowtie T$. An optimizer may estimate the unknown selectivities incorrectly, and pick the join order $((R \bowtie S) \bowtie T) \bowtie U$. A plan-based system will not notice these mistakes during execution because the selectivities of $R \bowtie T$ and $R \bowtie U$ cannot be observed directly from the join order $((R \bowtie S) \bowtie T) \bowtie U$. If the system monitors the size of $R \bowtie S \bowtie T$ during execution, it will find that this size is larger than the estimated value, and trigger re-optimization. However, re-optimization may not improve the situation—the optimizer may pick a suboptimal plan again—because the system has not learned the accurate values of the selectivities of $R \bowtie T$ and $R \bowtie U$. $\square$

Concern	Plan-based	CQ-based	Routing-based
Avoiding duplicate results	(i) No result output until processing is complete (e.g., by buffering) (ii) Keep track of tuples output so far to eliminate duplicates in future	Access methods over streams are one-pass scans, so duplicate results are never generated	Routing constraints are enforced to avoid generating duplicate results [33]
Reusing work done so far	Materialized subexpressions reused on a cost basis	Migrate state in temporary structures (e.g., hash tables) to new plan [41]	Migrate state in temporary structures (e.g., hash tables) to new plan [18]
Reducing switching time	Start new plan on new input; combine data partitions processed by different plans only after all sources are exhausted [24]	(i) Caches (soft state) that can be dropped quickly and built incrementally [6] (ii) Techniques to migrate state in parallel with processing [41]	(i) Pipelined processing techniques that do not maintain extra state [33] (ii) Fine-grained primitives for tuple router to migrate state [18]

Table 5: Techniques used by AQP systems to address issues in plan switching

4.3.2 Plan Switching

Re-optimization in a plan-based or CQ-based system may decide a new plan is better than the current one. In that case, some important issues need to be dealt with when switching between plans:

- *Correctness:* The new plan must not output result tuples that have already been output by previous plans (or miss tuples), particularly in pipelined plans.
- *Reuse of work:* The current plan and plans before it may have processed a substantial part of the query. It is important to consider whether the new plan can reuse this work instead of restarting query processing from scratch. However, it may not always be beneficial to reuse work, so reuse must be considered in a cost-based manner.
- *Plan state:* Techniques for switching between plans must account for the state captured by a plan, which has four components:
  - Base relations in the query (these may be windows over streams in a continuous query)
  - Intermediate materialized subexpressions
  - In-flight tuples in pipelined segments
  - Temporary structures like hash tables and sorted sublists

Table 5 summarizes the techniques and assumptions used in the three system families to address the above issues.

During re-optimization, plan-based systems consider reusing query subexpressions materialized by previous plans to reduce the cost of plan switching. Recall that CQ-based and routing-based systems primarily deal with input that arrives continuously, e.g., data streams [10, 11, 31]. Minimizing response time is an important goal in these systems [10]. Therefore it is often more important in those systems to get the new plan up and running as quickly as possible after a decision to switch than it is to minimize the overall cost of plan switching. One technique that has been proposed is to let the new plan process new data as it arrives. Results from the combination of “old” and “new” data can be computed later, either after the entire data set has arrived [24, 25], or in an incremental fashion alongside the processing of the new data [38, 41]. Another technique is to store state only in *caches* that can be dropped at any time, and created incrementally whenever needed [6]. If plan state is limited to caches, then switching between plans is quick: drop all caches in the current plan, and start the new plan with its caches initially empty. (Caches that are identical between the old and new plans can be reused.)

4.4 Performance Issues

4.4.1 Quality of Re-optimization

The techniques used by current AQP systems to track statistics at run-time (Section 4.2), to trigger re-optimization (Section 4.3.1), and to switch between plans (Section 4.3.2) dictate their ability to (i) detect when re-optimization is needed because of a change in conditions or an incorrectly estimated statistic, (ii) find an efficient plan after the change or mistake, and (iii) switch efficiently to the new plan. Table 6 summarizes the issues related to these three aspects of re-optimization for the three system families.

4.4.2 Run-time Overhead

An AQP system incurs run-time overhead to ensure adaptivity, compared to a traditional plan-first execute next approach. The main contributors to this overhead are tracking statistics, invoking the optimizer for re-optimization, and switching between plans. The effect of these three factors depends primarily on the rate at which conditions change, and is summarized in Table 7. Note that changes in conditions in CQ-based and routing-based systems are similar to optimizer mistakes in plan-based systems because the effect is the same in both cases: The system discovers during query execution that the statistics and assumptions used to derive the current plan are not valid any more. Therefore,

Re-optimization Aspect	Plan-based	CQ-based	Routing-based
Detecting a change or mistake	(-) Can only detect mistakes that can be observed in plan	(+) Monitors all statistics, so will detect change quickly	(-) Exploration may take time to detect change
Finding an efficient plan after a change or mistake	(-) May not find best plan unless all uncertain statistics were measured at run-time	(+) Approximation algorithms guaranteed to find near-optimal plans	(-) Greedy optimization may not find best plan
Switching to the new plan	(+) Reuse of work considered during re-optimization	(+) Fast switching, e.g., because caches are used [6]	(+) Fine-grained primitives to migrate state [18]

Table 6: Advantages ("+") and disadvantages ("-") of the families with respect to the various aspects of re-optimization

Rate of Change	Plan-based	CQ-based	Routing-based
Low	(+) Very low overhead	(-) Overhead is dominated by tracking statistics	(+) Low overhead of exploring alternative paths
High	(-) May use inefficient plans because of limited re-optimization opportunities	(+) More resilient because profiling enables faster, more complete statistics tracking	(-) Inefficient routes may be used always because exploration takes time to converge

Table 7: Run-time overhead with respect to the rate at which system conditions change, or with respect to the number of unknown statistics in a plan-based system

e.g., a high rate of change in conditions in Table 7 is similar to having many unknown statistics.

4.4.3 Thrashing

We say that an AQP system is thrashing if it is spending most of its resources in adaptivity-related overhead, e.g., plan switching, and is not making enough progress in query execution. For example, if a CQ-based system adapts too rapidly, then a series of short-lived changes may make the system spend all its resources re-optimizing. A number of safeguards have been proposed to minimize thrashing in AQP systems:

1. Limiting re-optimization points, e.g., only re-optimizing at blocking operators in plans [28]
2. Limiting the number of times re-optimization can be invoked during query execution [30]
3. Setting a minimum number of tuples processed or time interval between any two invocations of re-optimization [6, 18]

4.4.4 Scalability

AQP systems need to scale in all the traditional dimensions such as data size, query complexity, number of concurrent queries, and data arrival/update rate. In addition, AQP systems must scale in the number of unknown statistics or the rate at which statistics can change. (Techniques such as parametric optimization [16, 22] and expected-cost-based optimization [15], which are alternatives to re-optimization, do not scale in the number of unknown statistics.) While we are not aware of any work that studies scalability of AQP systems, it is clear that scalability depends on the quality of re-optimization (Section 4.4.1), the run-time overhead for adaptivity (Section 4.4.2), and also on the robustness to thrashing (Section 4.4.3).

5 New Approaches to AQP

In this section we identify two fresh approaches for AQP and explain how shortcomings of existing AQP systems lead to these new approaches. The first one, which we call *proactive re-optimization*, is an AQP technique targeted at improving plan-based and CQ-based systems. The second one, which we call *plan logging*, is a technique for scaling CQ-based systems to handle large and complex plan spaces adaptively.

5.1 Proactive Re-optimization

Current plan-based systems use an optimizer to generate a plan, and then detect and respond to suboptimalities in this plan during execution. We call this approach *reactive re-optimization*. Specifically, these systems use a conventional optimizer that can produce an efficient plan for a desired cost metric using statistics about the input data, then add a post-processing phase. The post-processing phase adds *assertions* to the chosen plan to track actual statistics values during execution and verify that these values match the optimizer's estimates and are within the range of values for which the current plan is efficient [28, 30].

Reactive re-optimization is handicapped by its use of a conventional optimizer, which is blind to issues affecting re-optimization. As we will show in Sections 5.1.1–5.1.4, the benefits, costs, and flexibility of re-optimization depend strongly on the query plan used for execution. To solve this problem, we propose *proactive re-optimization* in which query plans are selected with re-optimization in mind. Like a conventional optimizer, the overall goal of a proactive optimizer is to minimize the desired performance metric, e.g., time to completion or response time. However, while choosing plans, a proactive optimizer also considers four issues relevant to re-optimization:

1. The potential overhead of tracking statistics during execution and the chances of needing re-optimization and plan switching
2. The potential for reuse of work in case a plan switch is required
3. The ability to identify quickly whether the current execution plan is suboptimal
4. The ability to decrease uncertainty in input statistics quickly during execution

Some of these issues are conflicting, which makes proactive re-optimization a challenging problem and an interesting avenue for future work. These four issues are discussed next.

5.1.1 Potential Run-time Overhead for Adaptivity

We use an example to motivate the need for considering adaptivity overhead when selecting an execution plan. Consider the join of relations R and S. An indexed nested-loop join (INLJ) with R as the outer may outperform a hybrid hash join (HHJ) if R is small, S is large, and S has a clustered index on the join attribute. As the size of R increases, the performance of HHJ quickly starts to dominate that of INLJ. Suppose the size of R is unknown. Then, it may be better to use the *safe* HHJ instead of the *risky* INLJ because using the INLJ will require the system to incur run-time overhead to track statistics on R during execution, and in the worst case, an expensive re-optimization and a plan switch will occur. However, if the size of R is known with high confidence to be small, then we prefer INLJ to HHJ.

This simple example shows how a proactive optimizer can explore the tradeoff between plan safety and the risk of incurring higher run-time overhead, based on knowledge of uncertainty in statistics. (Technically, plan safety may be specified in terms of the *expected cost* of the plan [15].) To achieve this goal, a proactive optimizer needs to take into account uncertainties in statistics. Some initial work in this direction was done by Re-Opt [28] which quantified the uncertainty in a statistic using a discrete-valued categorization.

5.1.2 Potential Reuse of Work

In the previous section, we saw an example of a plan P that is risky because the chances of triggering re-optimization are high if P is used. A plan P may also be risky because the system may not be able to reuse any of the work done by P if re-optimization and a plan switch is required. In general, it is hard to keep track of and reuse the work done by a pipelined plan if the query needs to be re-optimized during the execution of the pipeline. Therefore, one simple approach that a proactive optimizer can use to increase the potential for reusing work is to generate plans with shorter pipelined segments and more materialization points.

5.1.3 Identifying Plan Suboptimality Faster

Recall that re-optimization is triggered by the violation of an assertion about statistics in the current plan during execution. Some plans allow for more assertions to be checked concurrently, or to be checked sooner, thereby enabling faster detection and correction of suboptimality. For example, in a fully pipelined plan, we can maintain running selectivities for all operators in the plan. So, the system can discover suboptimalities involving unknown selectivities of downstream operators in the plan long before the upstream operators have finished execution. Note that the goal of picking a plan that enables suboptimality to be detected quickly may conflict with the goal of maximizing potential reuse of work.

5.1.4 Decreasing Uncertainty in Statistics

A proactive optimizer may want to reduce the uncertainty in some statistics as quickly as possible, even at the cost of delaying query execution. For example, consider the join of relations R and S from Section 5.1.1, and suppose that the uncertainty in the size of R comes from the presence of selection predicates p_1 and p_2 on R that may be correlated. In this situation, the optimizer can choose to first estimate the combined selectivity of p_1 and p_2 from a random sample of R before choosing the join algorithm to use for $R \bowtie S$. An interesting avenue for work would be to merge such random sample processing alongside regular query execution to hide the extra overhead. For example, if R tuples are stored in random order on disk, then after 5% of R tuples have been scanned as part of a table scan, a fairly accurate estimate of the combined selectivity of p_1 and p_2 can be obtained. Furthermore, the scan may not need to be restarted if the selected R tuples in the sample are buffered.

Other techniques that a proactive optimizer can use to decrease the uncertainty in input statistics quickly are: it can choose a pipelined plan for some queries, and use profiling (recall Section 4.2) to estimate required statistics; it can explore multiple selected subplans to collect statistics on query subexpressions.

5.2 Plan Logging

Most current CQ-based and routing-based systems adapt execution only within Select-Project-Join blocks [5, 7, 18]. It is important to scale these systems to handle larger and more complex queries and plan spaces adaptively: to simultaneously consider access methods and plan shapes, memory allocation to operators, parallelism, and sharing of data and computation. However, run-time overhead to support adaptive processing increases significantly as plan spaces expand: the optimizer must search through the larger plan space, more statistics need to be tracked or more routes need to be explored, and plan-switching costs increase because of increased plan state. Furthermore, larger plan-switching costs exaggerate the effect of thrashing if the system reacts rapidly to changes. In this section, we outline *plan-logging*, a promising direction to attack these problems in a CQ-based system.

With plan-logging for a continuous query Q, the statistics tracker continuously logs the statistics relevant to Q that it collects, and the optimizer logs the corresponding plan that it picked for Q based on these statistics. For example, entries in the log for query Q may have the form $\langle t, S, P \rangle$ indicating that the optimizer picked plan P for Q based on statistics S observed at time t.

Over time, the entries for Q in the log capture the path traversed by the system in the space of statistics relevant to Q. That is, we can consider a high-dimensional space containing a dimension for each independent input statistic relevant to Q, e.g., the rate of each input stream in Q. For each point S in the space of statistics relevant to Q, there is a plan P that is optimal for Q with respect to the desired cost metric, e.g., query completion time. (If the properties of the input data are as specified by S, then plan P has the minimum cost among all plans for Q.) The information in the log for Q, which captures the space of statistics relevant to Q, can be used in an AQP system as follows:

- By grouping together log entries that contain the same plan P for query Q, the AQP system can identify regions in the space of statistics relevant to Q where P is efficient. This information can be used to pick plans without full optimization when Q is re-optimized under statistical conditions that fall in regions seen previously.
- The AQP system can identify those statistics whose changes most contribute to significant changes in plan performance. This information can be used to reduce run-time overhead by reducing the rate at which “unimportant” statistics are tracked.
- The history captured by the log can be used to do an online *what-if* analysis. For example, the AQP system can identify the performance that it could have achieved without adaptivity, e.g., the average performance of the single best plan over a time interval. This information can be used to track the *return-of-investment* on adaptive processing, so the system can control the run-time resources allocated to maintain adaptivity.
- The AQP system can identify statistics that are prone to transient changes, so that it can reduce chances of thrashing on such changes.

Note that plan logging and proactive re-optimization are orthogonal and compatible ideas, so the best approach may be to do both.

6 Conclusions

Our main contribution in this paper is the identification of three families of AQP systems (plan-based, CQ-based, and routing-based), and the detailed comparison of these system families with respect to the most important aspects of AQP: plan quality, statistics monitoring and re-optimization, plan migration, and scalability. We hope that our work is the first step toward answering some important questions about AQP at this point in its evolution.

- Given some fundamental differences among the systems that support AQP, will adaptive techniques developed for one system be useful on another system?
- AQP considers many aspects of a physical plan, e.g., access methods, plan shapes, memory allocation, and scheduling of operators. Can we quantify the benefits of AQP with respect to each of these aspects? For example, which of these aspects would degrade performance most if it is not considered for adaptive optimization?
- Each AQP system in Table 1 has its applicable domain, i.e., the data and query workloads for which it is the most effective compared to the plan-first execute-next approach. Can we build an AQP system that can perform well in all domains? If not, are there fundamental tradeoffs among the domains that prevents us from doing so?
- Are there data and query workloads for which no AQP system listed in Table 1 works well? If so, how can we build AQP systems that are effective for these workloads?

7 Acknowledgments

We are extremely grateful to Jennifer Widom, David DeWitt, Kamesh Munagala, and Rajeev Motwani for helpful feedback and discussions.

References

[1] G. Antoshenkov and M. Ziauddin. Query processing and optimization in oracle rdb. *VLDB Journal*, 5(4):229–237, 1996.

[2] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. *VLDB Journal*. (To appear).

[3] R. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. on Computer Systems*, 21(1):36–86, 2003.

[4] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[5] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 407–418, June 2004.

[6] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *Proc. of the 2005 Intl. Conf. on Data Engineering*, 2005. (To appear).

[7] S. Babu and J. Widom. StreaMon: An adaptive engine for stream query processing. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, June 2004. Demonstration proposal.

[8] P. Bizarro, S. Babu, D. DeWitt, and J. Widom. Content-based routing for continuous query-optimization. Technical Report 1511, University of Wisconsin, Madison, Apr. 2004.

[9] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. Dynamic query scheduling in data integration systems. In *Proc. of the 2000 Intl. Conf. on Data Engineering*, pages 425 – 434, 2000.

[10] D. Carney et al. Monitoring streams–a new class of data management applications. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, Aug. 2002.

[11] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. First Biennial Conf. on Innovative Data Systems Research*, Jan. 2003.

[12] S. Chandrasekaran and M. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB Journal*, 12(2):140–156, 2003.

[13] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[14] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. on Database Systems*, 9(2):163–186, 1984.

[15] F. Chu, J. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *Proc. of the 1999 ACM Symp. on Principles of Database Systems*, pages 138–147, 1999.

[16] R. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 150–160, 1994.

[17] B. Dageville and M. Zait. SQL memory management in Oracle9i. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, pages 962–973, Aug. 2002.

[18] A. Deshpande and J. Hellerstein. Lifting the burden of history from adpative query processing. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, Aug. 2004.

[19] S. Ganguly. Design and analysis of parametric query optimization algorithms. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 228–238, Aug. 1998.

[20] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the 1989 ACM SIGMOD Intl. Conf. on Management of Data*, pages 358–366, 1989.

[21] J. Hellerstein, M. J. Franklin, et al. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[22] A. Hulgeri and S. Sudarshan. AniPQO: Almost non-intrusive parametric query optimization for nonlinear cost functions. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 766–777, Aug. 2003.

[23] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis. Parametric query optimization. In *Proc. of the 1992 Intl. Conf. on Very Large Data Bases*, pages 103–114, Aug. 1992.

[24] Z. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, Seattle, WA, USA, Aug. 2002.

[25] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–310, June 1999.

[26] Z. Ives, A. Halevy, and D. Weld. Adapting to source properties in processing data integration queries. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 395 – 406, 2004.

[27] Z. Ives, A. Levy, et al. Adaptive query processing for internet applications. *IEEE Data Engineering Bulletin*, 23(2):19–26, June 2000.

[28] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 106–117, June 1998.

[29] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, June 2002.

[30] V. Markl, V. Raman, D. Simmen, G. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 659–670, June 2004.

[31] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, Jan. 2003.

[32] K. Ng, Z. Wang, R. Muntz, and S. Nittel. Dynamic query re-optimization. In *Proc. of the 1999 Intl. Conf. on Scientific and Statistical Database Management*, pages 264–273, July 1999.

[33] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, Mar. 2003.

[34] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, June 1979.

[35] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 9–28, Sept. 2001.

[36] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, Sept. 2003.

[37] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, Sept. 2001.

[38] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 130–141, June 1998.

[39] E. Wong and K. Youssefi. Decomposition - a strategy for query processing. *ACM Trans. on Database Systems*, 1(3), 1976.

[40] V. Zadorozhny, L. Raschid, M. Vidal, T. Urhan, and L. Bright. Efficient evaluation of queries in a mediator for websources. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 85–96, 2002.

[41] Y. Zhu, E. Rundensteiner, and G. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 431–442, 2004.