Bulletin of the Technical Committee on

Data Engineering

June 2000 Vol. 23 No. 2

IEEE Computer Society

Letters

Special Issue on Adaptive Query Processing

Announcements and Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

The Data Engineering Bulletin web page is http://www.research.microsoft.com/research/db/debull.

Letter from the Editor-in-Chief

Changing Bulletin Editors

The Bulletin practice is to appoint editors for two years. Each editor is responsible for producing two issues, one per year, during that time. The last of the four editors of the previous two year cycle have now completed their issues and are due to "retire". I want to thank Amr El Abaddi and Elke Rundensteiner for doing a very capable job and producing high quality issues for Bulletin readers. I'd like to be sure that all readers are aware that each issue is, in fact, a substantial undertaking. We owe a debt to Amr and Elke for their fine issues and their hard work.

To complete the appointment of new editors for the next two years, I am delighted to announce that Luis Gravano and Gerhard Weikum have accepted my invitation to be Bulletin editors.

Luis Gravano is a faculty member in the computer science department at Columbia University. Luis has done some really inovative work in query processing and data mining, including querying over the web, and across heterogenous data collections, involving multimedia and textual as well as tabular data. Luis received his Ph.d from Stanford.

Gerhard Weikum is on the faculty at the University of the Saarlands in Germany. Gerhard received his doctoral degree from ETH in Zurich and worked for a number of years at MCC in Austin. He is currently both a TODS and a VLDB Journal editor. Gerhard has worked on a wide range of research topics, transactions, where he invented multi-level transactions, query processing, performance analysis, and recovery, on which he and I have collaborated.

About the Current Issue

Query processing is a subject that is as old as our database research community. But it is an area which continues to stir intense interest. One fundamental problem is that we never have enough information at the time that query optimization is done to do the best possible job. This has led, over a number of years now, to exploration of "dynamic" or "adaptive" query processing, in which the query plan attempts to take into consideration information discovered in the process of executing the query.

The editor for this issue is Alon Levy, who has assembled articles with a span from current industrial practice to cutting-edge research. The query setting varies from the traditional to heterogeneous data sources to web centric ones, from theory to architecture to operational system to reflections upon prior work. This provides readers with a sense of where the field is now, where it is going, where it has been, and why. I want to thank Alon for his very successful effort in bringing this issue to fruition.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Adaptive query processing has been a subject of research from the early days of query processing, but has only recently received significant attention in the literature. Techniques for adaptive query processing address the problem that the query optimizer may not, at compile time, have good selectivity estimates, knowledge of the run-time bindings in the query, or knowledge of the available system resources. As a result, a static query plan produced by the optimizer may be far from optimal. The need for adaptive query processing is even greater in novel data management tasks where systems are processing queries over multiple, autonomous data sources that may be on a wide-area network. In this context we have even less statistics about the data sources (which may be changing without central control), and we witness unpredictable data transfer rates over the network. Broadly speaking, adaptive query processing focuses on techniques for changing the execution plan at run-time, re-optimizing the query when necessary, and design of novel operators that deal more flexibly with unpredictable conditions. The papers in this issue describe the current state of the art in adaptive query processing and outline the future challenges that need to be addressed in this realm.

The first three papers present overviews of different areas of adaptive query processing. In the first paper, Graefe discusses some of the lessons learned from using adaptive query processing techniques in commercial database systems, and he suggests future areas of focus in this context. In the second paper, Hellerstein et al. provide an overview of techniques for adaptive query processing, comparing them by their granularities of adaptivity (e.g., inter-operator, intra-operator, per tuple). The authors also describe the adaptive query processing techniques used in the Telegraph project at UC Berkeley. The paper by Ives et al. points discusses issues that affect the context in which adaptive query processing is employed, such as the underlying data model, the number of queries expected, and whether the data is streaming. The paper also describes the adaptive query processing techniques developed in the Tukwila System at the University of Washington.

The second set of papers describe specific techniques for adaptive query processing. The paper by Urhan and Franklin describes XJoin which is an extension of pipelined hash joins to handle out-of-memory situations when the source data is bursty. The paper by Cole describes the use of decision theory for evaluating the utility of interleaving query planning and execution. Finally, the paper by Bouganim et al. describes how the architecture of a query processor needs to be modified in order to accommodate adaptive query processing.

Clearly, the area of adaptive query processing is likely to receive significant attention in the upcoming years. I hope this issue will clarify the state of the art in this field and foster more research on this exciting topic.

Alon Levy
University of Washington

Dynamic Query Evaluation Plans: Some Course Corrections?

Goetz Graefe
SQL Server Development
Microsoft Corporation
goetzg@microsoft.com

Purpose: In database query processing, optimization is commonly regarded as "the hard" part, whether in relational, post-relational, object-oriented, textual-spatial-temporal, federated or web-based database systems. Query execution, on the other hand, is considered a mostly straightforward exercise in algorithm implementation, with the currently "hot" twist to consider CPU caches. There is, however, a third piece to the puzzle, namely physical database design, e.g., the set of useful indexes. The purpose of this brief paper is to tie these three pieces together and to encourage students and researchers to adopt a broader perspective of adaptive query processing. Another purpose is to present some contrarian viewpoints about interesting and worthwhile research topics.

Compilation effort: It is well known that compilation and early binding are good ideas, where they can be applied. Thus, compile-time optimization wins over interpretation, in particular for repetitive query execution, which typically is the bulk of database activity. On the other hand, query optimization based on incomplete information often results in plans that perform poorly in many invocations. The Achilles heel of query optimization is selectivity estimation; thus, missing statistics such as histograms and counts of unique values are a perennial worry for database administrators, unless the DBMS automatically creates, refreshes, and drops statistics as appropriate. The other important source of selectivity estimation errors is lacking compile-time knowledge about run-time parameter values. Traditionally, the only remedy has been run-time optimization. Early research into dynamic or adaptive plans was motivated to address this issue. The basic theme was that the optimized query plan contains, in some reasonably compact form, multiple alternative execution plans, and selects among them at start-up time or even at some few carefully pre-planned points during execution. It turned out, not surprisingly, that execution primitives were relatively easy to design, whereas building effective optimizers for dynamic plans is still much more art than science, and could greatly benefit from more research into practical techniques, which then would greatly facilitate technology transfer into products. After all, the point of dynamic plans is to avoid the effort for frequent re-compilation and re-optimization; if the effort for finding a single dynamic plan is just as large as or even larger than the effort for repeated re-compilation, not much is gained, other than perhaps start-up latency.

Cost estimation and dynamic execution techniques: A lesser source of query optimization errors is fluctuating resource availability and therefore inaccurate cost calculations. For example, the relative costs of index-to-index-and-record-to-record navigation compared to set-oriented sort- and hash-based query plans depends on the available memory, available CPU processing bandwidth, available disk bandwidth for permanent and temporary

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

files and tables, etc. Consequently, there have been some, albeit not very many, proposals to address changes in resource availability using dynamic query evaluation plans. However, rather than using dynamic plans with alternative algorithms or alternative plan shapes, more typically resource availability is left to individual algorithms. Thus, there are many proposals and literature on dealing with resources within a single operator or algorithm, in particular memory resources, e.g., hybrid hash join, dynamic destaging, merge optimizations in external merge sort, etc. An interesting algorithm for data-driven join processing in memory-rich parallel and distributed systems is the symmetric hash join using two hash tables. Rather few papers, however, deal with memory management among multiple operators, in nested queries, or in dynamic query plans. In modern commercial query execution engines, there are many adaptive techniques, not only within operators but also among operators within a single query and among multiple concurrent queries. Examples include setting or adjusting the degree of parallelism depending on current load, reordering and merging ranges to optimize repeated probes into an index, sharing scans among multiple queries and user connections, ignoring read-ahead hints if most are buffer hits and therefore pointless, "bailing out" of hash partitioning and resorting to loops- or merge-based algorithms when partitioning is not effective (e.g., due to duplicate key values), etc. In spite of this wealth of adaptability, all of these adaptive techniques are typically ignored in the cost functions of commercial query optimizers, partially because they are difficult to incorporate and partially because a sufficiently strong case for incorporating them has not yet been made. What does that say about techniques as adaptive as dynamic query evaluation plans?

Nested iteration: While most resource issues have relatively little impact on optimal plan choices (even if they affect the ranking among different plans of fairly similar costs), one issue that is crucial in practice but usually ignored in academic research is the effect of buffer hits and faults in complex query plans. In particular nested SQL queries as well as nested iteration as operator in both the logical and physical query algebra have been usually neglected in research. Interesting issues arise if multiple invocations of the same nested computation affect each other, e.g., the first invocation warms up the I/O buffer for subsequent ones. Other interesting issues arise if different nested computations compete for resources, e.g., I/O buffer or memory for sort and hash operations within inner queries. Nested computations are very important in practice, both because queries are authored using nested SQL queries and because nested iteration based on index-to-index navigation often is the best execution plan. Therefore, nested computations could be a very fruitful research topic, both execution and optimization, and could probably also benefit from more dynamic and adaptive techniques than those in use today. However, we will need a conceptual model of nested queries that is substantially simpler that nested SQL, e.g., based on algebra expressions with a table of parameter values.

Indexed views: Perhaps the most exciting recent development in database query processing has been the commercial use of materialized views. In a way, materialized views take the concept of early binding a step further than compile-time optimization. The benefit can be tremendous, as many OLAP tools and applications demonstrate every day with sub-second response times even for very large volumes of detail data. In effect, when materialized views work really effectively, complex query processing is reduced to index navigation very similar to OLTP processing, except for differences in the update load. Fancy techniques for large queries, e.g., shared scans, bitmap indexes, hash joins, and parallel query processing might once again be of little importance, just as they were when databases were used only for business operations (OLTP), not data analysis and business intelligence. There are many issues with respect to building redundant indexes on materialized views as well as to coherency, updates, invalidation, re-computation, incremental updates, etc.; probably the simplest policy (certainly from the perspectives of the application developer and the end user) is to treat materializations of view results similar to indexes, meaning instant updates within the original transaction, or even simply to index views in addition to tables, supporting both non-clustered and clustered indexes (the latter contain all columns in the table or the view). By doing so, the semantics of views is unchanged whether they are materialized and indexed or not; thus, indexing a view is (correctly) purely an issue of physical database design and performance, not one of logical database design and correctness.

Caching recent queries: Another exciting technology that is only starting to take off is closely related to materialized views: caching the results of recent actual queries in order to exploit them in future queries. Clearly, some OLAP tools achieve amazing performance using this technique today already, in particular if it is used both on the server and on the desktop, i.e., both for local and distributed queries. The issues with respect to indexing as well as invalidation or incremental maintenance are the same as for materialized (or indexed) views. The hard part, as with materialized views, is the policy deciding which query result to keep and to maintain for how long.

Performance metrics: The goal of indexing and result caching is to improve system performance. However, there are different definitions of performance. An excellent illustration can be found in the arguments going back and forth whether the overall performance in the TPC-D benchmark should be based on the arithmetic or geometric mean of the individual query times. For example, is it better to improve one query from 10 seconds to 1 second (9 seconds difference or factor 10) or to improve another query from 10 minutes to 5 minutes (300 seconds difference but only a factor of 2)? A more important performance goal than high performance, however, is consistent and predictable performance. The vast majority of database servers in production is running on CPUs much slower than the fastest available CPUs, often by a factor 2. Given that, clearly top performance is not the top priority; if it were, CPUs would be replaced much more rapidly. In other words, predictability versus risk is a more important dimension than fast versus slow, within the limits of common sense.

Predictable performance: Predictable performance has several aspects. One of them is that the same request or very similar requests always take the same time. For example, once a user has seen a certain query complete in 2 minutes, it is a source of great annoyance (and support calls) if that query usually takes 5 minutes. One lesson from this observation is that parallel query processing might be a very bad idea indeed, because it may well lead to unfulfilled expectations. Another aspect of predictability is that a "regular" person must be able to predict the performance. In other words, fancy adaptive techniques that make a chosen dynamic query plan hard to predict or to understand are a disservice to developers and users, unless these adaptive techniques work so well that developers and users never have much interest in looking at query plans, just as little as they have nowadays in understanding the inner workings of virtual memory.

Index tuning: Several database vendors offer (or include in their main database products) tools for index tuning, including materialized views and their indexes. These tools perform a complex analysis of the workload and can create indexes as well as drop indexes due to update costs or to space constraints. Clearly, the current versions of these tools have left room for improvement, in particular in the expense of the analysis and in the immediacy, i.e., how fast they adapt a physical database design to an initial workload and to changes in the workload. Moreover, most of these tools are based on the assumptions that database tuning is an activity that is separate from processing requests, requires a trained individual such as a DBA, and takes a substantial amount of system effort, e.g., for analyzing query plans for various proposed index configurations. Due to their computational overhead, they typically do not lend themselves to complete automation.

Indexing tuning heuristics: It turns out, however, that fairly simple indexing schemes often work amazingly well. For example, clustered indexes on primary keys and non-clustered indexes on foreign keys are a pretty good starting point. Since they rarely change and are important in many applications, non-clustered indexes on all columns with "date" or "time" data types often help. Finally, columns that appear in "=" predicates probably benefit from indexes, which actually would include indexes on primary and foreign keys if not already indexed. Indexes on "in" predicates as well as selective "between" predicates typically are worthwhile, too. To refine these rules, one can add the primary key and all foreign keys to all indexes, just to make index-to-index navigation faster. Finally, one can avoid any non-clustered indexes on very small tables, say less than a page, and avoid using more space for indexes than for the main data.

Queries building indexes: With these simple indexing rules, many query sets, e.g., TPC-D, work pretty well (definitely for query processors supporting index intersection) – certainly within the same factor of 2 many people are willing to forgo by not keeping up with the latest hardware. An interesting question is whether these indexes can be created quickly and quietly. For example, if an equality predicate is found that is not supported by an existing index, how can one be built with minimal overhead? A separate index creation has to tread lightly, e.g., it must not "hog" the CPU or the disk, and it must not prevent concurrent user activity. In other words, online index creation is just as important a feature for fully automated "low-end" data management systems as for carefully monitored high-end 24x7 systems that strive for "five nines" or better (99.999% availability, or 5 minutes down time per year, including planned down time). If, on the other hand, an equality predicate without suitable index results in a full scan, can we design adaptive plans that scan, sort, merge join, and leave behind the appropriate index for the next execution? Can that still work even if there are concurrent updates that the index must reflect but the query result must not? Can the new index be fully exploited when the same query is run again, without re-compilation and re-optimization, by means of a dynamic query plan that exploits an index if it exists or creates one if it doesn't?

Heuristics for indexing views: While fairly obvious and immediate rules might give satisfactory results for indexes on tables, this is not the case for materialized (indexed) views. There are some very effective algorithms and heuristics for pre-aggregations, both in the literature and in products, but many queries used in data analysis, in particular if dimensions and hierarchies are used, can benefit not only from pre-aggregations but also from joins, outer joins, nested queries, semi joins, set operations, etc. Crystallizing simple and effective heuristics and their integration into query processing could substantially benefit the use of database systems, in particular relational database systems, in data analysis and data mining.

Research for orders of magnitude: Finally, a brief word of caution. A performance- or throughput-oriented software technique that results in a performance improvement measured as a percentage might very well be important for a vendor under pressure to win today's benchmark, but it isn't a worthy research goal or result. An improvement measured by a small factor (say factor 3) is laudable and useful, but not a breakthrough – improvements in hardware technology will give us the same improvement (fairly predictably!) in just one or two years, the time it takes to publish a journal paper about the new software technique or to ship it in a product. A typical example for small-factor performance improvement is research into algorithms exploiting CPU caches, typically by adapting techniques proven for disks and disk pages, e.g., B-trees, external sorting, and horizontal and vertical partitioning. In order to be truly a breakthrough, a performance improvement has to be measured in orders of magnitude. Materialized views are one such technique. Dynamic query plans, on the other hand, so far have not achieved this level of success on a broad scale. Is it possible to achieve it? Can we achieve consistent and predictable orders-of-magnitude performance improvements for relational and post-relational database systems by combining dynamic query plans with on-the-fly indexing and materialized views?

Adaptive Query Processing: Technology in Evolution

Joseph M. Hellerstein Michael J. Franklin Sirish Chandrasekaran Amol Deshpande
Kris Hildrum Sam Madden Vijayshankar Raman Mehul A. Shah

Abstract

As query engines are scaled and federated, they must cope with highly unpredictable and changeable environments. In the Telegraph project, we are attempting to architect and implement a continuously adaptive query engine suitable for global-area systems, massive parallelism, and sensor networks. To set the stage for our research, we present a survey of prior work on adaptive query processing, focusing on three characterizations of adaptivity: the frequency of adaptivity, the effects of adaptivity, and the extent of adaptivity. Given this survey, we sketch directions for research in the Telegraph project.

1 Introduction

Adaptivity has been an inherent – though largely latent – aspect of database research for the last three decades. Codd's vision of data independence was predicated on the development of systems that could adapt gracefully and opaquely to changing data and data structures. Query optimization, with its attendant technologies for cost estimation, served as an early differentiator between DBMSs and other computer systems. This tradition of dynamic, statistically-driven system optimization remains one of the crown jewels of the database systems research community.

In the last few years, broad sectors of computer science have been exploring the design of systems that are adaptive to their environment. As computer systems scale up and federate, traditional techniques for system management and performance tuning must become more loosely structured and more aggressively adaptive. In the context of database systems, this stretches the traditional techniques for adaptive query processing to the breaking point, since very large-scale query engines operate in unpredictable and changeable environments. This unpredictability is endemic in large-scale systems, because of increased complexity in a number of dimensions [AH00]:

Hardware and Workload Complexity: In wide-area environments, variabilities are commonly observable in the bursty performance of servers and networks [UFA98]. These systems often serve large communities of users whose aggregate behavior can be hard to predict, and the hardware mix in the wide area is quite heterogeneous. Large clusters of "shared-nothing" computers can exhibit similar performance variations, due to a mix of user requests and heterogeneous hardware evolution. Even in totally homogeneous environments, hardware performance can be unpredictable: for example, the outer tracks of a disk can exhibit almost twice the bandwidth of inner tracks [Met97].

Data Complexity: Selectivity estimation for static alphanumeric data sets is fairly well understood, and there has been initial work on estimating statistical properties of static sets of data with complex types [Aok99] and methods [BO99]. But federated data often comes without any statistical summaries, and complex non-alphanumeric

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Frequency of Adaptivity				
Batch	Per Query	Inter-Operator	Intra-Operator	Per Tuple
System R [SAC+79], Late Binding [HP88, GW89] [INSS92, GC94] [ACPS96, LP97]	ASE [CR94], Sampling [BDF+97], Mariposa [SAP+96],	Query Scrambling [AFTU96, UFA98], Reoptimization [KD98], Tukwila [IFF+99]	Ingres [SWK76], RDB [AZ96] WHIRL [Coh98]	River [AAT+99], Eddies [AH00]

Table 1: Prior work on adaptive query processing, ordered by frequency of adaptivity. We omit specific adaptive operators from the table, including sorting and joins as they are inherently intra-operator in frequency. We discuss them further in Section 2.7.

data types are now widely in use both in object-relational databases and on the web. In these scenarios – and even in traditional static relational databases – selectivity estimates are often quite inaccurate.

User Interface Complexity: In large-scale systems, many queries can run for a very long time. As a result, there is interest in Online Aggregation and other techniques that allow users to "Control" properties of queries while they execute, based on refining approximate results [HAC+99].

1.1 Telegraph in Context

The goal of the Telegraph project at Berkeley is to architect and build a global-scale query engine that can execute complex queries over all the data available online[1]. In designing Telegraph, we are facing new challenges in wide-area systems, including the quickly shifting performance and availability of resources on the Internet. We also want Telegraph to serve as a plug-and-play shared-nothing parallel query engine, which must be trivial to manage and scale incrementally. Finally, we want to support sensor networks (e.g., [EGH99, KKP99, KRB99]), which are emerging as an increasingly important – and unpredictable – aspect of "ubiquitous" ("calm", "post-PC") computing.

Key to Telegraph is a continuously adaptive query processing engine that can gather and act upon feedback at a very high frequency. This high-frequency adaptivity opens up both new opportunities and challenges in our research agenda. In this paper we outline our view of earlier advances in adaptive query processing research, which set the stage for the challenges we are addressing in building Telegraph. Table 1 provides a brief overview of the work we survey.

1.2 Framework

Unfortunately, the phrase "adaptive system" is not canonical. These systems are sometimes refered to as "dynamic" or "self-tuning" systems, or systems that change their behavior via "learning", "introspection", and so on. There are many tie-ins to work in related fields, such as control theory [Son98] or machine learning [Mit97]. To avoid confusion in terminology, we present our own definition of adaptivity in this context. We will call a query processing system *adaptive* if it has three characteristics: (1) it receives information from its environment, (2) it uses this information to determine its behavior, and (3) this process iterates over time, generating a feedback loop between environment and behavior

We wish to stress a distinction between adaptive systems, and systems that do static optimization. Static optimization contains the first two of these characteristics, but not the third. The feedback involved in an adapative

[1] The name refers to Telegraph Avenue, the volatile and eclectic main street of Berkeley.

system is key to its efficacy: it allows the system to make – and observe the results of – multiple decisions. This allows it to consider its own effects on the environment in concert with external factors, and to observe the effects of "exploiting" previously beneficial behavior, and "exploring" alternative behavior [Mit97].

Based on this definition of adaptivity, we can isolate three distinguishing features of an adaptive system: (1) the *frequency* of adaptivity (how often it can receive information and change behavior), (2) the *effects* of adaptivity (what behavior it can change), and (3) the *extent* of adaptivity (how long the feedback loop is maintained). As we survey prior work on adaptive query processing, we attempt to address all three of these issues.

2 A Survey of Adaptive Query Processing

Without embracing historical relativism too broadly, we admit that our survey says as much about our research agenda as about the history we present. Indeed, our goal in this short paper is to place our research agenda for adaptive query processing in the context of our view of the promise and shortcomings of the research to date.

We organize our survey partly chronologically, and partly by increasing frequency of adaptivity. As is clear from Table 1, the two orderings are roughly correlated, with some interesting exceptions. The progression toward increasing frequency over time seems to be natural, as systems are designed for increasingly complex and changeable environments.

2.1 Early Relational Systems

From the very first prototypes, relational query processors have incorporated some minimal notion of adaptivity, typically to capture the changing distributions of data in the database, and use that to model subquery result sizes, and hence query operator costs.

The System R query optimizer [SAC$^+$79] (which inspired essentially all serious commercial DBMS implementations today) kept a catalog of statistics, including cardinalities of tables and coarse distributions of values within columns. By our definition, System R was not exactly adaptive: it did not have any explicit feedback within the system. On the other hand, the system administrator could manually direct System R to adapt its behavior to the data: on command, the system would scan the entire database and update its statistics, which upon completion would instantly affect all decisions made in query optimization. This heavyweight, periodic *batch adaptivity* approach remains in nearly all commercial DBMSs today, though of course the statistics gathered – and the means for gathering them – have become more sophisticated over time. While the frequency of adaptivity is quite low in these systems (statistics are typically updated once a day or once a week), the effects are broad and the extent far-reaching: as a result of new statistics, the optimizer may choose completely different access methods, join orders and join algorithms for all subsequent queries.

The Ingres "query decomposition" scheme [SWK76] was less effective but much more adaptive than System R. Ingres alternated between subplan selection and execution. For a query on n tables, it operated as a greedy but adaptive sequence of nested-loops joins. The smallest table[2] was chosen to be scanned first. For each tuple of the smallest table, the next-smallest table was probed for matches via the "one-variable query processor" (i.e. the table-scan or index lookup module), and a (duplicate-free) result of that probe was materialized. This greedy process was then recursively applied on the remaining $n-1$ tables – i.e., $n-2$ base tables and one materialized sub-result. After the recursion unwound, the process began again for the next tuple of the smallest table. Note that this does not correspond to a static "join order" in the sense of System R: the materialized result of each "tuple probe" could vary in size depending on the number of matches to each individual tuple, and hence the order of joins in the recursive call could change from tuple to tuple of the smallest table. Although this greedy optimization process often resulted in inefficient processing in traditional environments, it had a relatively high frequency of adaptivity, gathering feedback after each call to the one-variable query processor. The frequency of adaptivity

[2] Actually, the smallest table after single-table predicates were applied and results materialized ("one-variable detachment").

in Ingres was thus *intra-query*, and even *intra-operator*, in the sense that adaptation could happened from tuple to tuple of the outermost table in a sequence of nested loops joins. This remained one of the highest-frequency schemes in the literature until quite recently. The feedback gathered after each table-scan or index lookup had the effect of adapting the join *order* on subsequent iterations. Note that the effects of adaptivity in Ingres extended only across the lifetime of a single query.

2.2 Late Binding Schemes

The schemes we discuss next are in some sense no more adaptive than that of System R, gathering no more information than is available to a standard optimizer. However, these schemes have a flavor of dynamism, and are often discussed in the context of adaptive query processing, so for completeness and clarification we cover them here.

The focus of this body of work is to improve upon a particular feature of System R's optimizer for frequently re-executed queries. In addition to its other features, System R introduced the ability for queries to be optimized, compiled into machine code, and stored with the system for subsequent reuse. This technique, which is available in commercial RDBMSs today, allows the cost of query optimization to be amortized across multiple executions of the same query, even if the query's constants are left unbound until runtime.

In the late 1980's and early 90's, a number of papers addressed a weakness in this scheme: subsequent runs of the query occur under different easily-checked runtime parameters, including changes in user-specified constants that affect selectivity, changes in available memory, and so on [HP88, GW89, INSS92, GC94, ACPS96, LP97]. Query execution performance might be compromised under such changes, but the cost of complete reoptimization on each small perturbation of the environment could be wasteful. To strike a happier medium, these systems do some optimization in advance, and need to consider only a subset of all possible plans at runtime. The plan eventually chosen for execution is intended to be the one that would be achieved by running the full System R optimizer at runtime. In a typical example of this work, Graefe and Cole describe *dynamic query plans* [GC94]. Given constraints on possible changes in the runtime environment, their optimizer would discard only those query plans that were suboptimal in all configurations satisfying these constraints. The result of their optimizer was a *set* of possible query plans, which was searched at runtime based on easily checkable parameters of the environment.

These schemes focus on the problem of postponing a minimal decision until runtime, effectively doing "late binding" of unknown variables for frequently re-executed queries. But they do not take any special advantage of iterative feedback, and offer the same frequency, effects and extent of adaptivity that one gets by running a System R optimizer whenever a query is to be executed.

2.3 Per-Query Adaptivity: Making System R More Adaptive

System R's statistics-gathering scheme was coarse grained, running only periodically, requiring administrative oversight, and consuming significant resources. Chen and Roussopoulos proposed an *Adaptive Selectivity Estimation* (ASE) scheme to enhance a System R-style optimizer by piggybacking statistics-gathering on query processing [CR94]. When a query was executed in their scheme, the sizes of sub-results would be tracked by the system, and used to refine statistical metadata for future optimization decisions. This provided an organic feedback mechanism, leveraging the natural behavior of query processing to learn more and perform better on subsequent queries. ASE operated on a moderately coarse *per-query* frequency – still inter-query like System R, but finer grained and more naturally adaptive. The effects of feedback in ASE were potentially as significant as those of System R, affecting access method selection, as well as join order and join method selection, with a long-term, inter-query extent. Note however that while ASE exploited information available from queries once they were issued, it did not gather information on tables that had not yet been referenced in queries.

The Mariposa distributed DBMS enabled per-query adaptivity with federated administration [SAP+96]. Mariposa used an economic model to let sites in a distributed environment autonomously model the changing costs of their query operations. During query optimization, Mariposa requested “bids” from each site for the cost of performing subplans. The bidding mechanism allowed sites to observe their environment from query to query, and autonomously restate their costs of operation for subsequent queries. In the initial experiments, Mariposa used this flexibility to allow the sites to incorporate load average and other transient parameters into their costs. More sophisticated “pricing agents” could be programmed into the system via a scripting language, and the commercial version of the system (Cohera [HSC99]) exposes an extensible API suitable for 3rd-party pricing agents (or “bots”). In terms of frequency, this is essentially like ASE: inter-query adaptivity at a per-query granularity. The novelty in Mariposa comes from the way that economics enables autonomous and extensible control of adaptive policies at sites in a federation. Mariposa used a “two-phase” optimization scheme, in which join orders and methods are chosen by a central optimizer, and bids only affect the choice of sites to do the work.

2.4 Competition and Sampling

One of the more unusual query optimizers to date is that of DEC (later Oracle) RDB [AZ96]. RDB was the first system to employ *competition* to help choose query plans. The RDB designers focused in particular on the challenge of choosing access methods for a given single-table predicate. They noted that the relative performance of the two access methods could be differentiated on-line by running both for only a short time. Based on this insight, they chose to simultaneously execute multiple access methods for a given table, and halt all but the most promising access method after a short time. RDB was the first system after Ingres to support adaptivity at an *intra-operator* frequency, though it only made one decision per table in a query, and only had effects on the choice of access method.

The RDB scheme is not unlike sampling-based schemes for selectivity estimation, which also perform partial query executions to learn more about the performance of a full run. A sequence of papers in the last decade has studied how to use sampling to estimate the selectivity of predicates (see Section 9 of [BDF$^+$97] for a survey). Sampling has typically been proposed for use during query optimization, and used only to direct the initial choice of a query plan (with the exception of online query processing, Section 2.7.2). Thus like ASE or Mariposa, this is *per-query* frequency – a finer grain than System R, but not the intra-query frequency of RDB. The effects are like those of System R: changing statistics can affect all aspects of query processing. However the extent is much shorter than System R, lasting only for the run of a single query.

2.5 Inter-Operator Reoptimization and Query Scrambling

As outlined in the introduction, the need for adaptivity grows with the uncertainty inherent in the query processing environment. Highly unpredictable environments, such as the Internet, require adaptivity even during the exeuction of a single query. A number of projects focusing on uncertain environments have employed intra-query adaptivity. Inter-operator adaptivity was a natural first step in this agenda. One approach used in distributed systems was to send subqueries to remote sites and then to use the arrival of the subquery results to drive the scheduling for the remaining parts of the query that used them [TTC$^+$90, ONK$^+$96]. Such approaches deal with uncertainties in the execution costs of the remote subqueries, and if subquery results are materialized, can also cope with unexpected delays to some extent.

Query Scrambling [AFTU96] was developed specificaly to cope with unexpected delays that arise when processing distributed queries in a wide-area network. With Query Scrambling, a query is initially executed according to a plan generated by a System R-style query optimizer. If, however, a significant performance problem is detected during the execution, the query plan is modified on the fly. Query Scrambling uses two basic techniques to cope with unexpected delays: 1) it changes the execution order of operations in order to avoid idling, and 2) it synthesizes new operations to execute in the absence of other work to perform. As described in [UFA98], the scrambling process can be driven by a lightwieght, response-time based query optimizer.

Kabra and DeWitt proposed a *reoptimization* scheme [KD98] to address uncertainties in the sizes of subquery results. As in Query Scrambling, an initial query plan is chosen by a traditional System R-style optimizer. After every blocking operator in that plan, the remainder of the plan is reoptimized with the knowledge of the size of the intermediate result generated thus far. The Tukwila system proposed a very similar technique, with a rule language to specify logic for deciding when to reoptimize, and the preservation of optimizer state to reduce the cost of reoptimization [IFF^{+}99].

In essence, inter-operator adaptivity is a long-postponed marriage of the Ingres and System R optimization schemes: like Ingres, it takes advantage of the cardinality information in materialized subresults; like System R it uses cost-based estimation of unknown work to be done. These schemes adapt at an inter-operator frequency, with arbitrary effects on the *remaining steps* after a block in a query plan; the extent does not go beyond the rest of the query.

2.6 Intra-Operator Adaptivity Revisited: WHIRL

WHIRL is a data integration logic designed at AT&T labs that focuses on textual similarity predicates, with an information-retrieval-style semantics of ranked results. The AT&T implementation of WHIRL attempts to return the top ranked answers quickly. The basic query processing operators in the WHIRL implementation are essentially scans ("exploding a literal") and ranked index lookups from inverted-file indexes ("constraining a literal"). At the end of each table-scan or index lookup, the implementation can choose among all subsequent possible table-scans or index lookups. The AT&T WHIRL implementation is remarkably similar to INGRES in its intra-operator adaptivity: it supports only nested loops joins, and adapts join order during a pipeline of these joins – potentially changing the order after each complete call to an access method.

2.7 Adaptive Query Operators

Up to this point, the effects of adaptivity that we have discussed have been at the level of query optimization: the choice of access methods, join methods, and join orders (and in the case of distributed queries, the choice of sites). Adaptivity can occur at the level of individual operators, which can adapt at an intra-operator frequency even within the context of a fixed query plan.

2.7.1 Memory-Adaptive Sorting and Hashing

Two of the basic operations in query processing are sorting and hashing. Both of these operations have costs that are a function of the amount of main memory available. Query optimizers estimate these costs under assumptions of memory availability. In some systems, adequate memory is reserved to guarantee these costs prior to execution; in others, memory allocation is allowed to proceed without regard to global consumption. The former technique is conservative, potentially resulting in under-utilization of resources and increases in query latency. The latter is aggressive, potentially resulting in paging and both decreased query throughput and increased latency.

A third approach is to modify sorting and hashing algorithms to adapt to changing amounts of available memory. Such schemes typically address both sudden losses and sudden gains in memory. Losses of memory are typically handled by spilling hash partitions or postponing the merger of sorted runs; gains in memory are exploited by reading in spilled hash partitions, or adding sorted runs to a merge phase. Exemplary work in this area was done by Pang, Carey and Livny, who studied both memory-adaptive variants of hash join [PCL93b] (based on earlier work including [NKT88, ZG90]), and memory-adaptive variants of out-of-core sorting [PCL93a] (followed by related work in [ZL97]).

2.7.2 Pipelining and Ripple Join Algorithms

The pipelining property of join algorithms has been exploited for parallelism; Wilschut and Apers proposed the symmetric hash join to maximize pipelined parallelism in their PRISMA/DB parallel, main-memory DBMS [WA91]. More recently, both the Tukwila and Query Scrambling groups extended the pipelining hash join in a natural fashion so that it would run out-of-core. The Query Scrambling group's *XJoin* algorithm [UF99] pushed the out-of-core idea further, to exploit delayed data feeds: when a particular tablescan is delayed, the XJoin expolits the idle time by pulling spilled partial partitions off of disk and joining them. Neither of these extensions is adaptive in the sense of our earlier definition, but both were used in the context of adaptive systems.

Recent work on *online query processing* in the Control project highlighted the ability for pipelining operators to provide continuous feedback and hence drive adaptivity [HAC+99]. In the initial work on Online Aggregation [HHW97], pipelining query processing algorithms were cited as a necessary condition for allowing *user* feedback during query processing. In subsequent work, system feedback was used internally by *ripple join* algorithms to automatically adapt the relative rates at which they fetch data from different tables, subject to a statistical performance goal [HH99]. The access methods in online query processing have typically been assumed to do progressive sampling without replacement, putting this work in the tradition of sampling for cost estimation.

Haas and Hellerstein [HH99] broadly define the ripple joins as a family of pipelining join algorithms that sweep out the cartesian plane in ever-larger rectangles, with the opportunity to adaptively control or tolerate changing "aspect ratios" of those rectangles based on observed statistical and performance behavior. They include in their definition the earlier pipelining hash join and index nested-loops joins, along with new iterative and "block" ripple joins that they introduce as modifications of the traditional nested loops joins[3]. Ripple joins as a class provide *intra-operator* frequency of adaptivity: behavior is visible at a *per-tuple* frequency, though decisions are made in [HH99] at a slightly coarser granularity, at the end of each "rectangle" swept out in processing. This adaptivity was exploited only for controlling the relative rates of I/O from different tables, during the run of a single query.

2.8 Rivers: Adaptive Partitioning

In a shared-nothing parallel DBMS, intra-operator parallelism is achieved by partitioning data to be processed among the nodes in the system. Traditionally this partitioning is done statically, via hashing or round-robin schemes. *River* [AAT+99] is a parallel dataflow infrastructure that was proposed for making this partitioning more adaptive. In developing the record-setting NOW-Sort implementation [AAC+97], the River designers noted that large-scale clusters exhibited unpredictable performance heterogeneity, even across physically identical compute nodes. As a result, they designed two basic mechanisms in River to provide data partitioning that adapted to the relative, changing rates of the various nodes. A *Distributed Queue* mechanism balanced work among multiple consumers running at different (and potentially changing) rates; a *Graduated Declustering* scheme dynamically adjusted the load generated by multiple redundant producers of data. Both these mechanisms adapted at a per-tuple frequency, affecting the assignment of work to nodes, with effects lasting for the duration of the operator.

2.9 Eddies: Continuous Adaptivity

In a query plan (or subplan) composed of pipelining operators like ripple joins, feedback is available on a tuple-by-tuple basis. As a result, it should be possible for a pipelined query (sub)plan to adapt at that frequency as well. Ripple joins, rivers, and other schemes leverage the feedback within an operator to change the behavior of the operator; *Eddies* are a mechanism to get inter-operator effects with intra-operator frequency of adaptivity [AH00].

[3]The aspect ratio of index nested-loops join is actually not subject to modification: the indexed relation is fully "swept" upon each index probe. However it was included in the family since it can present a performance enhancement over the iterative ripple joins when an index is available

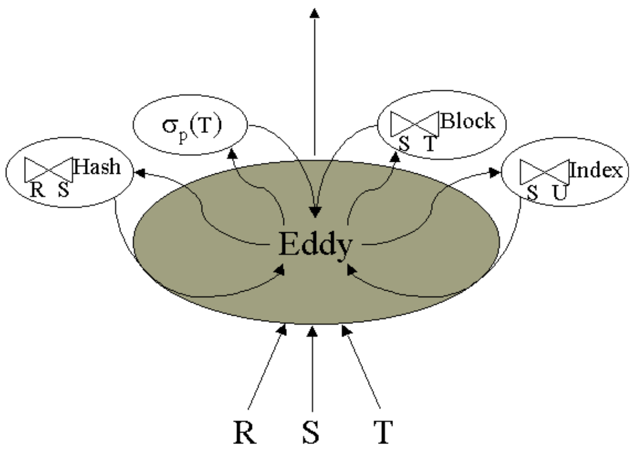


Figure 1: An eddy in a pipeline. Data flows into the eddy from input relations R, S and T. The eddy routes tuples to pipelining operators; the operators run as independent threads, returning tuples to the eddy. The eddy sends a tuple to the output only when it has been handled by all the operators. The eddy adaptively chooses an order to route each tuple through the operators.

An eddy is an encapsulated dataflow operator, akin to a join, sort, or access method, with an *iterator* interface. In the same way that the *exchange* operator of Volcano encapsulates parallelism, eddies are used to encapsulate adaptivity. An eddy is intended to be interposed between the tablescans in a plan (which produce input tuples to the eddy), and other "upstream" operators in the plan (which serve both to consume the eddy's output tuples and produce additional inputs).

As originally envisioned, the eddy – combined with pipelining operators like ripple joins – serves to adapt join order on a tuple-by-tuple basis. Every operator "upstream" of the eddy returns its results to the eddy, which passes them along to remaining operators for further processing. As a result, the eddy encapsulates the ordering of the operators by routing tuples through them dynamically (Figure 1). Because the eddy observes tuples entering and exiting the pipelined operators, it can adaptively change its routing to effect different operator orderings. It does so via a lottery scheduling scheme [WW94]. It can also control the rates of input from tablescans (if this is under the control of the system, as in the original online aggregation scenario.) In their original incarnation, the extent of adaptivity in eddies is restricted to a single query. In the next section we describe our agenda to extend the effects and extent of eddies.

3 Challenges in Adaptive Query Processing

While adaptivity has long been an issue in database research, we believe that the frequency of adaptivity in prior systems has been insufficient for many of the emerging large-scale environments. Work on continuous adaptivity has just begun, and we see a number of challenges and opportunities in this space.

In designing Telegraph, we are exploring new ideas in continuously adaptive query processing. The Telegraph query engine is a dataflow architecture based on rivers, eddies, and pipelining query processing operators like ripple joins and XJoins. As such, it is architected for fine-grained adaptivity and interactive user control. Rivers and eddies are quite general concepts, and we view them more broadly than the specific incarnations presented in the original papers. In particular, a major goal of Telegraph is to enhance both the extent and the effects of both schemes. Enhancing their extent is relatively simple: a metadata store can be used to save information at the end of each query, and this information consulted to set up initial conditions for subsequent queries (data partitioning, relative numbers of lottery tickets, etc.) Enhancing the set of effects from rivers and eddies is more complicated, and we are considering a number of challenges in this regard:

Spanning trees: As originally presented, eddies have no effect on the spanning tree for the query: i.e., the set of binary joins that connect the relations in the query. For cyclic join graphs, it is attractive to consider adapting the choice of spanning tree on the fly, by connecting the eddy to joins (or cross-products) between more pairs of tables than is strictly necessary. This is essentially a form of competition among spanning trees, and hence it can consume significant resources and produce duplicate results, two issues that we are currently investigating.

Data sources: River handles multiple potential data sources via graduated declustering, but requires a particular layout for those sources. We would like to extend this idea to a federated environment like the Internet, where data layout – and even data contents – are not under our control. This may produce duplicate or even inconsistent results, which need to be resolved.

Join and access methods: The choice of join and access methods can only be made on the fly by an eddy if they are run competitively, allowing the eddy to explore their behavior. In exploring this idea, we need to watch resource contention as well as the production of duplicates.

Moving state in rivers: The original River paper did not explicitly discuss how per-node state (e.g. hash partitions in a hash-join) could be migrated as the system adapts. We are investigating applying the ideas from memory-adaptive hash joins in this context.

In addition to extending rivers and eddies to have more general effects over a longer extent of time, we are considering a number of additional adaptivity questions:

Two-dimensional interactivity: Prior work on online query processing allowed "horizontal" user feedback on the *rows* to be favored for processing. We are also considering the "vertical" case in Telegraph: partially-joined tuples should be available at the output, and users should be able to express their relative desire for different *columns*, or even combinations of rows and columns. Partially-joined tuples may not necessarily have matches in all tables in the query, meaning that even though they are passed to the output, they may not be contained in the final answer. This has implications for user interfaces, client APIs, and of course the policies used in eddies.

Initial delays: Prior work on query scrambling addressed initial delays, but it is not clear how this work might dovetail with eddies and rivers. Challenges arise from the complete lack of feedback available in initial-delay scenarios.

Caching and prefetching: Caching is probably the most well-studied adaptive problem in computer systems, and is particularly relevant for a high-latency Internet environment. Prefetching is a similarly robust adaptive problem, dynamically predicting data fetches in the face of changing workloads. We are aggressively exploring both techniques in the context of Telegraph.

In addition to the constructive goals of Telegraph, many analytic questions remain as well. The first is to more carefully characterize the vagaries of all the application environments we are considering, including the wide area, clusters, and sensor networks. We have evidence that each of these environments merits a fine-grained frequency of adaptivity, and we believe that broad effects and extent of adaptivity will be appropriate as well. However the more we know about these environments, the better our adaptive systems should be able to perform. Second, we are actively engaged with our colleagues in theoretical computer science and machine learning in developing a formal framework for eddies, to try and characterize questions of convergence and stability, and to tune the policies used for adapting.

Acknowledgments

Thanks to Christos Papadimitriou, Stuart Russell and Alistair Sinclair for discussions on formal aspects of eddies and adaptivity. This work was supported by two grants from IBM Corporation, a grant from Siemens AG, a grant from ISX Corporation, NSF grant IIS-9802051, and DARPA contract N66001-99-2-8913. Hellerstein was supported by a Sloan Foundation Fellowship, Raman was supported by a Microsoft Graduate Fellowship.

Computing and network resources for this research were provided through NSF RI grant CDA-9401156.

References

[AAC+97] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-Performance Sorting on Networks of Workstations. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997.

[AAT+99] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, and Katherine Yelick. Cluster I/O with River: Making the Fast Case Common. In *Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, May 1999.

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of the ACM SIGMOD Int. Conf.*, Montreal, Canada, 1996.

[AFTU96] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling Query Plans to Cope With Unexpected Delays. In *4th International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, December 1996.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Dallas, 2000.

[Aok99] Paul M. Aoki. How to Avoid Building DataBlades(r) That Know the Value of Everything and the Cost of Nothing. In *11th International Conference on Scientific and Statistical Database Management*, Cleveland, July 1999.

[AZ96] Gennady Antoshenkov and Mohamed Ziauddin. Query Processing and Optimization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[BDF+97] Daniel Barbara, William DuMouchel, Christos Faloutsos, Peter J. Haas, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Theodore Johnson, Raymond T. Ng, Viswanath Poosala, Kenneth A. Ross, and Kenneth C. Sevcik. The New Jersey Data Reduction Report. *IEEE Data Engineering Bulletin*, 20(4), December 1997.

[BO99] J. Boulos and K. Ono. Cost Estimation of User-Defined Methods in Object-Relational Database Systems. *SIGMOD Record*, 28(3):22–28, September 1999.

[Coh98] William W. Cohen Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, 1998*, pages 201–212.

[CR94] Chung-Min Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 161–172. ACM Press, 1994.

[DH00] Amol Deshpande and Joseph M. Hellerstein. Decoupled query optimization in federated databases. Technical report, University of California, Berkeley, 2000.

[EGH99] Deborah Estrin, Ramesh Govindan, and John Heidemann. Scalable coordination in sensor networks. In *Proc. of ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, Seattle, August 1999.

[GC94] G. Graefe and R. Cole. Optimization of Dynamic Query Evaluation Plans. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, 1994.

[GW89] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 358–366. ACM Press, 1989.

[HAC+99] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, and Peter J. Haas Tali Roth. Interactive Data Analysis: The Control Project. *IEEE Computer*, 32(8):51–59, August 1999.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 287–298, Philadelphia, 1999.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, 1997.

[HP88] Waqar Hasan and Hamid Pirahesh. Query Rewrite Optimization in Starburst. Research Report RJ 6367 , IBM Almaden Research Center, August 1988.

[HSC99] Joseph M. Hellerstein, Michael Stonebraker, and Rick Caccia. Open, independent enterprise data integration. *IEEE Data Engineering Bulletin*, 22(1), March 1999.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc Fiedman, Alon Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Philadelphia, 1999.

[INSS92] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 103–114. Morgan Kaufmann, 1992.

[KD98] Navin Kabra and David J. DeWitt. Efficient Mid-Query Reoptimization of Sub-Optimal Query Execution Plans. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 106–117, Seattle, 1998.

[KKP99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile networking for smart dust. In *Proc. of ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, Seattle, August 1999.

[KRB99] Joanna Kulik, Wendi Rabiner, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. of ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, Seattle, August 1999.

[LP97] L. Liu and C. Pu. A dynamic query scheduling framework for distributed and evolving information systems. In *The IEEE Int. Conf. on Distributed Computing Systems (ICDCS-17)*, Baltimore, 1997.

[Met97] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the Usenix 1997 Technical Conference*, Anaheim, January 1997.

[Mit97] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[NKT88] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. Hash-partitioned join method using dynamic destaging strategy. In François Bancilhon and David J. DeWitt, editors, *Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings*, pages 468–478. Morgan Kaufmann, 1988.

[ONK+96] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac. Dynamic query optimization on a distributed object management platform. In *Conference on Information and Knowledge Management*, Baltimore, Maryland, November 1996.

[PCL93a] HweeHwa Pang, Michael J. Carey, and Miron Livny. Memory-adaptive external sorting. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 618–629. Morgan Kaufmann, 1993.

[PCL93b] HweeHwa Pang, Michael J. Carey, and Miron Livny. Partially preemptive hash joins. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 59–68. ACM Press, 1993.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[SAP+96] Michael Stonebraker, Paul M. Aoki, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, 5(1):48–63, January 1996.

[Son98] Eduardo D. Sontag. *Mathematical Control Theory: Deterministic Finite-Dimensional Systems, Second Edition*. Number 6 in Texts in Applied Mathematics. Springer-Verlag, New York, 1998.

[SWK76] M.R. Stonebraker, E. Wong, and P. Kreps. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[TTC+90] G. Thomas, G. Thompson, C. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman. Heterogeneous distributed database systems for product use. *ACM Computing Surveys*, 22(3), 1990.

[UF00] Tolga Urhan and Michael Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 2000. In this issue.

[UFA98] Tolga Urhan, Michael Franklin, and Laurent Amsaleg. Cost-Based Query Scrambling for Initial Delays. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Seattle, June 1998.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. First International Conference on Parallel and Distributed Info. Sys. (PDIS)*, pages 68–77, 1991.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 1–11, Monterey, CA, November 1994. USENIX Assoc.

[ZG90] Hansjörg Zeller and Jim Gray. An adaptive hash join algorithm for multiuser environments. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 186–197. Morgan Kaufmann, 1990.

[ZL97] Weiye Zhang and Per-Åke Larson. Dynamic memory adjustment for external mergesort. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 376–385. Morgan Kaufmann, 1997.

Adaptive Query Processing for Internet Applications

Zachary G. Ives
University of Washington
zives@cs.washington.edu

Alon Y. Levy
University of Washington
alon@cs.washington.edu

Daniel S. Weld
University of Washington
weld@cs.washington.edu

Daniela Florescu
INRIA Rocquencourt
Daniela.Florescu@inria.fr

Marc Friedman
Viathan Corp.
marc@viathan.com

Abstract

As the area of data management for the Internet has gained in popularity, recent work has focused on effectively dealing with unpredictable, dynamic data volumes and transfer rates using adaptive query processing techniques. Important requirements of the Internet domain include: (1) the ability to process XML data as it streams in from the network, in addition to working on locally stored data; (2) dynamic scheduling of operators to adjust to I/O delays and flow rates; (3) sharing and re-use of data across multiple queries, where possible; (4) the ability to output results and later update them. An equally important consideration is the high degree of variability in performance needs for different query processing domains: perhaps an ad-hoc query application should optimize for display of incomplete and partial incremental results, whereas a corporate data integration application may need the best time-to-completion and may have very strict data "freshness" guarantees. The goal of the Tukwila project at the University of Washington is to design a query processing system that supports a range of adaptive techniques that are configurable for different query processing contexts.

1 Introduction

Over the past few years, a new set of requirements for query processing has emerged, as Internet and web-based query systems have become more prevalent. In this emerging data management domain, queries are posed over multiple information sources distributed across a wide-area network; each source may be autonomous and may potentially have data of a different format. In some applications, the Internet query systems' results are fed into other data management tools; in other cases, the system interacts directly with the user in an ad-hoc environment. In certain contexts, the query processing system will handle a small number of concurrent queries; in others, there can be hundreds or even thousands of simultaneous requests. These different Internet query applications have many common requirements, but also require certain context-specific behaviors.

Modern query processors are very effective at producing well-optimized query plans for conventional databases, by leveraging I/O cost information as well as histograms and other statistics to determine the best executable

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

plans. However, data management systems for the Internet have demonstrated a pressing need for new techniques. Since data sources in this domain may be distributed, autonomous, and heterogeneous, the query optimizer will often not have histograms or any other quality statistics. Moreover, since the data is only accessible via a wide area network, the cost of I/O operations is high, unpredictable, and variable.

These factors can be mitigated through *adaptive query processing*, where the query processor adapts its execution in response to data sizes and transfer rates as the query is being executed. Moreover, the high I/O costs suggest that data should be processed as it is streaming across the network (as is done in relational databases with pipelining), scheduling of work should be dynamic to accommodate I/O latencies and data flow rates, and re-use and sharing of intermediate query results should be done wherever possible — both across concurrent queries, and between successive queries that execute within some short time delta of each other.

Adaptive query processing helps solve most of the problems common to Internet query systems. However, another important issue in using adaptive techniques, one that has seldom been considered, is the needs of the specific query context: the performance goals, as well as the applicable adaptive techniques, may vary widely depending on the application. For an ad-hoc, interactive query domain, the user may wish to see incomplete results quickly, but for a business-to-business environment, the emphasis may be on providing complete results as quickly as possible, with strict guarantees about data freshness.

The Tukwila project at the University of Washington is a data integration system, in which we attempt to answer queries posed across multiple, autonomous, heterogeneous sources. All of these data sources are mapped into a common *mediated schema*. The data integration system attempts to reformulate the query into a series of queries over the data sources, then combine the data into a common result. Tukwila's ancestors, the Information Manifold [LRO96] and Razor [FW97], focused on the problems of mapping, reformulation, and query planning; Tukwila attempts to address the challenges of generating and executing plans efficiently with little knowledge and variable network conditions.

The goal of Tukwila is to support efficient query processing of streaming XML data using adaptive query processing techniques, including display of incremental results and the sharing of sub-results across queries. In conjunction with this, we believe there is need for a method of expressing query processing *policies* — providing different query performance behaviors for different contexts.

In this paper we discuss a number of important areas that must be addressed using adaptive techniques for an effective wide-area XML data integration system. The paper is organized as follows: we begin in Section 2 with an overview of the different dimensions of adaptive query processing, which underly our research agenda for the Tukwila system. Section 3 describes adaptive techniques currently used within the Tukwila data integration system, and explains how they address some of the problems in this domain; in Section 4, we discuss the current focus areas of the Tukwila project. Finally, we conclude in Section 5.

2 Context-Specific Requirements in Adaptive Query Processing

Adaptive query processing encompasses a variety of techniques, some of which date back to the beginnings of relational database technology. These techniques can be classified by the granularities of adaptivity, as in the paper by Hellerstein et al. in this issue. Here we present an orthogonal classification of these techniques, based on the set of dimensions over which applications of adaptive query processing differ. We argue that adaptive query processors should be built in a flexible fashion, so they can be easily configured for any of these different application contexts. Below we begin by identifying the set of dimensions for Internet query processing, and in Sections 3 and 4, we discuss how current and future work on Tukwila addresses the requirements of these dimensions.

Data model To this point, most adaptive query processing techniques have focused on a relational (or object-relational) data model. While there are clearly important research areas within this domain, other data models may require extensions to these techniques. In particular, XML, as a universal format for describing data, allows for hierarchical and graph-structured data. We believe an execution model similar to pipelining is important for the XML realm, as processing of streaming data is of growing impact.

Remote vs. local data Traditional database systems have focused on local data. Recent work has focused on techniques for increasing the performance of network-bound query applications, including [UFA98, UF99, IFF+99, AH00, HH99]. (See the Hellerstein et al paper in this volume for greater detail.)

First vs. last tuple For domains in which the data is used by an application, the query processor should optimize for overall query running time — the traditional focus of database query optimizers. Most of today's database systems do all of their optimization in a single step; the expectation (which does not generally hold for large query plans [AZ96] or for queries over data where few statistics are available) is that the optimizer has sufficient knowledge to build an efficient query plan. The INGRES optimizer [SWKH76] and techniques for mid-query re-optimization [KD98] often yield better running times, because they re-optimize later portions of the query plan as more knowledge is gained from executing the earlier stages. Similar re-optimization techniques can also be applied to interactive domains, as discussed in [IFF+99, AH00, UFA98], because they can often produce output faster by using a superior query plan.

Approximate results In interactive domains, we may wish to see incremental display of the query results, with incomplete or approximate answers that evolve towards their final values. Operators supporting output of partial results have been a focus of recent work in [HHW97], which provided incremental display of approximate results for root-level aggregation, and [STD+00], which proposed a more general approach for providing partial results on demand. However, another important aspect of this area is a method of specifying *when* to provide partial answers, as the user may only want to see tentative results for certain data items. Moreover, a more formal definition is needed for the semantics of when a partial or approximate result is meaningful.

Incremental updates In certain applications where data constantly changes, it is important to be able to execute the query over an initial data set, and thereafter to process "deltas" describing updates to the original data values. Early work in this area includes the partial-results feature of the Niagara system [STD+00].

Number of queries If the domain includes large numbers of similar queries being posed frequently, the query processor should generate query plans with a focus on materialization of partial results for future reuse, and it should make use of common subexpressions. Work in this area includes the NiagraCQ [CDTW00] system at Wisconsin and the OpenCQ and WebCQ projects at Georgia Tech. This problem is similar to that of multi-query optimization [Sel88, RSSB00] but with a more "online" character — as optimization is done for potential *future* reuse of subquery results — and generally larger numbers of queries.

Freshness Data may often be prefetched and cached by the query processor, but the system may also have to provide data freshness guarantees. Caching and prefetching are well-studied areas in the database community. Likewise, the work on rewriting queries over views [Lev00] can be used to express new queries over cached data, rather than going to the original data sources.

3 The Tukwila Data Integration System

In a domain where costs are unpredictable and dynamic, such as data integration for the wide area, a query processing system must react to changing conditions and acquired knowledge. This is the basic philosophy behind the Tukwila project, which focuses on providing a configurable platform for adaptive query processing of streaming data.

In this section, we present an overview of the basic techniques implemented in Tukwila. There are three primary aspects to the Tukwila adaptive framework: an event-condition-action-based rule system, a set of adaptive operators, and the ability to incrementally re-optimize a query plan as greater knowledge about the data is gained. Here we provide a brief overview of these capabilities; for more information, see [IFF^{+}99].

3.1 Controlling Adaptive Behavior

An important need in dealing with network-based query sources is the ability to respond to unexpected conditions: slow data sources, failed sources, amounts of data that are much larger than expected, etc. In order to handle conditions such as these, Tukwila incorporates event-condition-action rules that can respond to execution events such as operator start, timeout, insufficient memory, end of pipeline, and so forth. In response to these events, Tukwila can return to the query optimizer to re-optimize the remainder of a query plan; it can modify memory allocations to operators; it can switch to an alternate set of data sources. Note that these rules are at a lower granularity than triggers or active rules: they respond to events at the sub-operation level, and can also modify the behavior of query plan operators.

3.2 Intra-Operator Adaptivity

The Tukwila system provides two operators that can respond to varying network conditions and produce optimal behavior. The first is an implementation of the pipelined hash join [WA91] with extensions to support overflow of large hash tables to disk; in many ways it resembles the hash ripple join [HH99] and the XJoin [UF99].

A pipelined hash join operates with two hash tables, rather than the single hash table of a typical hybrid hash join. A tuple read from one of the operator's inputs is stored in that input's hash table and probed against the opposite table. Each input executes in a separate thread, and this provides two highly desirable characteristics: it allows overlap of I/O and computation, which is important in an I/O-bound environment, and it produces output tuples as early as possible. The pipelined hash join also adjusts its behavior to the data transfer rates of the sources. The trade-off is that it uses more memory than a standard hybrid hash join; however, this problem can be mitigated with the overflow strategies implemented by Tukwila or the XJoin operator.

In many web applications, there may be multiple sites from which the same input data can be obtained; some of these data sources may be preferable to others, perhaps because of connection speed or cost. Tukwila's *collector* operator provides a robust method for reading data from sources with identical schemas: according to a *policy* specified in Tukwila's rule language, the collector attempts to read from a subset of its sources; if a given source is slow or unavailable, the collector can switch to one or more alternate data sources. This operator allows the query engine to choose data sources based on criteria such as availability or speed.

3.3 Incremental Re-Optimization

Adaptive behavior during query execution is key in situations where I/O costs are variable and unpredictable. When data sizes are also unpredictable, it is unlikely that the query optimizer will produce a good query plan, so it is important to be able to modify the query plan being executed. As a result, Tukwila supports incremental re-optimization of queries during particular plan execution points.

The Tukwila re-optimization model is based on *fragments*, or pipelined units of execution. Fragment boundaries, at which a pipeline is broken and the results are materialized, are chosen by the optimizer according to their cost and potential benefits. In general, a large query plan must already be broken into smaller pipelines so operators will fit into memory; this is particularly true if memory intensive operators such as the pipelined hash join are used. At each materialization point, Tukwila's execution system can check whether the result cardinality was close to that expected by the optimizer; if the cardinality is sufficiently divergent, Tukwila will keep the current query subresults and re-optimize the remainder of the query plan, using the subresults as inputs for a new and better plan.

The Tukwila model for re-optimization is similar to that proposed in [KD98], but it allows the optimizer to choose fragmentation points in an integrated, cost-based approach, rather than adding the capabilities in a separate postprocessing step.

4 Current Areas of Focus in Tukwila

The Tukwila system already supports a number of adaptive techniques, but the system is being extended in a number of ways. Our current work focuses on many of the areas discussed in Section 2.

4.1 XML: a Foundation for Data Integration

Tukwila was initially developed for a relational data model, and required *wrappers* for every data source, to translate data from source formats into the standard Tukwila data format. XML reduces the difficulty of building wrappers, as most data sources have begun to include XML export capabilities, and as various HTML-to-XML wrapper toolkits (e.g. [SA99, LPH00]) have emerged[1]. However, the use of XML, which supports flat, hierarchical, and graph-structured data, has led to a natural extension of the Tukwila data model to fully support semistructured data.

Our data model is based on an ordered, directed-graph approach like that of XML-QL [DFF$^+$99]. This model is powerful enough to support any of the proposed XML query languages, including XML-QL, XQL [RLS98], XPath [CD99], and Quilt [CRF00]. As discussed below, one of our goals in Tukwila is to support the processing of streams of changing data. To this end, we are developing a language for specifying updates to XML documents.

4.2 Processing Streaming XML Data

To this point, XML query processors have worked by mapping XML data into an underlying local store — relational, object-oriented, or semistructured — and have done their processing within this store. For a network-bound domain where data may need to be re-read frequently from autonomous sources to guarantee "freshness," this approach does not produce good performance. Thus it is imperative that an XML data integration system support direct querying of data as it streams in, much as a relational engine can pipeline tuples as they stream in.

In order to provide pipeline-like processing of network data, we must be able to support efficient evaluation of *regular path expressions* over the incoming data, and incremental output of the values they select. Regular path expressions are a mechanism for describing traversals of the data graph using edge labels and optional regular-expression symbols such as the Kleene-star (for repetition) and the choice operator (for alternate subpaths). Regular path expressions bear many similarities to conventional object-oriented path expressions, and can be computed similarly; however, the regular expression operators may require expensive operations such as joins with transitive closure.

We have developed the *x-scan* operator, which evaluates regular path expressions across incoming XML data, and which binds query variables to nodes and subgraphs within the XML document. X-scan is discussed in greater detail in [ILW00], and includes support for both tree- and graph-structured documents, while preserving document order.

Building upon X-scan, we are developing a complete query processor over streaming XML data. The key additional operators we are considering are a *nest* operator, which nests subelements under parents, in a join-like

[1] Legacy applications will still need wrappers to convert from their formats to XML, but those wrappers should be generic enough to be usable by all XML data consumers, unlike the previous situation where a separate wrapper needs to be created for each data provider and consumer.

fashion; and a *fuse* operator, which can be used to support graph-model features in an XML output document by consolidating multiple output nodes.

4.3 Specifying Adaptive Behavior

In Section 2, we discussed a number of dimensions of adaptive query processing. Different data management applications have very different needs within these dimensions. Factors may include whether to optimize for first or last tuple, how fresh the data from each query must be (and thus how long data can be cached), and whether (and when) the user should see approximate or incomplete data.

Additionally, the query optimizer should behave quite differently if the domain is one in which many similar queries are being posed, rather than one in which a few simple queries are given. For the multi-query case, the query processor should evaluate the potential benefits of materializing subresults for reuse in future queries.

Although optimizing for each of these various needs has been fairly well-studied, much less work has been done on actually expressing the query processing requirements, and on being able to support all of these cases within a single unified framework. This is an area we plan to address within Tukwila: developing a system to support a wide range of applications, and, equally important, providing a configuration language for specifying the requirements of a given domain.

4.4 Increasing Pipelined Behavior

In terms of query execution, an important need for interactive applications is to facilitate output of first tuples — the user should receive results as soon as possible. Tukwila includes adaptive operators whose intent is to address this requirement.

However, these needs must be balanced by the fact that the query optimizer, which does not have good knowledge of the data sources, may have produced a suboptimal plan. The optimizer initially divides the plan into fragments (pipelines with materialization points) based on expected memory usage and other factors such as confidence in its statistics. Each of these materialization points breaks the pipeline, generally slowing time to first tuple — however, if the plan gets re-optimized into a more efficient form, the net result should be a faster time to completion, and potentially even a better time to first tuples.

4.4.1 Dynamically Choosing Materialization Points

Clearly, there is a trade-off between the number of materialization points and the query processing time. Unfortunately, with few statistics available, the optimizer is unlikely to be able to choose good materialization points; it is likely to have too many, too few, or poorly placed breaks in the pipeline. (This problem is also present in traditional systems with quality statistics, appearing for complex queries with many join operations.) We are investigatating the performance implications of choosing the materialization points adaptively, during plan execution. In our approach, the query optimizer creates long pipelines; when these pipelines run out of memory (e.g. a join algorithm needs to overflow to disk), the execution engine will, using "hints" provided by the optimizer, insert a new materialization point into the middle of the pipeline. All operators "upstream" of this new materialization point will flush their results to disk; execution of the operators below the materialization point will continue. Once they complete, the upstream operators will reload their intermediate results, begin reading from the materialized file, and resume normal operation.

We expect that there will be several benefits to this approach. First, early results will likely be able to percolate through the entire long pipeline before the system runs out of memory — this speeds time to initial tuples. Second, the system will only insert materialization points where necessary, something that is extremely difficult to do statically. Third, the cost of breaking a pipeline should generally be less than that of having multiple join algorithms simultaneously overflowing, as it allows the query processor to "stage" portions of the data that exceeds memory.

4.4.2 Returning Incremental Results

Another important feature that is important for interactive applications is the ability to display approximate results incrementally. Initial work in this area has been done in the context of the Niagara system [STD$^+$00], which allows the user to request partial results at any time, and also within the CONTROL project [HHW97] for top-level aggregate operators.

Our focus in this area is on two important problems. First, an interactive query system would ideally provide incremental results for all query types in an interactive, browsable window, where the query processor "focuses" on finalizing the results currently in the user's view. Second, it will often be the case that for a given domain or query, approximate or partial results are only useful for certain items within the data set. A system that supports partial results should also support a mechanism for expressing which data items should be approximated. We believe this should be one aspect of the configuration language discussed in Section 4.3.

5 Conclusions

Adaptive query processing is a rapidly growing field, as evidenced by this special issue. Certain aspects of this work go back to the early days of relational databases, but the evolution of data integration and data management systems for the Internet has led to a number of recent developments.

We believe that one of the most important areas of future exploration should be in developing a system flexible enough to meet the wide range of domain-specific needs, and providing a means of specifying the relevant parameters to the system. The Tukwila project is attempting to address aspects of both of these problems, using XML as the standard data format and data model. We believe that the current system has taken a number of steps in this direction, and that our current and future work will take us much closer to a comprehensive data management solution for Internet-based data.

References

[AH00] Ron Avnur and Joseph M. Hellerstein. Continuous query optimization. In *SIGMOD 2000, Proceedings ACM SIGMOD International Conference on Management of Data, May 14-19, 2000, Dallas, Texas, USA*. ACM Press, 2000.

[AZ96] Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optmization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[CD99] James Clark and Steve DeRose. XML path language (XPath) recommendation. http://www.w3.org/TR/1999/REC-xpath-19991116, November 1999.

[CDTW00] Jianjun Chen, David DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD 2000, Proceedings ACM SIGMOD International Conference on Management of Data, May 15-18, 1999, Dallas, TX, USA*, 2000.

[CRF00] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings) 2000*, pages 53–62, 2000.

[DFF$^+$99] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *Proceedings of the International Word Wide Web Conference, Toronto, CA*, 1999.

[FW97] Marc Friedman and Daniel S. Weld. Efficiently executiong information-gathering plans. In *Proceedings of the Fifteenth International Join Conference on Artificial Intelligence*, pages 785–791, 1997.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 287–298. ACM Press, 1999.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen Wang. Online aggregation. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 171–182. ACM Press, 1997.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc T. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 299–310, 1999.

[ILW00] Zachary G. Ives, Alon Y. Levy, and Daniel S. Weld. Efficient evaluation of regular path expressions over streaming XML data. Technical Report UW-CSE-2000-05-02, University of Washington, May 2000.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 106–117. ACM Press, 1998.

[Lev00] Alon Y. Levy. Answering queries using views: A survey, 2000. Manuscript available from `www.cs.washington.edu/homes/alon/views.ps`.

[LPH00] Ling Liu, Calton Pu, and Wei Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, CA USA*, pages 611–621, 2000.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufmann, 1996.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, September 1998.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD 2000, Proceedings ACM SIGMOD International Conference on Management of Data, May 14-19, 2000, Dallas, Texas, USA*. ACM Press, 2000.

[SA99] Arnaud Sahuguet and Fabien Azavant. Building light-weight wrappers for legacy web data-sources using W4F. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 738–741. Morgan Kaufmann, 1999.

[Sel88] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[STD+00] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey F. Naughton, and David Maier. Architecting a network query engine for producing partial results. In *WebDB (Informal Proceedings) 2000*, pages 17–22, 2000.

[SWKH76] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The design and implementation of INGRES. *TODS*, 1(3):189–222, 1976.

[UF99] Tolga Urhan and Michael J. Franklin. XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, University of Maryland, College Park, February 1999.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 130–141, 1998.

[WA91] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. First International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 68–77, December 1991.

XJoin: A Reactively-Scheduled Pipelined Join Operator†

Tolga Urhan
University of Maryland, College Park
urhan@cs.umd.edu

Michael J. Franklin
University of California, Berkeley
franklin@cs.berkeley.edu

Abstract

Wide-area distribution raises significant performance problems for traditional query processing techniques as data access becomes less predictable due to link congestion, load imbalances, and temporary outages. Pipelined query execution is a promising approach to coping with unpredictability in such environments as it allows scheduling to adjust to the arrival properties of the data. We have developed a non-blocking join operator, called XJoin, which has a small memory footprint, allowing many such operators to be active in parallel. XJoin is optimized to produce initial results quickly and can hide intermittent delays in data arrival by reactively scheduling background processing. We show that XJoin is an effective solution for providing fast query responses to users even in the presence of slow and bursty remote sources.

1 Wide-Area Query Processing

The explosive growth of the Internet and the World Wide Web has made tremendous amounts of data available on-line. Emerging standards such as XML, combined with wrapper technologies address semantic challenges by providing relational-style interfaces to remote data. Beyond the issues of structure and semantics, however, there remain significant technical obstacles to building responsive, usable query processing systems for wide-area environments. A key performance issue that arises in such environments is *response-time unpredictability*. Data access over wide-area networks involves a large number of remote data sources, intermediate sites, and communications links, all of which are vulnerable to overloading, congestion, and failures. Such problems can cause significant and unpredictable *delays* in the access of information from remote sources. These delays, in turn, cause traditional distributed query processing strategies to break down, resulting in unresponsive and hence, unusable systems.

In previous work [AFTU96] we identified three classes of delays that can affect the responsiveness of query processing: 1) *initial delay*, in which there is a longer than expected wait until the first tuple arrives from a remote source; 2) *slow delivery*, in which data arrive at a fairly constant but slower than expected rate; and 3) *bursty arrival*, in which data arrive in a fluctuating manner. With traditional query processing techniques, query execution can become blocked even if only one of the accessed data sources experiences such delays.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

†This work was partially supported by the NSF under grant IRI-94-09575, by the Office of Naval Research under contract number N66001-97-C8539 (DARPA order number F475), by a Siemens Faculty Development Award, and by an IBM Partnership Award.

We developed Query Scrambling to address this problem and showed how it can be used to hide initial delays [UFA98] and bursty arrivals [AFT98]. Query Scrambling is a *reactive* approach to query execution; it reacts to data delivery problems by on-the-fly rescheduling of query operators and restructuring of the query execution plan. Query Scrambling is aimed at improving the response time for the *entire* query, and may actually slow down the return of some initial results in order to minimize the time required to produce the remaining portion of a query answer once all necessary data has been obtained from all of the remote sources.

In this paper we explore a complementary approach using a non-blocking join operator we call XJoin. XJoin is based on two fundamental principles:

1. *It is optimized for producing results incrementally as they become available.* When used in a fully pipelined query plan, answer tuples can be returned to the user as soon as they are produced. The early delivery of initial answers can provide tremendous improvements in the responsiveness observed by the users.
2. *It allows progress to be made even when one or more sources experience delays.* There are two reasons for this. First, XJoin requires less memory, which allows for bushier plans. Thus, some parts of a query plan can continue while others are stalled waiting for input. Second, by employing background processing on *previously received* tuples from both of its inputs, an XJoin operator can produce results even when both inputs are stalled simultaneously.

XJoin is based on the Symmetric Hash Join (SHJ) [WA91, HS93] which was originally designed to allow a high degree of pipelining in parallel database systems. As originally proposed, however, SHJ requires that hash tables for both of its inputs be kept in main memory during most of the query execution. As a result, SHJ cannot be used for joins with large inputs, and the ability to run multiple joins (e.g., in a bushy query plan) is severely limited. XJoin extends the symmetric hash join to use less memory by allowing parts of the hash tables to be moved to secondary storage. It does this by partitioning its inputs, similar in spirit to the way that hybrid hash join solves the memory problems of classic hash join.

Simply extending SHJ to use secondary storage, however, is insufficient for tolerating significant delays in receiving data from remote sources. For this reason, a key component of XJoin is a *reactively scheduled* background process, which opportunistically utilizes delays to produce more tuples earlier. We show that by using XJoin it is possible to produce query execution plans that can better cope with data delivery problems and that can deliver initial results orders of magnitude faster than traditional techniques, with in many cases, little or no degradation in the time required to deliver the entire result.

The main challenges in developing XJoin include the following:

- Managing the flow of tuples between memory and secondary storage.
- Controlling the background processing that is initiated when inputs are delayed.
- Ensuring that the full answer is ultimately produced (i.e., no answers should be lost).
- Ensuring that no duplicate tuples are inadvertently produced.

The work described in this paper is related to other recent projects on improving the responsiveness of query processing, including techniques for returning initial answers more quickly [BM96, CK97] and those for returning continually improving answers to long running queries [VL93, HHW97]. Our work differs from this other research due to (among other reasons) the focus on coping with unpredictable delays arising from wide-area remote data access. The Tukwila system [IFFL$^+$99] incorporates an extension of SHJ called Double Pipelined Hash Join (DPHJ) that can work with limited memory. DPHJ differs from XJoin in several details such as the way in which tuples are flushed to secondary storage. More importantly, as originally specified, DPHJ does not include reactively-scheduled background processing for coping with delayed sources. Both DPHJ and XJoin can be thought of as types of Ripple Joins [HH99] which are a class of pipelined join operators that allow the order of data delivery to be adjusted dynamically.

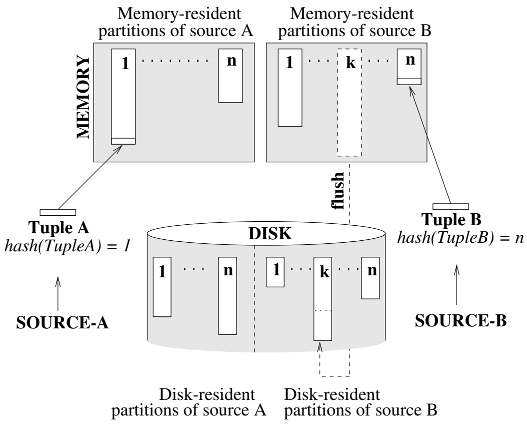


Figure 1: Handling the partitions.

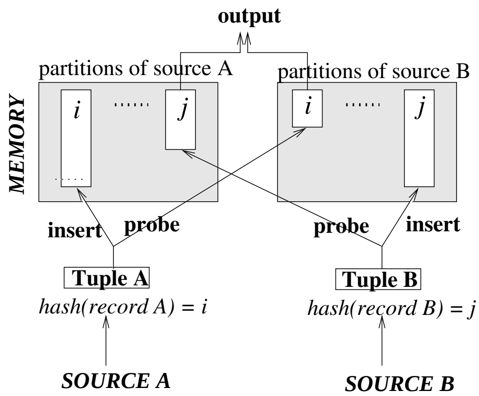


Figure 2: Stage 1 - Memory-to-Memory joins

2 The Design of XJoin

In this section we only give a brief overview of the mechanisms used by XJoin. A more detailed description of XJoin is given in [UF99].

2.1 The Three Stages of XJoin

XJoin proceeds in three stages, each of which is performed by a separate thread. The first stage joins memory-resident tuples, acting similarly to the standard symmetric hash join. The second stage joins tuples that have been flushed to disk due to memory constraints. The third stage is a clean-up stage, which performs any necessary matching to produce results missed by the first two stages. The first and second stages run in an interleaved fashion — the second stage takes over when the first becomes blocked due to a lack of input. These stages are terminated after all input has been received, at which point the third stage is initiated.

First Stage

The first stage works similarly to the original symmetric hash join. The main difference is that in XJoin, the tuples are organized in partitions (Figure 1). In general each partition can consist of two portions: a memory-resident portion, which stores the most recently arrived tuples for that partition, and a disk-resident portion, which contains tuples of the partition that have been flushed to disk due to memory constraints. When an input tuple arrives from a source, if there is memory available for the tuple then it is simply placed in its partition and used to probe the memory-resident portion of the corresponding partition for the other source (Figure 2). If, however, memory is full, then one of the partitions is chosen as a victim and its memory-resident tuples flushed to disk (i.e., appended to its disk-resident portion). Join processing then continues as usual. The first stage runs as long as at least one of its inputs is producing tuples. If the first stage ever times out on both of its inputs (e.g., due to some unexpected delays), it blocks and the second stage is allowed to run. The first stage terminates when it has received all of the tuples from both of its inputs.

Second Stage

The second stage is activated whenever the first stage blocks. It first chooses a partition from one source using optimizer-generated estimates of the output cardinality and the cost of performing the stage using the partition. [1] It then uses the tuples from the disk-resident portion of that partition to probe the memory-resident portion of the corresponding partition of the other source. Any matches found are output (subject to duplicate detection as described in Section 2.2) as result tuples. After a disk-resident portion has been completely processed, the operator checks to see if either of the join inputs have resumed producing tuples. If so, then the second stage halts and the first stage is resumed, otherwise a different disk-resident portion is chosen and the second stage is

[1] Note that the same partition can be used multiple times, as the partition grows over the course of the join execution.

continued. As an additional optimization, tuples brought into memory during one iteration of the Second Stage can be probed with disk-resident tuples from the corresponding partition of the other source in the subsequent iteration.

It is important to note that XJoin follows the Query Scrambling philosophy of hiding delays by performing other work. In particular, the second stage incurs processing and I/O overhead in the hope of generating result tuples. *This work is essentially free as long as the inputs of the XJoin are delayed*, as no progress could be made in that situation anyway. This is where the benefit of the second stage comes in. The risk is that when one or both of the inputs become unblocked it is not noticed until after the current disk-resident partition has been fully processed. In this case, the overhead of the second stage is no longer completely hidden.

Third Stage

The third stage executes after all tuples have been received from both inputs. It is a clean-up stage that makes sure that all the tuples that should be in the result set are ultimately produced. This step is necessary because the first and second stages may only partially compute the result.

2.2 Handling Duplicates in XJoin

The multiple stages of XJoin may produce spurious duplicate tuples because they can perform overlapping work. Duplicates can be created in both the second and third stages. To address this problem XJoin uses a duplicate prevention mechanism based on timestamps.

XJoin augments the structure of each tuple with two persistent timestamps: an Arrival TimeStamp (ATS), which is assigned when the tuple is first received from its source and a Departure TimeStamp (DTS), which is assigned when the tuple is flushed from memory. The ATS and DTS together describe the time interval during which a tuple was in the memory-resident portion of its partition.

These timestamps are used to check whether two tuples have previously been matched by the first stage or second stage. If so these tuples are not matched again. Checking for the matches from first stage is easy. For a pair of tuples to have been matched by the first stage they both must have been in memory at the same time, thus they must have overlapping ATS and DTS ranges. Any such pair of tuples are not considered for joining by the second or third stages.

The ATS and DTS are not enough to detect tuples matched in the second stage. In order to solve this problem XJoin maintains a linked list for each partition processed by the second stage. The entries in the list are of the form $\{DTS_{last}, ProbeTS\}$ where DTS_{last} is the DTS value of the last tuple of the disk-resident portion that was used to probe the memory-resident tuples, and $ProbeTS$ is the timestamp value at the time that the second stage was executed. [2] These entries can be used to infer that all tuples of disk-resident portion having DTS values up to (and including) DTS_{last} were used by the second stage at time $ProbeTS$.

When two tuples, T_A and T_B, are later matched we first check when T_A was used to probe memory-resident tuples using the linked list maintained for the partition it belongs. If T_B was memory-resident during this time we do not join these two tuples again. The same check is performed for the symmetrical case to determine if T_A was memory resident when T_B was used to probe memory-resident tuples.

2.3 Controlling the second stage

Recall that the overhead incurred by the second stage is hidden only when both inputs to the XJoin experience delays. As a result, there is a tradeoff between the aggressiveness with which the second stage is run, and the benefits to be obtained by using it. To address this tradeoff, our implementation includes a mechanism that can be used to restrict the second stage to processing only those partitions that are likely to yield a significant number of result tuples. This *activation threshold* is specified as a percentage of the total number of result tuples expected

[2]Note that the timestamp value remains unchanged during an execution of the second stage since no tuples can be added to or evicted from memory while it is executing.

to be produced from a partition during the course of the entire join. For example, if the join of a partition is expected to contribute 1000 tuples to the result, a threshold value of 0.01 will allow the second stage to process the partition as long as it is expected to produce 10 or more tuples. Thus, a lower activation threshold results in a more aggressive use of the second stage.

In our implementation of XJoin we dynamically change the value of *activation threshold*, starting with an aggressive value (0.01) and gradually make it more conservative (up to 0.20) as more output tuples are produced. This has the effect of emphasizing the interactive performance at the beginning of the execution and overall performance (i.e., a more traditional criterion) towards the end of the execution.

3 Experimental Results

We have implemented XJoin in an extended version of PREDATOR [SP97], an Object-Relational DBMS, and performed detailed experiments to investigate performance issues associated with various aspects of XJoin. Due to space limitations, however, we only present a portion of the results here. Detailed results can be found in [UF99].

3.1 Experimental Environment

In the experiments we modeled the behavior of the network using trace data that was obtained by fetching large files from 15 randomly chosen sites. From these arrival patterns, we chose two as representatives of the behavior of a bursty and a fast source (figures 3, and 4). The arrival patterns in these figures show the quantity of data received at the query site. We refer to the bursty pattern also as "slow" arrival pattern due to its low transfer rate.

The database used in the experiments contained up to six 100,000 tuple Wisconsin benchmark relations [BDT83]. Each input tuple is 86 bytes after projections have been applied. Join attributes used are one-to-one, producing 100,000 result tuples. We ran the experiments on a Sun Ultra 5 Workstation with 128 MBytes of memory. In the experiments the XJoin operator is given 3 MBytes of memory.

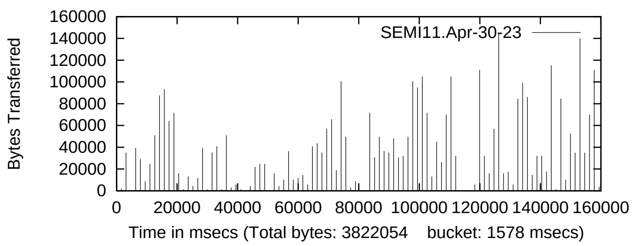


Figure 3: Bursty arrival. Avg. Rate 23.5 KBytes/sec.

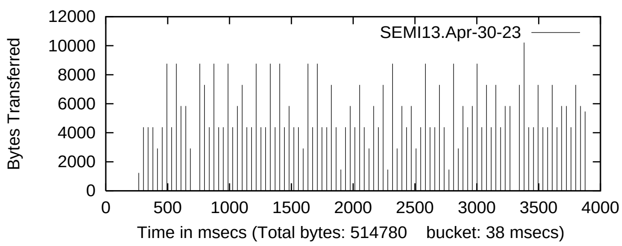


Figure 4: Fast arrival. Avg. Rate 129.6 KBytes/sec.

3.2 Results

We compared the performance of XJoin to that of Hybrid Hash Join (HHJ). In order to separate out the contributions of the major components of the algorithm we also examined two other XJoin variants. The first variant, labeled *XJoin-No2nd*, does not use the second stage at all. The second variant, labeled *XJoin-Aggr* is an aggressive version of XJoin which uses an aggressively set *activation threshold* (i.e., 0.01) . We also tried to improve the responsiveness of HHJ by allowing base tuples to be fetched in parallel in the background. This parallelism allows HHJ to overlap delays from one input with the processing of the other.

Figures 5 and 6 show the cumulative response times for the four algorithms for the bursty and fast arrival cases respectively. The x-axis shows a count of the result tuples produced and the y-axis shows the time at which that result tuple was produced. In both cases XJoin and its variants produce the first answers several orders of

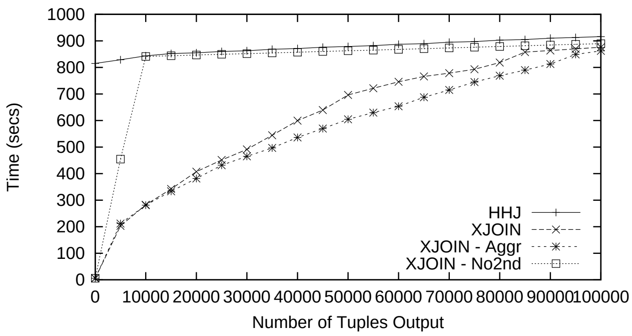


Figure 5: Slow arrival

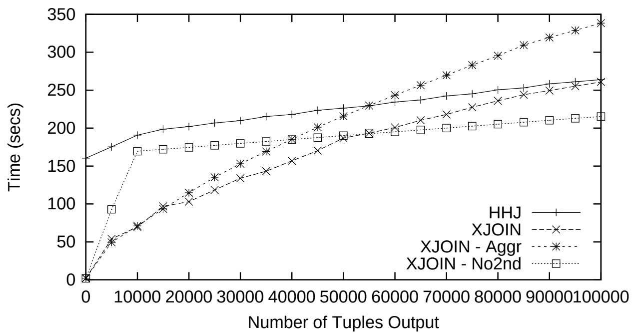


Figure 6: Fast arrival

magnitude faster than HHJ, thereby providing far superior interactive performance. XJoin also outperforms HHJ in terms of the time to return the entire result.

A comparison between the XJoin and XJoin-No2nd highlights the importance of the second stage in improving the responsiveness of the system. XJoin-No2nd, although performing competitively for the very initial results, fails to maintain this performance for majority of the results. A comparison between XJoin and XJoin-Aggr is perhaps more interesting as it demonstrates the tradeoff of executing the second stage. In the slow network case XJoin-Aggr performs slightly better than XJoin in the middle range. This is because there is enough delay to hide the extra work introduced by XJoin-Aggr. However this improvement is at the expense of poor response in the fast network case (Figure 6). In the absence of enough delay to overlap the overhead of second stage XJoin-Aggr falls behind XJoin.

Other results, not included in this paper, have also showed the superiority of XJoin in delivering the initial portion of the result under variety of conditions. Experiments measuring the effect of memory size have shown that XJoin has robust performance even with very limited memory. Further experiments stress tested XJoin by running queries involving up to 5 join operators (up to 6 inputs). In all the cases XJoin was able to outperform HHJ in delivering the initial portion of the result with only minor degradation in delivering the last tuple.

4 Conclusion and Future Work

In this paper, we described the design of XJoin, a reactively scheduled pipelined join operator capable of providing responsive query processing when accessing data from widely-distributed sources. XJoin incorporates the Query Scrambling philosophy of hiding unexpected problems in data arrival by performing other (non-scheduled) useful work. The smaller footprint is obtained through the use of partitioning. The delay-hiding feature is implemented through the use of a reactively-scheduled "second stage", which aims to produce result tuples during periods of delayed or slow input by joining tuples of one input that have been spooled to secondary storage with the memory-resident tuples of the other input.

In terms of future work, we plan to investigate the scheduling issues in complex query plans with multiple XJoin operators. Currently XJoin operators are scheduled in a round-robin fashion. Rate at which initial portion of the result delivered can be improved by scheduling more productive operators (i.e., low cost operators that contribute more to the result) frequently. We also plan to work on delivering more "interesting" portions of a result (such as some subset of columns) faster in wide-area environments. Such query behavior is desirable when the semantics of the application are such that some identifiable portions of the data are substantially more important than others.

In the larger context, XJoin represents one piece of technology that can help extend database systems to the wide-area environment. In fact, there are a spectrum of techniques for making query processing more adaptive, ranging from delayed-binding, to adaptive re-optimization and beyond. One interesting recent development is the "Continuous Query Optimization" (CQO) developed by Avnur and Hellerstien [AH00], which foregoes traditional optimization for an adaptive queue-based scheduler that in effect learns an efficient query plan during the query execution. We plan to investigate the integration of XJoin with such mechanisms as part of the Telegraph project at Berkeley.

References

[AFTU96] L. Amsaleg, M. J. .Franklin, A. Tomasic, and T. Urhan. Scrambling Query Plans to Cope With Unexpected Delays. *PDIS Conf.*, Miami, USA, 1996.

[AFT98] L. Amsaleg, M. J. .Franklin, and A. Tomasic. Dynamic Query Operator Scheduling for Wide-Area Remote Access. *Journal of Distributed and Parallel Databases*, Vol. 6, No. 3, July 1998.

[AH00] R. Avnur, J. Hellerstien. Continous Query Optimization. *ACM SIGMOD Conf.*, Dallas, TX, 2000.

[BDT83] D. Bitton, D. J. DeWitt, C. Turbyfill. Benchmarking Database Systems, a Systematic Approach. *VLDB Conf.*, Florence, Italy, 1983.

[BM96] R. Bayardo, and D. Miranker. Processing Queries for the First Few Answers. *Proc. 3rd CIKM Conf.*, Rockville, MD, 1996.

[CK97] M. J. Carey, and D. Kossman. On Saying "Enough Already!" in SQL. *ACM SIGMOD Conf.*, Tucson, AZ, 1997.

[HH99] P. J. Haas, J. M. Hellerstein. Ripple Joins for Online Aggregation. *ACM SIGMOD Conf.*, Philadelipha, PA, 1999.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. *ACM SIGMOD Conf.*, Tucson, AZ, 1997.

[HS93] W. Hong, M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1):9-32, 1993.

[IFFL+99] Z. Ives, D. Florescu, M. Friedman, A. Levy, D. S. Weld. An Adaptive Query Execution System for Data Integration. *ACM SIGMOD Conf.*, Philadelphia, PA, 1999.

[SP97] P. Seshadri, M. Paskin. PREDATOR: An OR-DBMS with Enhanced Data Types. *ACM SIGMOD Conf.*, Tucson, Arizona, 1997.

[UF99] T. Urhan, M. J. .Franklin. XJoin: Getting Fast Answers from Slow and Bursty Networks. *University of Maryland Technical Report, CS-TR-3994.*, February, 1999.

[UFA98] T. Urhan, M. J. .Franklin, and L. Amsaleg. Cost Based Query Scrambling for Initial Delays. *ACM SIGMOD Conf.*, Seattle, WA, 1998.

[VL93] S. V. Vrbsky, and J. W. S. Liu. Approximate, A Query Processor that Produces Monotonically Improving Approximate Answers. *IEEE Transactions od Knowledge and Data Engineering*, Vol.5, No.6, December 1993.

[WA91] A. N. Wilschut, and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. *1st Int'l Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, 1991.

A Decision Theoretic Cost Model for Dynamic Plans

Richard L. Cole
Informix Software Inc.
485 Alberto Way
Los Gatos, CA 95032
rickcole@informix.com

Abstract

Since the classic optimization work in System R, query optimization has completely preceded query evaluation. Unfortunately, errors in cost model parameters such as selectivity estimation compromise the optimality of query evaluation plans optimized at compile time. The only promising remedy is to interleave strategy selection and data access using run-time-dynamic plans. Based on the principles of decision theory, our cost model enables careful query analysis and prepares alternative query evaluation plans at compile time, delaying relatively few, selected decisions until run time. In our prototype optimizer, these run-time decisions are based only on those materialized intermediate results for which materialization costs are expected to be less than the benefits from the improved decision quality.

1 Introduction

Query processing can be divided into query optimization and query execution; this dichotomy has been followed almost exclusively ever since the classic System R optimizer paper [SAC+79]. For queries with 10, 20, or 100 operators, this clear-cut distinction is no longer applicable. If each operator's selectivity is consistently underestimated by a mere 10%, the error after 10 operators is a factor of 2.8; the error after 20 operators is a factor of 8.2; and after 100 operators, it is a factor of 37,649. Beyond a level of query complexity somewhere between 10 and 20 operators, selectivity and cost estimates are more guesses than predictions, and plan choices are more gambles than strategic decisions [CVL+84, MCS88, IK91, Loh89]. Note that queries involving views, CASE tools, and data warehouse queries [KRRT98, Bri99] in particular often involve hundreds of operators. Thus, the problem addressed here is of great interest to database system vendors and implementors.

Not only estimates for intermediate result sizes, but also forecasts for available resources may err. The resource situation, e.g., available memory, processors, temporary disk storage, net bandwidth, internet site availability, and so on may change between compilation and execution, even from the time when a complex query is started to the time when the query completes its last operation. Since cost calculations and decisions among alternative plans depend on input sizes and available resources, selected optimization decisions and resource allocation decisions must be shifted from compile time into run time.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Research into this problem has taken several paths: improving the optimizer's statistical profile of the database system [MCS88, MRL98]; building static plans using a least-expected-cost metric [CHS99]; building start-up-time-dynamic plans [CG94]; materializing all intermediate results, i.e., Ingres Decomposition [WY76]; using intra-algorithm, i.e., record level adaptation [DG94, HRR99, AH00], decision theoretic techniques based on sampling [SBM93]; competitions between algorithms [Ant96]; heuristic approaches to materializing intermediate results [UFA98, BFMV00]; and partial plan re-optimization at run time [KD98, NWMN99, IFF$^+$99]. Note that many of these techniques are compatible with each other.

In the industrial arena, only the Informix® Red Brick® Decision Server™[Bri99] delays preselected optimization decisions until run time. Red Brick applies information available at run time specifically to optimization decisions for star-schema queries [KRRT98, Bri99]. The server materializes the intermediate results of preliminary plans, each one composed from a star-schema *dimension table*, i.e., a primary key defining table referenced by a foreign key defining *fact table*. (Note that fact tables may contain billions of records.) Based on the materialized intermediate results, cost based, run-time decisions are made between compile-time generated alternative plans. Run-time decisions are also made between fact table orders, alternative join indexes, view maintenance algorithms, as well as the final determination of data partitioning functions and degrees of query parallelism.

In combination with join algorithms specialized for star-schema query processing, the run-time techniques employed by the Red Brick server enable efficient query evaluation of complex data warehouse queries involving many fact and dimension tables. One successful application's schema design involves 33 fact tables and 160 dimension tables. In this context, queries routinely join a many as 32 tables. Unfortunately, because of the difficulty in quantifying the advantages of these run-time techniques, their application is limited to star-schema query plans and sub-plans. What we need is a generalized compile-time approach to quantifying the advantages of such run-time decisions.

In this short paper, we briefly describe an extension of our previous work on start-up-time-dynamic plans [CG94], in order to address the problems posed by complex queries and uncertain cost model parameters. *Start-up-time-dynamic query evaluation plans* are dynamic plans implementing decisions typically made at compile time that are delayed until the start of a query's evaluation. *Run-time-dynamic query evaluation plans* are plans that make decisions between alternative, potentially optimal, algorithms, operator ordering, and plan shapes based upon additional knowledge obtained while evaluating the plan. For example, in our prototype run-time-dynamic plan optimizer [Col99], this additional knowledge is obtained by materializing intermediate results, thus providing exact cardinality and allowing accurate planning of subsequent query evaluation. Materializing intermediate results may also supply additional information about the distributions of attributes contained in the result, e.g., dynamically constructed histograms of join attribute values may facilitate improved costing of join algorithm alternatives [KD98].

The difficult aspect of this approach is determining which subsets of plan operators to materialize. Because interrupting the plan's data-flow and intermediate data materialization has an associated cost, the compile-time optimizer must trade off this cost against the benefit of additional knowledge and subsequently improved query evaluation. The techniques we employ to solve this problem are provided by statistical decision theory and it is the generalized application of decision theory to query optimization that is the key contribution of this work.

2 Cost Model

A cost model which optimizes run-time-dynamic plans must first capture the benefit of run-time decisions and, second, the cost of obtaining run-time information to make these decisions. Because we restrict ourselves to complete materialization of data, rather than partial sampling, we simplify the cost calculation of obtaining run-time information. It is simply the cost of the materialization itself, which is a function of the expected number and size of records to be read and written to a temporary result. In order to capture the benefit of run-time decisions we use the techniques of decision theory [RS68, Lin72].

2.1 Decision Theory

There are several elements involved in making a decision (largely from Raiffa and Schlaifer [RS68]): the set of possible actions, $A = \{a\}$, where a is an action chosen from the domain of actions A; the current state of the world $\Theta = \{\theta\}$, where θ is a particular state from domain Θ and choosing an action a depends a θ that may not be predicted with certainty; a set of possible experiments, $E = \{e\}$, that may improve our knowledge of Θ; the potential outcome of an experiment, $Z = \{z\}$; for every combination of a, θ, e, and z there is a consequence, often expressed as the utility, $u(\cdot,\cdot,\cdot,\cdot)$, defined on $A \times \Theta \times E \times Z$, that captures the benefit and cost of performing an experiment, observing an outcome, and taking a particular action; and we have prior estimates of the probabilities of the various states of the world before and after an experiment.

How do these principles apply to our problem? To optimize run-time-dynamic plans we apply them as follows: the domain A is the set of alternative query evaluation algorithms and plans; the state of the world, Θ, consists of known or estimated distributions for the parameters of the cost model, e.g., predicate selectivities or resources such as available memory; the set of possible experiments, E, is comprised of methods for obtaining better estimates for distributions on Θ that are only known approximately, e.g, sampling techniques, or as in our case, intermediate result materialization; the cost of the experiment, e, depends on the potential materialized outcome, z, since its cost is a function of the number of records materialized; in decision theory one typically wants to maximize the utility, $u(a, \theta, e, z)$, however, rather than defining utility as negative cost, $-c(a, \theta, e, z)$, we choose to minimize positive cost, $c(a, \theta, e, z)$, a perspective compatible with typical query optimizers; and we have prior probabilities for the uncertain parameters of the cost model, such as predicate selectivities or available resources, that capture their accuracy.

2.2 The Value of Information

Since the utility or cost of our experiments and actions are additive, we can think about the increase in the utility, or in our case decrease in optimizer cost, of an action for a given experiment. In decision theory, this increase is called the *value of information*. For a given experimental outcome we can compute the *conditional value of information*, but since we do not know these outcomes a priori we must resort to computing the *expected value of information*. Before running an experiment, e.g., before materializing intermediate results, we typically have some idea about the expected value of the cost model parameter, in this case intermediate result cardinality, and the accuracy with which we know the expected value. That is, we have a probability distribution for the expected value.[1] Using this previously known distribution, referred to as the *prior distribution*, we compute the *conditional cost with original information* (CCWOI), conditional with respect to a and θ, and the *expected cost with original information* (ECWOI) as follows.

The form $c(a|\theta)p(\theta)$ is the computation of CCWOI for cost that is a function of the action taken, a (the algorithm used), and the cardinality of the algorithm's input, θ, weighted by the probability density function representing our prior knowledge of the distribution of that cardinality, $p(\theta)$. Integrating their product over the range of θ (the state of the world), and minimizing over a (the algorithm set), gives the expected cost of making a particular algorithm choice, i.e., the ECWOI is $min_a \int_\Theta c(a|\theta)p(\theta)\, d\theta$.

Even without further development, the expected cost with original information is already a powerful decision theoretic concept when applied to a cost-based query optimizer. By minimizing the ECWOI, a query optimizer can build a static plan that has increased robustness without requiring start-up-time or run-time decisions. (This concept is similar to that explored by Chu et al.[CHS99]).

If we materialize intermediate query results, then for decisions between algorithms that directly depend on these results we will have perfect information for making decisions at run time. We can compute the *conditional cost with perfect information* (CCWPI), conditional on θ, and the *expected cost with perfect information* (ECWPI) as $min_a c(a|\theta)p(\theta)$ and $\int_\Theta min_a c(a|\theta)p(\theta)\, d\theta$ respectively. Note the relocation of the min_a term in

[1] In lieu of continuous probability distributions we could instead use discrete distributions, as provided by histograms, etc.

these equations when compared to the ECWOI. From the ECWPI and ECWOI we may compute the *expected value of perfect information* or EVPI as ECWOI − ECWPI, providing an upper bound on the value of a perfect experiment.

2.3 The Value of Materialized Information

Materializing data may be an effective technique to obtain information for decision making immediately after materialization, in which case we have perfect information about cardinality and perhaps other information as well, e.g., domain histograms. However, the effectiveness of materializing data is reduced as subsequent operators operate on the materialized result. Decisions between alternative algorithms situated more than a single operator above (in the graph of operators) become less than perfect as intervening operators may introduce additional uncertainty, and the cost of materialization may ultimately outweigh its reduced benefit. The effect of this uncertainty is captured by the *expected cost with materialized information* (ECWMI).

To develop the ECWMI we use the *extensive* form of statistical analysis [Lin72]. Given the decision tree in

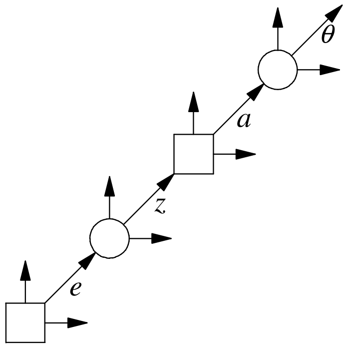
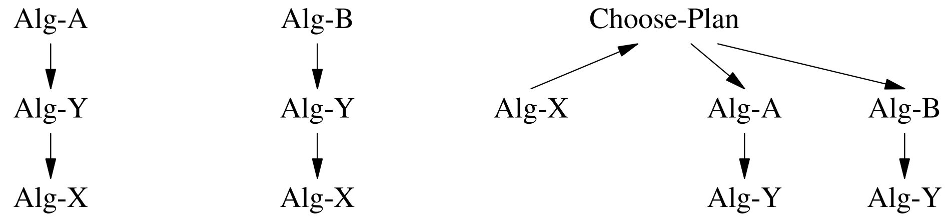


Figure 1: Experimental Decision Tree

Figure 1, where circles denote random variables and rectangles denote decisions, a formulation is developed to maximize the value of the experiment e. (Causality is in the direction of the arrows, from left to right and bottom to top.) First an experiment, e, is chosen, which produces experimental data, z. Based upon this information we choose an action, a, resulting in outcome θ and utility $u(a, \theta, e, z)$. Working backwards by first averaging the utility over θ where the applicable distribution is $p(\theta|z, e)$, it is then maximized over the set of available actions a, i.e., max_a, followed by averaging over the values of z, where the applicable distribution is $p(z|e)$, maximized over the set of experiments e, i.e., max_e, producing the equation $max_e \int_Z max_a \int_\Theta u(a, \theta, e, z)p(\theta|z, e)\, d\theta\, p(z|e)\, dz$, maximizing the expected utility.

The optimization of run-time-dynamic plans is analogous to the decision tree of Figure 1. The experiment e performed is the materialization of intermediate query results, the result of this experiment, z, is the size of the preliminary plan's intermediate result, the set of actions, a, are the set of alternative sub-plans available to Choose-Plan, and the final outcome, θ, is the actual result cardinality. Because we have chosen a priori to materialize intermediate results, the experiment, e, is fixed, i.e., we either materialize an intermediate result entirely or not at all. Therefore, we may dispense with the max_e term and references to e in general. A further simplification can be made because we know utility is not directly influenced by the experimental outcome. The cost of an algorithm in this example is only dependent on the cardinality of its immediate input. Therefore the ECWMI is $\int_Z min_a \int_\Theta c(a, \theta)p(\theta|z)\, d\theta\, p(z)\, dz$, and the *expected value of materialized information* (EVMI) is ECWOI − ECWMI.

2.4 An Extensive Form Example



(a) Static Plan A (b) Static Plan B (c) Run-Time-Dynamic Plan

Figure 2: Run-Time-Dynamic Plan with Additional Uncertainty

Consider Figure 2 with static plans in diagrams (a) and (b), composed from generic algorithms Alg-A and Alg-B both having inputs Alg-Y and Alg-X, and in diagram (c) the run-time-dynamic plan having two alternatives, one of which will be chosen at run time based upon the cardinality of preliminary plan Alg-X materialized by the Choose-Plan operator as part of its decision procedure. If algorithm Alg-Y also introduces uncertainty in the input cardinality and uncertainty in the costs of algorithms Alg-A and Alg-B, then its effect must be considered in the design of the cost model.[2] This additional uncertainty is exactly what the ECWMI captures. In this example, the experiment performed is the materialization of the intermediate result produced by Alg-X, therefore we replace z in our ECWMI with x in order to associate it with Alg-X. Similarly, the final outcome is the intermediate result cardinality of Alg-Y and we substitute y for θ in our original definition of the ECWMI. With these modifications, the ECWMI of the example run-time-dynamic plan in Figure 2 (c) is $\int_X min_a \int_Y c(a, y)p(y|x)\, dy\, p(x)\, dx$, and after substituting for the cost of the specific alternative algorithms we have
$\int_X min[\int_Y c(\text{Alg-A}|y)p(y|x)\, dy, \int_Y c(\text{Alg-B}|y)p(y|x)\, dy]\, p(x)\, dx.$

2.5 A Prototype Plan and Preliminary Results

Figure 3 presents the graph of a dynamic plan produced by a prototype optimizer that applies the decision theoretic concepts just described [Col99]. Each node in the graph represents a typical operator [Gra93], with its name abbreviated as follows: EX - Execute (performs plan initialization), CP - Choose-Plan (evaluate preliminary plan(s) and choose from alternative plans) HJ - Hash-Join, FL - Filter (apply a predicate), FS - File-Scan (sequentially scan a file), FJ - Functional-Join (row identifier based join), and BS - Btree-Scan (with range restricting predicate). Each edge in a plan graph has an associated numeric label indicating the absolute order of inputs for an operator. For example, label 0 indicates the first input of an operator, label 1 the second, and so on. Preliminary plans, i.e., materialized sub-plans, are identified by dashed edges proceeding directly from the root node of a preliminary plan to the Choose-Plan operator(s) benefiting from its materialization.

The plan is for a four table join query having equijoin predicates and other, local predicates on all base tables. For one of these local predicates, the statistical profile of the database indicated much uncertainty in the predicate's selectivity. In this run-time-dynamic plan, there are many plan alternatives because of the decisions to be made between different base data algorithms, Functional-Join:Btree-Scan versus Filter:File-Scan, and different join orders for the Hash-Joins. Note that the operator that implements the uncertain predicate has been identified and its intermediate result is planned to be materialized. The materialized information is shown to be propagated to all Choose-Plan operators and determines all run-time choices. In the decision theoretic sense, the evaluation of this plan will be optimal.

[2] Alg-Y need not be a single operator. It is representative of an arbitrarily complex sub-plan. Our decision analysis applies whether or not Alg-Y is a single operator or a sub-plan composed of multiple operators.

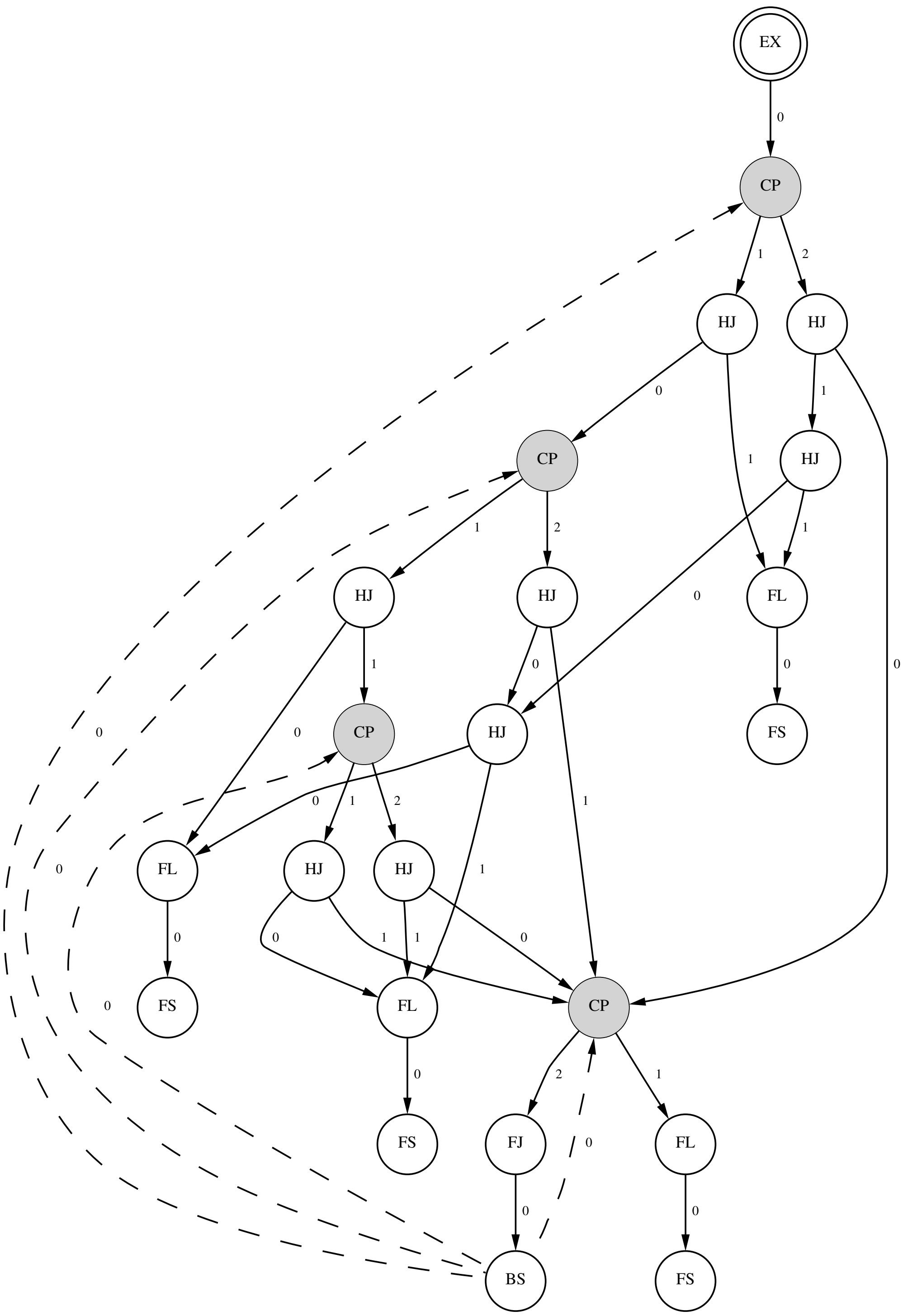


Figure 3: Discovering Localized Uncertainty: A Prototype Dynamic Plan

Such qualitative results are encouraging, as are preliminary quantitative results. Large cost improvements have been realized for reasonable increases in compile-time optimization. For simple queries of a single table, experiments are easily constructed that produce relative cost improvements, by comparing the cost of dynamic plans to their analogous static plans, of more than a factor of three for very little additional optimization effort. More complex queries can produce similar improvements for an increase in optimization time that is reasonable depending on the context. For example, a query having 20 tables, 39 predicates, and 60 operators derived from 10^{21} equivalent physical forms was optimized in 20 seconds, compared to 1 second for the analogous static plan. In absolute terms, 20 seconds is an acceptable optimization time, especially when the result of that optimization is amortized over many query evaluations via precompiled plans.

3 Conclusion

The application of decision theory to query optimization was first considered by Seppi et al.[SBM93]. The focus of Seppi's work was on the application of Bayesian decision theory to three selected database query optimization problems: making decisions between alternative algorithms for evaluating simple selection predicates, join predicates over distributed relations, and the transitive closure of binary relations. The emphasis was on preposterior analysis, used to compute the optimal amount of sampling information to obtain before making a decision. (A later application of decision theory to query optimization was proposed by Chu et al.[CHS99]; however, the resulting plans remain static and there appears to be no adaptation during run time.)

Rather than focusing on any one specific query algorithm, we take a general approach that is independent of the algorithms under consideration, one that is useful for modeling the effective cost of inter-operator uncertainty. We also avoid dependency on any one specific distribution or family of data distributions, e.g., the exponential family of distributions as in Seppi's work. Finally, our emphasis is on the terminal analysis, i.e., analysis of the expected cost of algorithm decisions, rather than on the preposterior analysis, i.e., analysis of the expected effect of different sampling plans.

The accumulation of even small errors in the estimation of a predicate's selectivity can result in static query evaluation plans that are poorly optimized. Run-time-dynamic query evaluation plans are a promising approach to solving this problem of uncertain cost model parameters. To our knowledge, there has been no other work that proposes the generalized application of decision theory to query optimization for run-time adaptation.

References

[AH00] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *to appear Proc. ACM SIGMOD*, Dallas, TX, May 2000.

[Ant96] G. Antoshenkov. Query processing and optimization in Oracle RDB. *The VLDB Journal*, 5(4):229–237, January 1996.

[BFMV00] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. Dynamic query scheduling in data integration systems. In *Proc. IEEE Int'l. Conf. on Data Eng.*, pages 425–434, San Diego, CA, February 2000.

[Bri99] Red Brick. *Informix Red Brick Decision Server (Version 6.0): Administrator's Guide*. Informix Software, Inc., Los Gatos, CA, 1999.

[CG94] R. L. Cole and G. Graefe. Optimization of dynamic query execution plans. In *Proc. ACM SIGMOD*, Minneapolis, MN, May 1994.

[CHS99] F. Chu, J. Y. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Sys.*, Philadelphia, Pennsylvania, May 1999.

[Col99] R. L. Cole. Optimizing dynamic query evaluation plans. *Ph.D. Thesis, University of Colorado at Boulder*, 1999.

[CVL+84] S. Christodoulakis, J. Vanderbroek, J. Li, T. Li, S. Wan, Y. Wang, M. Papa, and E. Bertino. Development of a multimedia information system for an office environment. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 261, Singapore, August 1984.

[DG94] D. L. Davison and G. Graefe. Memory-contention responsive hash joins. In *Proc. Int'l. Conf. on Very Large Databases*, page 379, Santiago de Chile, September 1994.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[HRR99] J. M. Hellerstein, V. Raman, and B. Raman. Online dynamic reordering for interactive data processing. *Proc. Intl' Conf. on Very Large Data Bases*, September 1999.

[IFF+99] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proc. ACM SIGMOD*, pages 299–310, Philadelphia, Pennsylvania, May 1999.

[IK91] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proc. ACM SIGMOD*, page 168, Denver, CO, May 1991.

[KD98] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. ACM SIGMOD*, pages 106–117, Seattle, WA, June 1998.

[KRRT98] R. Kimball, L. Reeves, M. Ross, and W. Thornthwaite. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses*. John Wiley & Sons, Inc., August 1998.

[Lin72] D. V. Lindley. *Bayesian Statistics, A Review*. SIAM, Philadelphia, PA, 1972.

[Loh89] G. M. Lohman. Is query optimization a 'solved' problem? In G. Graefe, editor, *Proc. Workshop on Database Query Optimization*, page 13, Beaverton, OR, May 1989. Oregon Graduate Center CS TR 89-005.

[MCS88] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191, September 1988.

[MRL98] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. ACM SIGMOD*, pages 426–435, Seattle, WA, June 1998.

[NWMN99] K. W. Ng, Z. Wang, R. R. Muntz, and S. Nittel. Dynamic query re-optimization. *Proc. Conf. on Scientific and Statistical Database Management*, July 1999.

[RS68] H. Raiffa and R. Schlaifer. *Applied Statistical Decision Theory*. MIT Press, Cambridge, MA, 1968.

[SAC+79] 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. ACM SIGMOD*, page 23, Boston, MA, May-June 1979. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

[SBM93] K. Seppi, J. Barnes, and C. Morris. A Bayesian approach to query optimization in large scale data bases. *ORSA J. of Computing*, Fall 1993.

[UFA98] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proc. ACM SIGMOD*, Seattle, WA, June 1998.

[WY76] E. Wong and K. Youssefi. Decomposition - A strategy for query processing. *ACM Trans. on Database Sys.*, 1(3):223, September 1976.

A Dynamic Query Processing Architecture for Data Integration Systems

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

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

[⋆]INRIA
Rocquencourt, France
FirstName.LastName@inria.fr

[‡]IBM Almaden Research
USA
Mohan@almaden.ibm.com

Abstract

Execution plans produced by traditional query optimizers for data integration queries may yield poor performance for several reasons. The cost estimates may be inaccurate, the memory available at run-time may be insufficient, or the data delivery rate can be unpredictable. All these problems have led database researchers and implementors to resort to dynamic strategies to correct or adapt the static QEP. In this paper, we identify the different basic techniques that must be integrated in a dynamic query engine. Following on our recent work [6] on the problem of unpredictable data arrival rates, we propose a dynamic query processing architecture which includes three dynamic layers: the dynamic query optimizer, the scheduler and the query evaluator. Having a three-layer dynamic architecture allows reducing significantly the overheads of the dynamic strategies.

1 Introduction

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

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

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

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Second, the characteristics of the sub-query results are difficult to assess, due to the autonomous nature of the data sources. The sizes of intermediate results used to estimate the costs of the integration QEP are then likely to be inaccurate. As the amount of available memory at runtime for processing the integration query may be much less than that assumed at compile time, executing the query as is might cause thrashing of the system because of paging [4,12].

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

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

2 Problem Formulation

An integration query is nothing else than a standard centralized query except that the data are collected in remote sources instead of being extracted from local storage units. In this section, we first present standard query processing techniques and show their problems. Query processing is classically done in two steps. The query optimizer first generates an "optimal" QEP for a query. The QEP is then executed by the query engine which implements an execution model and uses a library of relational operators [7].

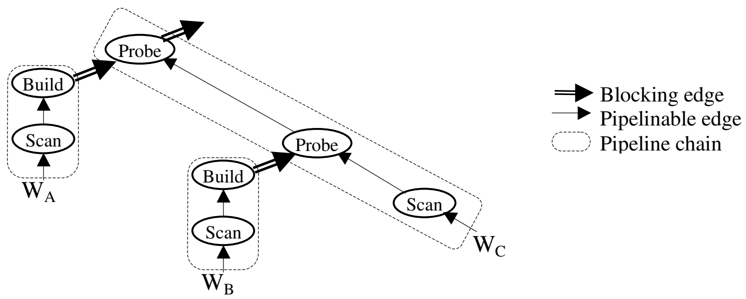


Figure 1: A simple Query Execution Plan (QEP)

A QEP is represented as an operator tree. Nodes represent atomic physical operators and edges represent data-flow. Two kinds of edges are distinguished: blocking and pipelinable. A blocking edge indicates that the data is entirely produced before it can be consumed. Thus, an operator with a blocking input must wait for the entire operand to be materialized before it can start. A pipelinable edge indicates that data can be consumed "one-tuple-at-a-time". Therefore, the consumer can start as soon as one input tuple has been produced. Figure 1 shows a QEP for a query integrating three data sources represented by wrappers W_A, W_B and W_C, respectively. The three wrapper's results are joined using asymmetric join operators (e.g., hash-join) that have one blocking input and one pipelinable input, and produce a pipelinable output. Such QEP can be decomposed in three fragments called *pipeline fragments*[1] (PF's): p_A, p_B and p_C. A pipeline fragment (PF for short) is a maximal set of physical operators linked by pipelinable edges.

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

The most popular execution model for processing relational queries is the *iterator model*[7]. It resembles closely those of commercial systems, e.g., Ingres, Informix and Oracle. In the iterator model, each operator of the QEP is implemented as an iterator supporting three different calls: open() to prepare the operator for producing data, next() to produce one tuple and close() to perform a final clean-up. A QEP is activated starting at the QEP root and progressing towards the leaves. The iterator model allows for pipelined execution. Moreover, the shape of the QEP fixes the order of execution, generally recursively from left to right. For example, a query engine implementing the iterator model would execute the QEP of Figure 1 following the order p_A, p_B, p_C. So, standard query processing techniques plan completely the execution and thus, cannot react to unpredictable situations which may compromise planning decisions taken at several levels:

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

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

3 Dynamic Query Processing

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

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

These techniques are complementary and should be used together to provide good performance [8,15]. Indeed, partial re-optimization of the query plan is difficult to implement and tune [15]. Moreover, the possibility for re-optimization decreases as query execution reaches completion (because of the results previously computed). Solving most problems at the operator or scheduling level can alleviate the need for dynamic re-optimization.

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

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

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

3.1 Unpredictable Data Arrival Rate

The iterator model produces a sequential execution. Such an execution, i.e., consuming entirely the data produced by one wrapper before accessing another one, leads to a response time with a lower bound equal to the sum of the times needed to retrieve the data produced by each wrapper. Thus, for a given wrapper, if the time to retrieve the data is larger than the time to process it, the query engine stalls. Handling delays in data arrival is therefore crucial for obtaining good performance. A first solution to the problem raised by a sequential execution is to interleave the execution of several PF's. For instance, with the QEP of Figure 1, p_A and p_B can be triggered concurrently at the beginning of the execution. If delivery delays occur while retrieving data from W_A, the query engine can avoid stalling by executing p_B. However, this approach is limited by the number of PF's which can be executed concurrently due to memory limitation or dependency constraints. For non-executable PF's (e.g., p_C which must be executed alone for dependency constraint reasons), the delays can be amortized by triggering a materialization of the associated wrapper's result (e.g., W_C). Such a materialization occurs while executing concurrently other executable PF's (e.g., p_A and p_B). However, scheduling materializations can incur high I/O overheads. Thus, these overheads must be estimated and compared with the potential benefit. Since the data delivery rate is typically unpredictable and may vary during the query's execution, we must monitor it along the execution and react to any important variation. Thus, the materialization of a given wrapper's result must stop as soon as the PF becomes executable or the benefit becomes too small (if for instance, the delivery rate increases).

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

3.2 Dynamic Query Engine Architecture

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

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

The *Dynamic Query Optimizer* (DQO) may implement dynamic re-optimization strategies such as the ones described in [4,8,9,15]. Each planning phase of the DQO can potentially modify the QEP, which is passed on to the dynamic query scheduler.

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

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

The DQO, the DQS and the QFE interact synchronously, i.e., they never run concurrently. The DQO calls the DQS passing the QEP as an argument. The DQS, in turn, calls the QFE passing the SP as an argument. The QFE, then evaluates the query fragments of the SP and returns an interruption event informing the DQS of the reason why the execution phase must be terminated. The interruption event can be processed by the DQS, or returned to the DQO depending on its nature, thereby starting new planning phases.

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

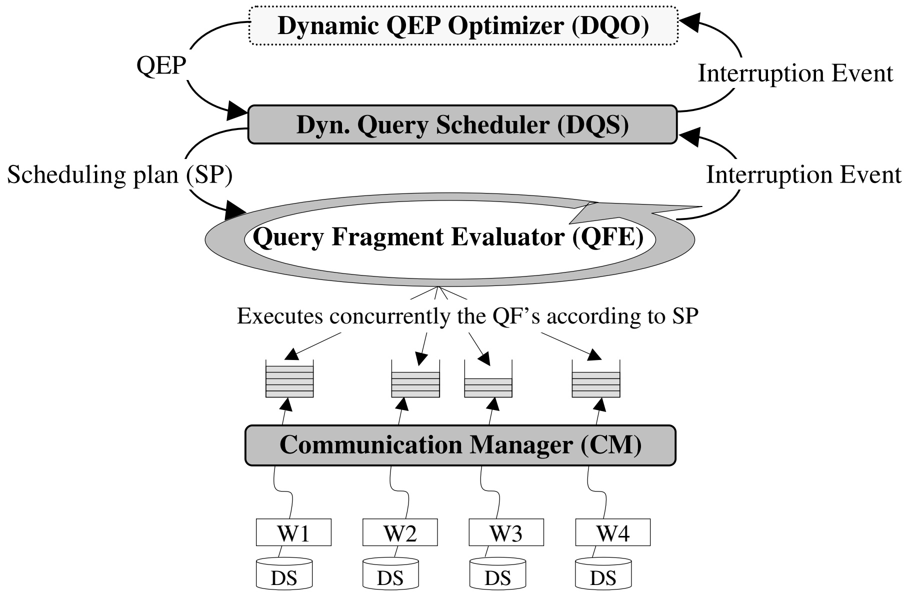


Figure 2: Dynamic Query Engine Architecture

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

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

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

4 Implementing Dynamic Strategies

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

4.1 Query Fragment Evaluator

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

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

4.2 Dynamic Query Optimizer and Scheduler

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

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

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

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

5 Conclusion

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

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

References

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

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

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

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

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

[6] L. Bouganim, F. Fabret, C. Mohan, P. Valduriez. Dynamic Query Scheduling in Data Integration Systems. Int. Conf. on Data Engineering, 2000.

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

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

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

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

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

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

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

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

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

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

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

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

CALL FOR PAPERS

DATA ENGINEERING

The 17th International Conference on Data Engineering

April 2-6, 2001, Heidelberg, Germany

Sponsored by the IEEE Computer Society

IEEE

The 2001 International Conference on Data Engineering will be held in a beautiful old town - Heidelberg. This is at the center of Germany's most active high-tech region where you find companies such as SAP, Software AG, Computer Associates, SAS, numerous software startups, many world-class research centers, and a number of universities with strong database research groups. Such combination of strong industry, ground breaking research institutions, economic prosperity, and a beautiful host town provide an ideal environment for a conference on Data Engineering.

ICDE 2001 provides a premier forum for:

- presenting new research results
- exposing practicing engineers to evolving research, tools, and practices and providing them with an early opportunity to evaluate these
- exposing the research community to the problems of practical applications of data engineering
- promoting the exchange of data engineering technologies and experience among researchers and practicing engineers

The conference will also include an industrial track that will give participants the opportunity to visit companies and research labs that are actively working on such data engineering problems.

Topics of Interest

- XML, metadata, & semistructured data
- Database engines & engineering
- Query processing
- Data warehouses, data mining, and knowledge discovery
- Advanced IS middleware
- Scientific and engineering databases
- Extreme databases
- E-commerce & e-services
- Workflow & process-oriented systems
- Emerging trends
- System applications and experience

Important Dates

Submission of abstracts:	Jul 28, 2000
Submission of papers:	Aug 4, 2000
Panel/tutorial proposals:	Sep 4, 2000
Demo proposals:	Sep 4, 2000
Notification of acceptance:	Oct 18, 2000
Camera-ready copies:	Dec 18, 2000

Conference Officers

General Chairs: Andreas Reuter, EML and International U., Germany; David Lomet, Microsoft Research, USA

Program Co-Chairs: Alex Buchmann, U. Darmstadt, Germany; Dimitrios Georgakopoulos, MCC, USA

Industrial Program Co-chairs: Peter Lockemann, U. Karlsruhe, Germany; Tamer Ozsu, U. of Alberta, Canada

Panel Program Chair: Erich Neuhold, GMD-IPSI, Germany

Tutorial Program Chair: Guido Moerkotte, U. Mannheim, Germany; Eric Simon, INRIA, France

Steering committee liaison: Erich Neuhold, GMD-IPSI, Germany; Marek Rusinkiewicz, MCC, USA

Organizing Chair: Isabel Rojas, European Media Laboratory, Germany

Demos & Exhibits: Wolfgang Becker, International U., Germany; Andreas Eberhart, International U., Germany

Program Vice-Chairs

XML, metadata, & semistructured data:
Dan Suciu, AT&T Labs-Research, USA

Database engines & engineering:
Bruce Lindsay, IBM Almaden, USA

Query processing:
Joseph Hellerstein, U. of California Berkeley, USA

Data warehouses, data mining, and knowledge discovery:
Rakesh Agrawal, IBM Almaden, USA

Advanced IS middleware:
Krithi Ramamritham, U. of Massachusetts, USA and IIT Bombay, India

Scientific and engineering databases:
Theo Haerder, U. of Kaiserslautern, Germany

Extreme databases:
Martin Kersten, CWI, Netherlands

E-commerce & e-services:
Umesh Dayal, Hewlett-Packard Laboratories, USA

Workflow & process-oriented systems:
Gustavo Alonso, ETH Zentrum, Switzerland

Emerging trends:
Katia Sycara, Carnegie Mellon U., USA

System applications and experience:
José Blakeley, Microsoft, USA

Paper Submission

Paper submissions must be received by September 29, 2000, 12:00 PM Central US Time. Paper length must not exceed 15 pages (including figures, tables, etc.) in not smaller than 11 pt font. Papers can be submitted in hardcopy or electronic form. In either case, an electronic abstract of no more than 300 words in ascii text has to be submitted until Sept. 22nd. The abstract submission must include the title of the paper, authors' names, an e-mail address of the contact author, a first and second preference among the twelve topical areas. For electronic submission, go to the paper submission web site and follow the instructions there. Papers must be submitted in pdf format. If electronic submission is not possible, seven (7) copies should be sent to one of the program co-chairs:

Alex Buchmann
Darmstadt University of Technology
Department of Computer Science
Database Research Group
Wilhelminenstr. 7
64283 Darmstadt, Germany
buchmann@informatik.tu-darmstadt.de

Dimitrios Georgakopoulos
MCC
3500 West Balcones Center Dr.
Austin, Texas 78759-5398, USA
dimitris@mcc.com

Conference Web Site: http://www.congress-online.com/ICDE2001

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398