Bulletin of the Technical Committee on

Data Engineering

September 1997 Vol. 20 No. 3

IEEE Computer Society

Letters

Letter from the Editor-in-Chief . . . . *David Lomet* 1
Letter from the Special Issue Editor . . . . *Michael Franklin* 2

Special Issue on Improving Query Responsiveness

Improving Responsiveness for Wide-Area Data Access . . . .
. . . . *Laurent Amsaleg, Philippe Bonnet, Michael J. Franklin, Anthony Tomasic,Tolga Urhan* 3
Processing *Top N* and *Bottom N* Queries . . . . *Michael J. Carey and Donald Kossmann* 12
Online Processing Redux . . . . *Joseph M. Hellerstein* 20
Dynamic Query Processing in DIOM . . . . *Ling Liu and Calton Pu* 30
Dynamic Query Optimization in Multidatabases . . . .
. . . . *Fatma Ozcan, Sena Nural, Pinar Koksal, Cem Evrendilek, Asuman Dogac* 38
Execution Reordering for Tertiary Memory Access . . . . *Sunita Sarawagi* 46

Conference and Journal Notices

Statements of the Candidates for TC Chair 54
Biographies of the Candidates for TC Chair 55
TC on Data Engineering Election Ballot . . . . back cover

Editorial Board

Editor-in-Chief
David B. Lomet
Microsoft Corporation
One Microsoft Way, Bldg. 9
Redmond WA 98052-6399
lomet@microsoft.com

Associate Editors
Daniel Barbará
Bellcore
445 South St.
Morristown, NJ 07960

Michael Franklin
Department of Computer Science
University of Maryland
College Park, MD 20742

Joseph Hellerstein
EECS Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776

Betty Salzberg
College of Computer Science
Northeastern University
Boston, MA 02115

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.

TC Executive Committee

Chair
Rakesh Agrawal
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
ragrawal@almaden.ibm.com

Vice-Chair
Nick J. Cercone
Assoc. VP Research, Dean of Graduate Studies
University of Regina
Regina, Saskatchewan S4S 0A2
Canada

Secretry/Treasurer
Amit Sheth
Department of Computer Science
University of Georgia
415 Graduate Studies Research Center
Athens GA 30602-7404

Geographic Co-ordinators
Shojiro Nishio (**Asia**)
Dept. of Information Systems Engineering
Osaka University
2-1 Yamadaoka, Suita
Osaka 565, Japan

Ron Sacks-Davis (**Australia**)
CITRI
723 Swanston Street
Carlton, Victoria, Australia 3053

Erich J. Neuhold (**Europe**)
Director, GMD-IPSI
Dolivostrasse 15
P.O. Box 10 43 26
6100 Darmstadt, Germany

Distribution
IEEE Computer Society
1730 Massachusetts Avenue
Washington, D.C. 20036-1992
(202) 371-1013

Letter from the Editor-in-Chief

Technical Committee Election

The Technical Committee on Data Engineering is having an election for a new TC Chair. Our current Chair, Rakesh Agrawal, has completed his term. I urge current members of the TCDE to vote. Two very able candidates have been nominated who have an interest in serving as TC Chair, Erich Neuhold and Betty Salzberg. Betty and Erich have provided statements and biographies on their candidacies that appear starting on page 55. **You may vote using the ballot included on the inside back cover. Mail it to the Computer Society, fax it to (+1-202-728-0884), or retrieve a Word document ballot from the Bulletin web site (http://www.research.microsoft.com/research/db/debull) and email it to (twoods@computer.org).**

This Issue

The query processing community is cursed in that whatever query optimization is realized, there are always queries that perform badly. Indeed, there are relatively simple queries that slip between the cracks of the best commercial optimizers. This "unfortunate" situation is a blessing to query processing researchers however. There are always better algorithms and new techniques that will improve things for users. This issue presents papers from leading researchers who have focused on improving query response time. This is, as the issue makes clear, not the same as improving query throughput or completion time. The goal is to deliver data to users quickly, and this issue explores strategies for achieving that result. Mike Franklin has done a fine job as the editor responsible for the issue. Mike had this issue done in August, but it was delayed while the specifics of the TC election were being finalized.

Changing Editorial Staff

The Bulletin practice is to appoint editors for two years. Each editor is responsible for producing two issues, one per year, during that time. Among the current editors, Mike Franklin and Betty Salzberg have now completed their two issues and are due to "retire". While it may be trite, it is nonetheless true, that our profession depends on hardworking volunteers for the vitality of publications like the Bulletin. So I would like to thank both Betty and Mike for their hard work and very successful results.

As I was delighted with the work of Betty and Mike, I am now delighted to announce the appointment of two new editors, Surajit Chaudhuri and Donald Kossmann. Both Surajit and Donald are outstanding database researchers.

Surajit Chaudhuri is a colleague of mine in the Database Research Group at Microsoft Research. His expertise includes query processing, physical database design, data mining, and a very broad understanding of database technology in general. Surajit began his career as a student at Stanford and subsequently worked at HP Labs before coming to Microsoft.

Donald Kossmann is on the faculty at the University of Passau in Germany. He is a graduate of the University of Aachen and has had extended visits to the University of Maryland and IBM's Almaden Research Center. Donald's research focus is on the performance aspects of distributed database systems, including the interoperation of heterogeneous dbms's.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Query Processing has been an active area of research for several decades, resulting in a well-accepted set of standard techniques. As time goes on, however, the limitations of these standard techniques have become increasingly apparent. Improvements in hardware, communications, and database systems software have opened the door to increasingly challenging applications. Databases are becoming larger, data formats are more diverse, and data sources are distributed around the world. At the same time, user expectations for flexible and responsive access to these data have increased. With the growing importance of highly-interactive applications such as On Line Analytical Processing (OLAP) and browser-based access to the WWW, users have come to expect on-demand answers to their increasingly sophisticated queries.

When traditional query processing techniques are applied in these interactive settings, the result is too often a system that seems unresponsive. Answers take too long to be produced, if they are produced at all, leading to user frustration and reducing the usefulness of the system. For on-line applications, performance issues become usability issues; a responsive system can be used to solve problems in an interactive fashion while a slow system cannot.

The papers collected in this issue describe techniques that have been developed to improve the responsiveness of query processing for a number of different settings, including wide-area distributed systems, on-line processing, and tertiary storage. All of the approaches propose modifications to existing query processing engines and/or optimizers whose aim is to get useful data back to users more quickly. The techniques can be grouped into the following classes:

- Dynamically choosing and/or reordering query execution strategies based on observed state.
- Exploiting better knowledge of what answer the user really wants.
- Providing approximations and/or early feedback rather than waiting for a complete answer.

The first paper, by Amsaleg et al., describes two dynamic techniques for reacting to unexpected delays that are encountered when querying data from widely-distributed sources. One technique is aimed at relatively short, transient delays, while the other responds to longer-term availability problems. The second paper, by Carey and Kossmann, shows the tremendous improvements in response time that can be achieved by making SQL slightly more expressive and by exploiting the additional information during query processing. Hellerstein presents an overview of his work in on-line processing, which aims to make systems more usable by providing early, sometimes approximate answers and allowing the user to control the execution of a query based on how well these results match his or her needs. Our next two papers describe systems developed for heterogeneous, distributed data access. Liu and Pu present an approach to construction and late binding of distributed query execution plans and the dynamic assembly of query results in order to cope with a constantly changing environment. Ozcan et al. provide an overview of the MIND system developed at METU, focusing on a novel approach to constructing query execution plans dynamically during runtime. Finally, Sarawagi describes a query engine that dynamically reorders query execution based on data location and request arrival. The aim of this work is to avoid where possible, accessing data from expensive tertiary storage devices, but the techniques are applicable for solving other responsiveness problems as well.

I thank the authors for taking the time to prepare these interesting summaries of their ongoing work, and once again thank Dave Lomet for his efforts in producing the Data Engineering Bulletin.

Michael Franklin
University of Maryland

Improving Responsiveness for Wide-Area Data Access*

Laurent Amsaleg — INRIA
Philippe Bonnet — Bull
Michael J. Franklin — University of Maryland

Anthony Tomasic — INRIA
Tolga Urhan — University of Maryland

Abstract

In a wide-area environment, the time required to obtain data from remote sources can vary unpredictably due to network congestion, link failure or other problems. Traditional techniques for query optimization and query execution do not cope well with such unpredictability. The static nature of those techniques prevents them from adapting to remote access delays that arise at runtime. In this paper we describe two separate, but related techniques aimed at tackling this problem. The first technique, called Query Scrambling, hides relatively short, intermittent delays by dynamically adjusting query execution plans on-the-fly. The second technique addresses the longer-term unavailability of data sources by allowing the return of partial query answers when some of the data needed to fully answer a query are missing.

1 Introduction

The continued dramatic growth in global interconnectivity via the Internet has made around-the-clock, on-demand access to widely-distributed data a common expectation for many computer users. Advances in resource discovery, heterogeneous data management, and semi-structured data management are providing *semantic* tools to enable the access and correlation of data from diverse, widely-distributed sources. A limiting factor of such work however, is the difficulty of providing responsive data access to users, due to the highly varying response-time and availability characteristics of remote data sources in a wide-area environment. Data access over wide-area networks involves a large number of data sources, intermediate sites, and communications links, all of which are vulnerable to congestion and failures. Such problems can introduce significant and unpredictable *delays* in the access of information from remote sources.

Current distributed query processing approaches perform poorly in the wide-area environment. They permit unexpected delays to directly increase the query response time, allowing query execution to be blocked for an arbitrarily long time when needed data fail to arrive from remote sites. The problem with current approaches is that they generate query execution plans statically based on a set of assumptions about the costs of performing

Copyright 1997 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
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 Bellcore, and by an IBM Shared University Research award. Laurent Amsaleg did this work while he was at the University of Maryland and was supported in part by an INRIA Fellowship. Philippe Bonnet and Anthony Tomasic performed this work in the context of Dyade, an R&D joint venture between Bull and INRIA.

various operations and the costs of obtaining data. Static approaches do not cope well with large fluctuations in these costs at *run-time*. Unfortunately , the apparent randomness of such delays makes planning for them during query optimization impossible.

We have developed two different but complementary approaches to address the issue of unpredictable delays in the wide-area environment. In [AFTU96] we introduced the concept of *Query Scrambling*, which reacts to delays by modifying the query execution plan *on-the-fly* so that progress can be made on other parts of the plan. In other words, rather than simply stalling for delayed data to arrive, as would happen in a typical query processing scheme, Query Scrambling attempts to *hide* unexpected delays by performing other useful work.

Query Scrambling assumes that all the data required by a query will eventually be received so that a complete result can be returned to the user. As such, once Query Scrambling has performed all possible work, the system waits for the missing data and will return the full result only once this data has been received. If the delays are too long, a user may not find it tolerable to wait for a complete answer. An approach to coping with longer-term delays is described in [BT97]. This latter approach allows the system to time-out on a data source or sources and return a *partial answer*, which encapsulates data obtained from available sources along with a description of the work remaining to be done.

The approaches we describe differ significantly from previous dynamic query optimization techniques. One popular approach to dynamic query optimization has been to delay binding of certain execution choices until query execution time (e.g., [CG94, ACPS96, SAL$^+$96, LP97]). In these approaches the final query plan is produced immediately prior to execution and then remains fixed for the duration of the query. Such approaches, therefore, cannot adapt to unexpected problems that arise *during* the execution. Approaches that do change the query plan during execution, have been proposed in [TTC$^+$90, Ant93, ONK$^+$96], to account for inaccurate estimates of selectivities, cardinalities, or costs, etc. made during query optimization, and in [Sar95] to adjust to the location of data in a deep-store memory hierarchy. In contrast, our work is focused on adjusting to the time-varying performance of loosely-coupled data sources in a wide-area network, and as a result, we have developed quite different solutions.

The reminder of this paper is organized as follows. Section 2 describes the query execution model assumed in this work, and discusses the inadequacy of current query processing techniques in more detail. Section 3 presents an overview of Query Scrambling. Section 4 presents the Partial Answer technique. Finally, Section 5 presents conclusions and future work.

2 Considerations for Wide-Area Query Processing

2.1 The Query Execution Model

We assume a query execution environment consisting of query source sites and a number of remote data sources. The processing work for a given query is split between the query source and the remote sites.[1] Query plans are produced by a query optimizer, based on its cost model, statistics, and objective functions. This arrangement is typical of mediated database systems that integrate data from distributed, heterogeneous sources.

An example query execution plan for such an environment is shown in Figure 1. The query involves five different base relations stored at four different sites. In the example, relations A and B reside at separate remote sites (sites 2 and 3 respectively), relation C resides locally at the query source (i.e., site 1), and relations D and E are co-located at site 4. In the plan shown in Figure 1, site 1 joins selected data received from sites 2 and 3, and joins C with the result of (D⋈E) that has been computed remotely at site 4. Site 1 also computes the final result delivered to the user.

[1] Note that as currently specified, both the Query Scrambling and Partial Answer approaches treat remote sources as black boxes, regardless of whether they provide raw data or the answers to sub-queries (e.g., [TRV96]). Therefore, both approaches operate solely at the query source site.

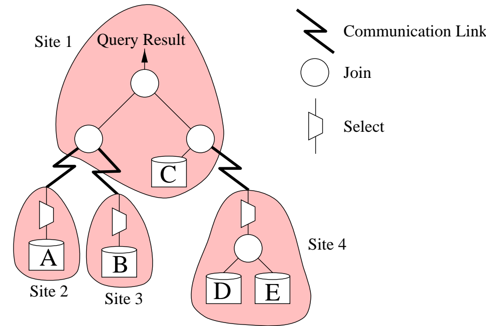

Figure 1: Example of a Complex Query Tree

The actual schedule of the operators that comprise the query shown in Figure 1 (joins, selects, etc.) depends on the execution model used (e.g., iterators, activation records) and on whether or not parallelism is supported. For simplicity, assume an iterator-based scheduler [Gra93], where the first data access would be to request a tuple of Relation A (from site 2). If there is a delay in accessing that site (say, because the site is temporarily down), then the scan of A is blocked until the site recovers. Using a static, iterator-based scheduler, the entire query execution would remain blocked until site 2 responded (i.e., pending the recovery of the remote site).

2.2 Inadequacy of Static Scheduling

The previous example demonstrated of the kind of problems that can arise due to unexpected delays when a traditional, iterator-based scheduling approach is used. A key point however, is that similar problems can arise with *any* static scheduling approach. In general, the producer-consumer relationships that exist between the operators of a query determine the impact of delays on query execution. When an operator that is accessing remote data from a particular site encounters a delay, that operator *stalls*. The operator itself stops producing data and thus, stalling propagates up the query plan through the producer-consumer pairs. Furthermore, in some cases the problem can even propagate to other parts of the plan. For example, when certain binary operations (e.g., merge joins) become blocked because of a delay experienced by one of their children, they stop consuming the tuples produced by their *other* child, which can eventually cause that child to stop producing tuples.

Parallelism can, to a limited extent, ameliorate some of the problems caused by remote access delays. In general, parallelism allows concurrent execution of operations at the expense of a more complex management of resources. A query optimizer can consider three types of parallelism: *intra-operator*, *pipeline*, and *bushy tree*. The optimizer chooses the appropriate type and degree of parallelism to exploit, but not overburden the resources of the system. For example, parallelism is limited by (among other things) the amount of memory expected to be available when the plan executes.

Like most other optimization decisions, parallelism is typically planned for in a *static* fashion. As a result, delayed data will impact parallel plans as well as sequential ones. For example, using pipelined parallelism, a blocked operator may eventually block the entire pipe; for intra-operator parallelism, the siblings of a blocked thread of a parallel operator may continue processing, but the operator itself will not be able to complete; for inter-operator parallelism, blocked operators will continue to consume resources (e.g., memory) that will limit the amount of other work the system can perform.

In summary, static scheduling limits the extent to which the DBMS can cope with delays that arise at runtime, regardless of whether or not parallelism is pre-compiled into a plan. Adapting to unexpected delays requires more flexible approaches, such as those we describe in the following sections.

3 Query Scrambling

Using Query Scrambling [AFTU96, AFT97], a query is initially executed according to the original plan and associated schedule generated by the query optimizer. If, however, a significant performance problem arises during the execution, then Scrambling is invoked to modify the execution *on-the-fly*, so that progress can be made on other parts of the plan. In other words, rather than simply stalling for delayed data, Query Scrambling attempts to *hide* unexpected delays by performing other useful work.

Query Scrambling reacts to delays in receiving data from remote data sources in two ways:

- *Rescheduling* - the execution plan of a query can be dynamically rescheduled when a delay is detected. That is, operators that already exist in the plan can be scheduled in response to delays detected with other operators. In this case, the basic shape of the query plan remains unchanged.
- *Operator Synthesis* - new operators (e.g., a join between two relations that were not directly joined in the original plan) can be created when there are no other operators that can execute. In this case, the shape of the query plan can be significantly modified through the addition, removal and/or re-arrangement of query operators.

Query Scrambling works by repeatedly (if necessary) applying these two techniques to a query plan. For example, assume that the query shown in Figure 1 stalls while retrieving tuples of A. Instead of waiting for the remote site to recover, Query Scrambling could perform rescheduling, and retrieve the tuples of B *while* A is unavailable. Note that these tuples would need to be stored temporarily at the query site. If, after obtaining B, A is still unavailable, then rescheduling could be invoked again, for example, to trigger the execution of (D⋈E) at site 4, and to join this result with C. If at this point, A is still unavailable, then Operator Synthesis can be invoked to create a new join between B and (D⋈E)⋈C. Note that in general, Operator Synthesis may result in new operators, which may later be run by a subsequent Rescheduling. Also note that operators initiated by Query Scrambling may as well experience delays, which may cause Scrambling to be invoked further.

3.1 Implementing Scrambling

To implement Query Scrambling, a *scrambling coordinator* must be added to the query execution engine. This coordinator passively supervises the execution of the query and watches for remote access delays. When a delay is detected, the coordinator reacts by performing Scrambling actions that it deems to be advantageous based on the current state of the system. In [AFTU96] we described a simple, heuristic-based algorithm for choosing scrambling actions. In our current work, we have developed *cost-based* scrambling policies, that use a query optimizer to help direct Scrambling.

The features that must be incorporated into a query processing system in order to support Query Scrambling include the following:

Detecting Delayed Sources. The coordinator has to detect when delays arise during remote data access. A *timer* associated with each operator that directly accesses data from a remote site can be used for this purpose.

Detecting Resumed Sources. The coordinator also has to detect the arrival of data from a remote source that was previously delayed.

Patching the Query Tree. To implement Scrambling Rescheduling, the coordinator needs to introduce a *materialization operator* between the rescheduled operator and its original parent. A materialization is a unary operator, which when opened, obtains the entire input from its child and places it in storage (typically disk). This operator enables a producer to run *independently* of its original consumer, at the expense of using local resources.

Thread Management. Scrambling also requires thread management in order to allow the scheduling of existing and newly synthesized operators to be changed dynamically.

Scrambling-enabled Optimizer. A cost-based optimizer is needed in order to make intelligent scrambling decisions. While a traditional optimizer can be used, a better solution is to develop a *lightweight* optimizer that would take an existing plan and quickly generate an appropriate modification to that plan in response to recently discovered delayed or resumed sources.

3.2 Making Intelligent Scrambling Decisions

Given the above features, it is possible to develop of a number of different Scrambling *policies*. Each policy can be designed to provide the best reaction to a given delay scenario and may differ, for example, by the degree of parallelism introduced into the execution of the query or the aggressiveness with which scrambling changes the existing query plan. In general, two basic questions must be addressed by the scrambling policy: 1) which and how many operators should be rescheduled and/or synthesized when a delay is detected, and 2) what should happen to the rescheduled operators when delayed data eventually arrives. The answers to these questions depend on a number of tradeoffs, which differ for the Rescheduling and Operator Synthesis phases of Query Scrambling.

3.2.1 Trade-offs for Rescheduling

One important question that arises during Rescheduling is that of how many operators to reschedule concurrently. The tradeoff here is one between the benefits of overlapping multiple delays and the cost of the materializations used to achieve this overlapping. Initiating more operators allows more remote sources to be accessed in parallel, and hence, a greater potential for overlapping the delays encountered from those remote sources. Not surprisingly, however, more materializations incur more overhead due to I/O at the query source. As a result, materializing remote data is beneficial only when the amount of hidden delay is greater than the cost of the additional I/Os. Materializations also have the potential to randomize disk I/O (particularly if multiple relations are materialized in parallel), which can reduce the efficiency of algorithms that could otherwise exploit (faster) sequential I/Os. Two factors are relevant here: the speed of the network compared to the speed of the local disk(s) and the way data is obtained through the network (i.e., page-at-a-time versus streaming) [AFT97].

A second set of tradeoffs revolve around the question of whether to Reschedule individual operators (in a bottom-up fashion) or entire subtrees. Subtrees can be rescheduled (i.e., executed out of order) given sufficient available memory. Such rescheduling enables the entire subtree to be executed at the cost of only a single extra materialization, thereby taking advantage of any pipelining that is possible within the subtree. This is especially interesting if the data produced by the subtree is much smaller than the amount of base data used at the leaves of the subtree. When memory is limited however, the scheduling of whole subtrees becomes difficult to manage. For example, finding enough memory to allow the rescheduling of a subtree might require swapping out the memory frames occupied by stalled operators. Thrashing could then arise due to operators that repeatedly stall and unstall. In contrast, materializing base relations to the local disk requires only minimal memory, but of course, requires additional I/O.

Finally, choosing which specific operator(s) to reschedule is also fundamental. In [AFTU96] the operator to reschedule was elected depending on the original order of the operators in the query before any rescheduling. As illustrated in [AFTU96], this simplistic policy has several severe performance drawbacks. As with Operator Synthesis, we have developed cost-based approaches to this problem that avoid the pitfalls of the simpler heuristic policies. The decision in this regard is based on the ratio of useful work performed by the operator (i.e., how much closer it gets us to the final answer) to the amount of extra work rescheduling it will cause.

3.2.2 Trade-offs for Operator Synthesis

Creating new operations also raises a number of interesting trade-offs. Because the operations that may be created were not originally chosen by the optimizer they may entail a significant amount of additional work. If the synthesized operations are too expensive Query Scrambling could result in a net degradation in performance. Operation Synthesis, therefore, has the potential to negate or even reverse the benefits of Scrambling if care is not taken. In [AFTU96] we used the simple heuristic of avoiding Cartesian products to prevent the creation of overly expensive joins. This approach has the advantage of avoiding the need to do cost-based optimization, but its performance was shown to be highly sensitive to the cardinalities of the new operators created.

More recently, we have developed and studied several different ways of to use a query optimizer in Operator Synthesis process. One policy, called *Pair*, is similar to the original heuristic in that it considers creating only one new operator at a time, but differs in that it uses the cost-model of the optimizer to choose which operator is likely to be most beneficial. A second policy, called *Exclude Delay*, uses the cost-based optimizer to generate execution plans for each of the connected components of the query join graph that remain when the delayed data source is removed. A third policy, called *Include Delay*, uses a query optimizer that is targeted to minimize response time rather than cost. In this approach, the optimizer is told that delayed relations will not arrive until far in the future. Because the optimizer tries to optimize response time, it tends to generate plans where the delayed relations are used as late as possible. This has the effect of causing other useful work to be performed first. Our fourth policy, called *Estimated Delay* works similarly to *Include Delay* but rather than assuming that the delay is effectively infinite, it initially generates plans assuming that delays will be relatively short, and successively increases its estimate if the delayed relations fail to arrive during the previous estimated delay period. This latter approach has the effect of making increasingly aggressive Scrambling decisions as delays increase. Our preliminary results indicate that while none of these policies is perfect, *Estimated Delay* provides the most robust performance over a range of query plans and delay scenarios.

3.2.3 When to Stop Scrambling

A remaining question is when to stop scrambled operations once they have been initiated. One approach is to suspend all scrambling operations as soon as a blocked operator becomes unblocked, and to resume normal processing. Since Scrambling is a reaction to an unanticipated event, it intuitively makes sense to resume the original plan as soon as possible. This is because Scrambling has the potential to add costs to the query execution and stopping it can help avoid such costs. Returning to the original schedule, however, raises other questions. First, there are costs associated with rescheduling operators, and as stated above, thrashing could be induced in the presence of repeated, intermittent delays. Secondly, since Scrambling performed some work while a delay was experienced, there is the question of what to do with the results of that work. Using the results in further computations may or may not be beneficial. For example, reading a materialized relation may be more costly than requesting it again through the network if the network behaves well. In contrast, using intermediate results computed by Scrambling might save time. The investigation of such trade-offs is one aspect of our ongoing research.

4 Partial Answers for Unavailable Data Sources

4.1 Overview

Because Scrambling uses the tactic of performing other useful parts of a query when a delay is detected, the amount of delay that it can successfully hide is limited by the amount of work that is contained in the original query. In a wide-area environment, however, data might be delayed for a very long period of time. For an interactive system, if a delay lasts longer than a user is willing to wait, then in effect, the delayed data is *unavailable*.

Because Query Scrambling returns only complete answers to users, it does not solve the problem of unavailable data.

[BT97] proposes an approach where in the presence of unavailable data, a *partial answer* is returned to the user. The motivation behind this approach is that even when one or more needed sites are unavailable, some useful work can be done with the data from the sites that are available. A partial answer is a representation of this work, and of the work that remains to be done in order to obtain the complete answer.

The uses of partial answers are twofold. First, a partial answer contains a query that can be submitted to the system in order to later obtain the complete answer efficiently. Second, a partial answer contains data from the available sites that can be extracted using secondary queries that we call *parachute queries*. Associated with each original query is a set of parachute queries that can be asked if the complete answer cannot be produced. The answer to a parachute query is a set of tuples; it is not necessarily a subset or a superset of the complete answer.

4.2 Framework for Partial Answers

We now detail the framework we have defined for partial answers. We assume that sites have atomic behavior: they are either available or unavailable. If a site starts producing an answer to a sub-query, we assume it is available and it produces its result completely. If the delay in the arrival of the first tuple is above a given limit, then we assume that the site is unavailable and produces no tuples. We plan to relax this atomicity assumption in our future work.

To understand our approach, consider a query that involves several sites, such as the query of Figure 1: *select * from* $(A \bowtie B) \bowtie (C \bowtie (D \bowtie E))$. If all sites are available, then the system returns a complete answer. Suppose, however, that site B is unavailable. In this case a complete answer to this query cannot be produced. The system proposed in [BT97] would perform the following steps:

- *Phase 1* - each available site is contacted, and all processing based on available data is performed. The results that are obtained are materialized on the query source site. In our example, tuples of A will be selected; the subquery will be evaluated by site 4, the result returned and joined with C. These results are materialized in temporary relations on site 1.

- *Phase 2* - queries representing the data obtained in Phase 1 are constructed. In our example:

  Q1 = *select * from A where* c_1

  Q2 = *select * from* $C \bowtie (D \bowtie E)$ *where* c_2

  These queries can be evaluated without contacting the remote sources. They denote data materialized locally.

- *Phase 3* - a query semantically equivalent to the original query is constructed using the queries constructed in Phase 2 and the relations from the unavailable sites. In our example:

  Qor = *select * from* $(Q1 \bowtie B) \bowtie Q2$ *where* c_3

When this processing is finished, a partial answer is returned to the user. The partial answer is a handle for the data obtained and materialized in Phase 1, as well as for the queries constructed in Phase 2 and Phase 3. One way to use a partial answer is to submit the query Qor in order to obtain the *complete* answer. Evaluating Qor requires contacting only the remote sites that were unavailable when the original query was evaluated. In the example, Q1 and Q2 denote local data, only B is located on a remote site. When Qor is submitted to the system, the query processor considers it in the same way as a regular query, and it is optimized accordingly. If Qor is evaluated when the remote sites are available, then a complete answer is returned. Under the assumption that no updates are performed on the remote sites, this answer is the answer to the original query. If some of the sources

that were unavailable during the previous evaluations remain unavailable, then another partial answer is returned. Possibly, successive partial answers are produced before the complete answer can be obtained.

Submitting Qor instead of resubmitting the original query has two advantages: First, a complete answer can be produced even if all the sites are never simultaneously available. It suffices that a site is available during the evaluation of one of the successive partial answers to ensure that the data from this site is used for the complete answer. Second, as Qor involves temporary relations that are materialized locally, evaluating Qor is typically more efficient than evaluating the original query.

An alternative way to use a partial answer is to extract data from it using parachute queries. Parachute queries are associated with the original query; they may be asked in case the complete answer to the original query cannot be produced. Consider a system where the relations *patient* and *surgeon* are on different sites and the query: *"list the names of all surgeons who have operated on patient X"*, In this scenario, some example parachute queries are: *"list the identifiers of all the surgeries patient X has undergone"*, or *"list the names of all surgeons"*. Using parachute queries, the user can still collect useful information concerning patients or surgeons in case the complete answer to the original query cannot be computed.

[BT97] proposes an initial algorithm for the extraction of information using parachute queries. When a parachute query is submitted, it is tested for containment against the queries generated in Phase 2. If the parachute query is contained in one of these queries then the system returns the complete answer to the parachute query, otherwise it returns null. The set of parachute queries that can be evaluated is limited in this scheme, however, because Phase 1 operates without knowledge of the parachute queries that may be asked. To overcome this limitation, we envisage a system where the parachute queries could be asked together with the original query. In such a system, parachute queries are no longer used solely to extract materialized information, rather, they are alternative queries that the system tries to evaluate if the complete answer to the original query cannot be produced. Defining such a system is part of our ongoing work.

5 Conclusions

Accessing distributed data across wide-area networks poses significant new problems for database query processing. In a wide-area environment, the time required to obtain data from remote sources can vary unpredictably due to network congestion, link failure or other problems. Traditional techniques for query optimization and query execution do not cope well with such unpredictability. In this paper we presented two different but complementary techniques to address the problem of unpredictable delays in remote data access. The first technique, called Query Scrambling, hides relatively short, intermittent delays by dynamically adjusting query execution plans on-the-fly. The second technique addresses the longer-term unavailability of data sources by allowing the return of partial query answers when some of the data needed to fully answer a query are missing.

This paper represents a current snapshot of our explorations into the development of flexible systems that dynamically adapt to the changing properties of the run-time environment. In our ongoing work, we are continuing to develop these ideas by improving our cost-based decision making, exploring the tradeoffs we have outlined, and examining a wider array of delay scenarios. Furthermore, the approaches presented in this paper may be useful for other types of problems that have been targeted by previous approaches to dynamic query optimization, such as optimizer estimation errors and the lack of cost information for remote data sources in heterogeneous environments. In the longer term, we plan to look at other approaches to reactive query processing, such as choosing among similar data sources that possibly vary in completeness, consistency, or "quality" in order to find useful trade-offs between responsiveness and accuracy. As distributed systems continue to grow in size, complexity, and general unmanagability, such adaptive techniques will continue to become increasingly important for providing users with responsive access to the data they need.

References

[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.

[AFT97] L. Amsaleg, M. Franklin, and A. Tomasic. Dynamic query operator scheduling for wide-area remote access. Tech. Report CS-TR-3811 and UMIACS-TR-97-54, Univ. of MD, College Park, July 1997.

[AFTU96] L. Amsaleg, M. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *Proc. of the Int. Conf. on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, Florida, December 1996.

[Ant93] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proc. of the Data Engineering Int. Conf.*, pages 538–547, Vienna, Austria, 1993.

[BT97] P. Bonnet and A. Tomasic. Partial answers for unavailable data sources. Technical Report RR-3127, INRIA, Rocquencourt, France, March 1997.

[CG94] R. Cole and G. Graefe. Optimization of dynamic query execution plans. In *Proc. of the ACM SIGMOD Int. Conf.*, pages 150–160, Minneapolis, Minnesota, May 1994.

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

[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.

[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.

[SAL+96] M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *The VLDB Journal*, 5(1):48–63, January 1996.

[Sar95] S. Sarawagi. Query processing and caching in tertiary memory. In *Proc. of the 21st Conference on Very Large Databases*, Zurich, 1995.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *The IEEE Int. Conf. on Distributed Computing Systems (ICDCS-16)*, Hong Kong, 1996.

[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.

Processing Top N and Bottom N Queries

Michael J. Carey
IBM Almaden Research Center
San Jose, CA 95120

Donald Kossmann
University of Passau
94030 Passau, Germany

1 Introduction

In certain application areas, such as those related to decision support or multimedia data, users wish to ask so-called *top N* and *bottom N* queries; these are queries that request a certain number of answers (N) having the highest or lowest values for some attribute, expression, or function. For example, rather than finding all publications on a certain topic, a researcher may want to retrieve the ten most heavily referenced papers on the topic at hand. A politician planning his or her next campaign might be interested in discovering the average salary of the wealthiest ten percent of the voters in a given district. Parents of a young child might want to find five mystery books that least well match the terms "crime" and "murder." These examples illustrate a variety of situations in which *top N* and *bottom N* queries are meaningful. In addition, they demonstrate the fact that such queries can involve standard relational data as well as text or other multimedia data.

To date, the SQL standard does not include statements that allow users to pose such *top N* and *bottom N* queries. There have, however, been several proposals in the literature (e.g., [KS95, CG96, CK97]), and database system vendors are beginning to extend their SQL dialects and query interfaces in order to support such queries. Given the obvious need and this growing interest, this paper addresses the question of how *top N* and *bottom N* queries can be processed efficiently; moreover, we address the question of how such support can be provided as a natural extension of existing relational query processing architectures. In a nutshell, our goal is to evaluate such queries with as little *wasted work* as possible. That is, if a query asks for the 10 most popular publications, we want to avoid work to process, say, the 11th, 12th, or 13th most popular publications.

We will begin by presenting a series of situations in which a traditional DBMS, i.e., one without integrated support for *top N* and *bottom N* queries, would end up wasting work. We then show how such a traditional DBMS could be extended – with relatively little effort, in fact – in order to avoid such wasted work and thereby achieve orders-of-magnitude improvements in many cases. Our goal here is to drive home the point that database systems must be extended in order to process *top N* and *bottom N* queries efficiently and to briefly touch upon each of the required extensions; a detailed description of our approach, as well as a performance evaluation, can be found in [CK97]. We will focus here on SQL and relational databases, using relational queries as examples and citing some measurements from a relational DBMS platform to illustrate the important performance gains that can be achieved; we note, however, that most of the effects and techniques discussed in this paper are applicable to any kind of database system.

Copyright 1997 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Forms of Wasted Work

In this section, we discuss five example queries that illustrate ways in which a traditional DBMS, which does not have built-in support for *top N* and *bottom N* queries, will end up wasting work when faced with applications that require this type of query support. These queries illustrate different forms of wasted work that occur in a traditional DBMS. We also explain how an enhanced system would be able to avoid the wasted work in each case. In addition, we cite measurements that we obtained by "simulating" our proposed system enhancements using IBM's DB2 for Common Servers DBMS product; the proposed enhancements were simulated by adding predicates to the test queries such that DB2 chose the same or equivalent plans that would be used by an enhanced system. The measurements clearly show that orders-of-magnitude performance improvements can be achieved by enhancing a DBMS with explicit support for *top N* and *bottom N* queries.[1]

To specify the example queries, we will use the syntax that we proposed in [CK97]; that is, the queries will be standard SQL queries with an additional (non-standard) `STOP AFTER` clause to specify the number of requested answers (i.e., N). In the following, we will first define the test database and then discuss the five examples.

2.1 The Test Database

To demonstrate the different forms of wasted work, we will use a simple database that contains information about a company's employees, departments, and employees' travel expense accounts (TEAs). Specifically, the database will consist of the following three tables:

Emp(<u>empId</u>, name, salary, *works_in*, *teaNo*)
Dept(<u>dno</u>, name, budget, function, description)
TEA(<u>accountNo</u>, expenses, comments)

Here, the underlined columns represent the primary keys of the tables, and the italicized columns represent foreign keys. In addition to key and referential integrity constraints, we assume that the database has constraints to enforce the fact that every employee works in some department and that every employee has a travel expense account; i.e., we assume that there are `NOT NULL` constraints in addition to foreign key constraints on the columns `Emp.works_in` and `Emp.teaNo`.

For our performance experiments, we generated synthetic `Emp`, `Dept`, and `TEA` tuples. For the measurements cited here, the `Emp`, `Dept`, and `TEA` tables had 10,000 tuples of 100 bytes each; i.e., all three of these tables contained 10 MB of data. Furthermore, we generated (clustered) primary indexes on `Emp.empId`, `Dept.dno`, and `TEA.accountNo`, and unless the text says otherwise, there was a secondary (and unclustered) index available on `Emp.salary` in each experiment. All of the indexes are B^+ trees, so they can be used in order to evaluate range and join predicates as well as reading data out of the corresponding tables in indexed-column order.

2.2 Example 1: Shipping Query Results

Our first example demonstrates the simplest form of wasted work. This form arises when the run-time system of the DBMS does not know how many tuples are desired by the application; in the absence of SQL support for *top N* or *bottom N* queries, the desired result-limiting effect will end up being implemented, e.g., simply by closing a cursor early at the application level. Specifically, this example addresses the impact of a feature called *row blocking*, which is implemented in most commercial database systems in order to reduce the overhead of shipping query results from the DBMS back to an application process. Using row blocking, the DBMS allocates a certain number of buffers for a given query. It fills these buffers with tuples of the query result and then ships the whole *block* of result tuples to the application process when the buffers become full. The application process reads the query results in a tuple-at-a-time fashion using, e.g., the SQL cursor interface. Once all the tuples of

[1] Details of the experiments that produced the results that we cite here can be found in [CK97].

the first block have been consumed by the application, the DBMS produces more query results and ships the next block of result tuples to the application process.

Row blocking is a simple yet important mechanism for efficient client-server query processing. The alternative, returning query results one tuple at a time, can become prohibitively expensive when a query result is large. For *top N* and *bottom N* queries, however, row blocking can be the cause of wasted work if the DBMS does not know the value of N and must therefore assume that the application will consume all of the tuples that satisfy a given query predicate. We will demonstrate this effect with the following example query, which asks for the first one hundred tuples of the `Emp` table (i.e., for the 100 `Emps` with the smallest *id* values):

```
SELECT     *
FROM       Emp e
ORDER BY   e.id
STOP AFTER 100
```

If the DBMS's result buffers have room for 500 tuples, a traditional DBMS would respond to this query by producing the first 500 `Emps` (using the primary index on `Emp.id`) and shipping them to the application process. In contrast, an enhanced DBMS would know that the application only wants 100 tuples, and it would therefore only produce 100 `Emp` tuples and send them as a block to the application process. In our experiments, the enhanced approach outperformed the traditional approach by a factor of four in this case:

Traditional	Enhanced
0.285s	0.075s

Two forms of wasted work are causing this performance difference – the traditional DBMS fetches too many tuples out of the `Emp` table, and it ships too much data back to the application process, both of which are unnecessary costs from the application's perspective. Obviously, the advantages of the enhanced approach become even more pronounced in this example when fewer tuples are requested by the application; the same is true of the other examples presented in this section as well.

2.3 Example 2: Access Path Selection

The second example demonstrates a basic reason why the query optimizer of a database system needs to know the value of N when optimizing a *top N* or *bottom N* query. The query in this example asks for the 100 best paid `Emps`:

```
SELECT     *
FROM       Emp e
ORDER BY   e.salary DESC
STOP AFTER 100
```

Assuming that there is a non-clustered index on `Emp.salary`, the best way to execute this query would be to use this index to find the 100 best paid `Emps` and then fetch them from the `Emp` table. Unfortunately, without the knowledge that only 100 `Emps` are being requested (i.e., assuming that all `Emps` must be returned in `salary` order), a traditional query optimizer would choose a different plan: it would choose a plan with a table scan followed by a sort operator, as the use of an unclustered index would have an extremely high estimated cost for disk seeks if the whole `Emp` table with 10,000 tuples is assumed to be read. Our experiments showed that, for this example, a stop-enhanced query optimizer would outperform a traditional optimizer by over two orders of magnitude:

Traditional	Enhanced
15.633s	0.076s

2.4 Example 3: Cost of Sorting

The third example shows that an enhanced database system requires new, specialized query operators in order to process certain kinds of *top N* and *bottom N* queries efficiently. Let us reconsider the query from the previous subsection, which asked for the 100 best paid `Emps`. Let us now assume that there is no index available on `Emp.salary`. A traditional system would again execute this query with a conventional sort operator. However, a better approach to find and sort only the 100 best paid `Emps` is to build a priority heap in main memory [Knu73]. The first 100 `Emps` would be inserted into that priority heap and, after that, every `Emp` tuple would be inspected to see if it has higher salary than the *bottom* `Emp` in the heap. If so, the tuple would be inserted into the heap, thereby replacing the current bottom `Emp` of the heap; if not, the tuple would simply be discarded because it cannot possibly be part of the requested query result. Our experiments showed that this stop-enhanced approach to executing this example query outperforms the traditional approach by a factor of three because it avoids the wasted work of sorting (many) tuples which are not part of the desired query result:

Traditional	Enhanced
15.633s	5.775s

2.5 Example 4: Pipelined Join Methods

The fourth example again demonstrates a potential improvement that can be achieved by enhancing the query optimizer of a traditional DBMS. This example shows that many *top N* and *bottom N* queries are best executed using a pipelined query plan; specifically, this example shows that a nested-loop index join can be a very efficient method to evaluate a multi-table *top N* and *bottom N* query. The following example query requests the employee and department information for the 100 best paid `Emps`:

```
SELECT      *
FROM        Emp e, Dept d
WHERE       e.works_in = d.id
ORDER BY    e.salary DESC
STOP AFTER  100
```

If both tables are large, the optimizer of a traditional system, which is oblivious to the desired final result cardinality, would likely choose a sort-merge or hybrid-hash join in order to evaluate this query. In contrast, an enhanced system would most likely generate a plan in which the `Emp` tuples are produced in order (using the `Emp.salary` index if there is one, or a sort-based operator if not) and then join one `Emp` tuple with the `Dept` table at a time using the index nested-loops method until 100 `Emp` tuples have qualified. Our experiments show that such an enhanced system would outperform a traditional system by two orders of magnitude in this case if there is an index available on `Emp.salary`:

Traditional	Enhanced
80.716s	0.195s

2.6 Example 5: Join Queries With No Good Pipelined Plan

For some *top N* and *bottom N* queries, no good pipelined query plan exists. Examples include queries with large N, where an index nested-loop join becomes too expensive for producing all N results. Another example is a multi-way join query where not all joins should be carried out in a pipelined fashion, e.g., because of excessive main-memory consumption, or because the query optimizer chooses a join order with a bushy query plan. In such situations, it is important to reduce the cardinality of intermediate results as early as possible in order to reduce the cost of subsequent expensive (e.g., join) operators. This is illustrated by the following example, which requests the department and travel expense information for the 100 highest paid employees:

```
SELECT      *
FROM        Emp e, Dept d, TEA t
WHERE       e.works_in = d.id AND t.teaNo = t.accountNo
ORDER BY    e.salary DESC
STOP AFTER  100
```

Given the facts (implied by the database's integrity constraints) that every `Emp` works in a department and has a travel account, an efficient way to execute this query is to identify the 100 best paid `Emps` first and then carry out the joins in order to find out the department and travel information for only these 100 tuples. A traditional system, however, would most probably carry out the joins first, using the whole `Emp` table, identifying the 100 best paid `Emps` with their `Dept` and `TEA` information only at the end (i.e., up in the application program). As a result, our experiments showed that a traditional system would be outperformed by an order of magnitude by a stop-enhanced system in this example:

Traditional	Enhanced
128.307s	6.381s

3 Extending a DBMS

In this section, we will describe how a traditional relational DBMS can be extended to provide good performance for all five of the examples from the previous section. Our approach is to use existing components as much as possible to process *top N* and *bottom N* queries, only adding new stop-specific code when absolutely necessary. As a result, the required changes are moderate, making it possible to implement and integrate the changes with a modest amount of effort; the handling of regular queries (i.e., non-top/bottom queries) is unchanged. After describing our approach, we will briefly discuss how some degree of support for *top N* and *bottom N* queries has recently been integrated into certain commercial database systems.

3.1 Our Approach

In the following, we list and briefly sketch out the changes that we propose in order to extend a traditional (relational) DBMS to handle *top N* and *bottom N* queries. A much more detailed description of these extensions and their impact can be found in [CK97].

SQL and Parser As mentioned earlier, we propose extending the syntax of SQL to include a `STOP AFTER N` clause as an optional suffix to SQL's `SELECT` statement; this allows users to express *top N* and *bottom N* queries. Just as any `SELECT` statement can be used as part of the definition of a subquery or a view, we allow a `STOP AFTER` clause (and a corresponding `ORDER BY` clause) to be used in the definition of subqueries and/or views. Furthermore, we will allow for N, the number of answers requested by the application, to be provided as an expression (including support for complex expressions such as a scalar subquery). The addition of the `STOP AFTER` clause is sufficient to give the DBMS the cardinality information needed to avoid wasted work due to row blocking (see the first example of Section 2), as the DBMS then has a strict upper limit on the number of result tuples that can result from the query.

Stop Operators The second extension we propose is the addition of a *Stop* operator to the repertoire of the DBMS's query execution engine. The Stop operator produces the top (or bottom) N tuples of its input stream, taking the stopping cardinality N plus a sort expression as parameters. The Stop operator serves to encapsulate the function of the `STOP AFTER` clause; the query execution plan for a *top N* or *bottom N* query will contain at least one Stop operator. In addition, given a Stop operator, the implementation of the other (existing) operators, such as join operators, sort, group-by, and scans, does not need to be changed at all.

Like many query operators, such as join, sort, and group-by, the Stop operator can be implemented in any of several different ways. If the input of the Stop operator is already ordered according to its sorting expression, then the Stop operator will simply return the first N tuples of its input and then signals "end-of-stream." If the input is not ordered, and N is relatively small, the Stop operator can be implemented using a priority heap as described in Section 2.4. Finally, if the input is not ordered, and N is very large, the Stop operator can be implemented using a conventional external sort algorithm that discards the remaining data after identifying the first N results.

New Optimization Rules and Modified Pruning for Stop Operator Placement To correctly enumerate plans with Stop operators, the query optimizer must be extended. This extension is quite easy to implement in modern optimizers, as modern optimizers are rule-based; the enumeration of plans with Stop operators can be arranged by adding new rules related to Stop operators to the optimizer [GD87, Loh88]. Note that it is important to enumerate all possible plans with Stop operators during cost-based query optimization because Stop operators influence the cost of other operators and, as a result, are likely to impact other cost-based decisions carried out by the optimizer such as join ordering, choice of join methods, and access path selection.

In addition to defining new rules to enumerate alternative plans with Stop operators, the pruning condition of a query optimizer that is based on dynamic programming needs to be changed. It is, for example, possible for a subplan with a Stop operator and a higher cost than a corresponding subplan without a Stop operator to end up being a building block for the best overall plan for the whole query. This situation can arise because Stop operators have a non-negligible cost, and the quality of a subplan with a Stop operator can only be seen for sure upon considering operators that come later in the query plan; this is because such operators benefit from the fact that the Stop operator yields a smaller intermediate query result. The optimizer therefore cannot safely prune a plan with one or more Stop operators in favor of another (otherwise equivalent) query plan without a Stop operator.

Stop Operator Placement New rules and a modified pruning condition will ensure that all possible plans with Stop operators are enumerated and that no query plans with Stop operators are pruned prematurely. Of course, we also need to decide *where* Stop operators are to be placed in a query plan. As shown in the fifth example of Section 2, Stop operators should be located as early as possible in a query plan so that the cost of subsequent operators will be reduced. On the other hand, Stop operators should ideally only be placed at *safe* points, i.e., at points in a query plan where their presence can never cause tuples to be discarded that may end up being needed in order to generate the requested N tuples of the query result. Thus, the goal is to find the earliest *safe* place for a Stop operator in a query plan.

Safe places for Stop operators in a query plan can be found by inspecting the database's integrity constraints together with the given query's predicates (i.e., the query's `WHERE` clause). For the `Emp-Dept-TEA` query of Section 2.6, for example, placing a Stop operator below the joins with `Emp-Dept` and `Emp-TEA` joins was safe because the referential and `NOT NULL` integrity constraints on `Emp.works_in` and `Emp.teaNo` guarantee that every `Emp` tuple will satisfy both join predicates; consequently, they ensure that (at least) 100 result tuples will be produced from joins involving the 100 best paid `Emps`. Now consider a different example, where the user asks for the `Dept` and `TEA` information for the 100 best paid `Emps` that work in a "research" department. In this case, a Stop operator could safely be placed below the `Emp-TEA` join, as before. However, it could not safely be placed below the `Emp-Dept` join unless there is an integrity constraint that guarantees that every department is a research department (because, in general, not all `Emp` tuples will survive a join with only the research `Dept` tuples).

In [CK97], we also studied *unsafe* (a.k.a. *aggressive*) Stop operator placement techniques; these techniques can be applied even if there are no appropriate integrity constraints. They would, for instance, allow the placement of a Stop operator below the `Emp-Dept` join in the "find the 100 best paid `Emps` that work in a research department" example discussed above. In this case, the optimizer will estimate the number of tuples that should

be filtered out by the (unsafe) Stop operator by considering cardinality and selectivity estimates and working backwards from the desired overall result cardinality. For example, if every other `Emp` works in a research department, the unsafe Stop operator will be instructed to stop after returning the 200 best paid `Emps`; these 200 tuples would be expected to include the requested 100 result tuples. Of course, cardinality estimates can easily be too high or too low, so that precautions must be taken to deal with such imprecise cardinality estimates. To deal with cardinality estimates that are too high, a final Stop operator is required at the top of the plan in order to make sure that no more than 100 query results are produced; to deal with overly low cardinality estimates, the plan must somehow be *restartable* at run-time in order to produce the missing tuples which were discarded by the unsafe Stop operator.

In [CK97], we studied the tradeoffs associated with an approach that allows *unsafe* Stop operators. We saw that in some cases, it is indeed better to have low-level *unsafe* Stop operators even when the cardinality estimates are imprecise. However, in other cases, we observed that excessive restarts due to overly low estimates hurt performance dramatically; in extreme cases, the performance of the enhanced system with *unsafe* Stop operators even dropped below the performance of a traditional system with no support for *top N* and *bottom N* queries. In contrast, an enhanced system with only safe Stop operator placement can never be outperformed by a traditional system.

3.2 Support in Current Database Systems

As stated in the introduction, several database vendors have recently added features that provide some degree of support for top and bottom queries. Unfortunately, none has published information how they have implemented those features. Thus, here we will discuss only the current version of DB2 for Common Servers (Version 2.1.1), as this is the only system that we were able to learn about and experiment with for the examples presented in Section 2. [2]

DB2 allows users to specify that they want the first N answers of a query quickly via an `OPTIMIZE FOR N ROWS` clause that can be given as an optional suffix for `SELECT` queries. The presence of an `OPTIMIZE FOR N ROWS` clause influences query processing in DB2 in two ways. First, it passes the value of N to the run-time system of DB2 so that DB2 can adjust its row blocking; as a result, DB2 can do well on the first example of Section 2 if the query is given as an `OPTIMIZE FOR 100 ROWS` query. Second, the presence of the `OPTIMIZE FOR N ROWS` clause makes the DB2 optimizer favor pipelined plans; this favoritism would allow DB2 to also perform well in the second and fourth examples of Section 2 [Cor95]. It is important to note, however, that DB2's `OPTIMIZE FOR N ROWS` clause is only a *hint* that instructs DB2 to produce the first N answers quickly; users are allowed to consume more than N tuples (at potentially lower performance) even when they give such a hint, whereas our `STOP AFTER` clause instructs a DBMS to produce exactly N (and never more) answers. DB2 thus never discards answer tuples, preventing it from exploiting some of the significant cost savings that are available through the use of specialized Stop operators (especially in non-pipelined query plans, such as those needed in the third and fifth example of Section 2). DB2 has not (yet) integrated any of the techniques of our approach described above.

4 Conclusion

We have shown various ways in which traditional database systems are susceptible to wasted work when evaluating *top N* and *bottom N* queries. We have seen that, in many cases, orders-of-magnitude improvements can be achieved with moderate extensions to the parser, query engine, and optimizer of an existing database system.

Most of the discussion of this paper has been concerned with *top N* and *bottom N* queries where the requested result cardinality (N) was specified in the query as part of a basic `STOP AFTER N` clause. Another important

[2] From comments in the manuals, it seems that Oracle 7 has followed an approach similar to that of DB2 [ABF+92].

class of queries are *percent* queries, which ask for, e.g., the top P percent of the answer set rather than the top N answers. Percent queries offer even more opportunities to avoid wasted work, though they also require additional work in order to determine what $P\%$ of the answer really is (cardinality-wise). We plan to study good strategies for percent query processing as future work; in addition, we plan to study specialized algorithms for sorts and joins in the context of any kind of top or bottom query, and and we plan to investigate new techniques for processing top and bottom queries in the context of subqueries and views.

References

[ABF$^+$92] E. Armstrong, S. Bobrowski, J. Frazzini, B. Linden, and M. Pratt. *ORACLE7 Server – Application Developer's Guide*. Oracle Corporation, Redwood Shores, USA, 1992.

[CG96] S. Chaudhuri and L. Gravano. Optimizing queries over mulitmedia repositories. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 91–102, Montreal, Canada, June 1996.

[Cha96] D. Chamberlin. *Using the New DB2: IBM's Object-Relational Database System*. Morgan-Kaufmann Publishers, San Mateo, USA, 1996.

[CK97] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 219–230, Tucson, USA, May 1997.

[Cor95] IBM Corporation. DB2 application programming guide for common servers. Manual, 1995.

[GD87] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, USA, May 1987.

[Knu73] D. Knuth. *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison-Wesley, Reading, USA, 1973.

[KS95] R. Kimball and K. Strehlo. Why decision support fails and how to fix it. *ACM SIGMOD Record*, 24(3):92–97, September 1995.

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, USA, May 1988.

Online Processing Redux

Joseph M. Hellerstein
U.C. Berkeley

Abstract

The term "online" has become an all-too-common addendum to database system names of the day. In this article we reexamine the notion of processing queries online. We distinguish between online processing and preprocessing, and argue that online processing for large queries requires redesigning major portions of a database system. We highlight pressing applications for truly online processing, and sketch ongoing research in these applications at Berkeley. We also outline basic techniques for running long queries online. We close by reevaluating the typical measurements of cost/performance for online systems, and propose a mass-market approach for designing and measuring data-intensive processing.

1 Introduction

In the parlance of today's database systems, "online" signifies "interactive", "within the bounds of patience." Online processing is the opposite of "batch" processing. In the dark days of computing, all serious work was done in batch mode — the COBOL or FORTRAN programmer submitted her "job" to the machine operator and busied herself with other tasks; the expectation was that the computer would be occupied for some time generating a solution. By contrast, newer "online" systems would return an answer while the programmer remained connected to the system. Because early systems typically required users to wait for the completion of one job before starting another, online processing had to have interactive performance.

For many of today's programmers – this author included – batch processing as described above is a distant memory. Yet there are still a variety of scenarios where computers tax the patience of both programmers and naive users. Most of these scenarios arise when processing significant amounts of data, either through a query engine, a network, or both. The prevalance of these problems in data-intensive systems has led many database vendors to describe their systems' processing as "online", particularly the recently much-ballyhooed "On Line Analytic Processing" (OLAP) systems.

In this paper, we revisit the phrase "online" as it applies to data-intensive applications. First, we argue that the term should be used more carefully — in particular, we point out that many of today's so-called OLAP systems do not process online at all. Second, we highlight a large class of applications which require online processing over large data sets. We present a suite of techniques for achieving online performance even for large, data-intensive jobs. We close by calling into question the commonly-held economic metrics for data-intensive systems.

Copyright 1997 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

1.1 Today's OLAP? PAP

The OLAP products available today provide interactive drill-down, roll-up and cube facilities [GBLP96]. Implementation techniques differ: some systems implement these facilities over relational query engines ("ROLAP"), others implement special-purpose multidimensional storage and retrieval engines ("MOLAP"). The interfaces are indeed "online", in the sense that users see interactive performance — with each "point" or "click", information is consolidated and displayed almost instantaneously.

OLAP operations typically involve major fractions of large databases. It should therefore be surprising that today's OLAP systems provide interactive performance. The secret to their success is *precomputation* — in most OLAP systems, the answer to each point and click is computed long before the user even starts the application. In fact many OLAP systems do that computation relatively inefficiently [ZDN97], but since the processing is done in advance the end-user does not see the performance problem.

Observe that today's OLAP systems do essentially no processing — online or otherwise — while the end-user is running them. They are better termed Precomputed Analytic Processing (PAP) systems. These systems have inherent space and time problems. Everything a user can do at the front end must be at least partially precomputed in order to guarantee interactive performance. This means that the system must precompute and store a large amount of information, some of which may never be used. This approach remains untenable today for large databases.

1.2 Truly Online Processing

Today's OLAP systems were intended to be a workable alternative to the batch-like speeds of traditional query engines, which discourage large scale data "browsing" and analysis. If traditional engines have batch behavior, and today's OLAP systems are really PAP, how do we implement truly online processing for large-scale data-intensive problems?

The answer is to change query processing — and indeed all data-intensive software — so that it runs in an online fashion. This does not mean traditional performance improvements involving expensive equipment and hand-tuned software. Rather it involves:

1. Carefully redesigning algorithms so that long-running operations return steady, incrementally improving estimates.

2. Allowing users to control the behavior of long-running operations on the fly.

Developing software that behaves this way requires a significant shift in focus, and major changes in implementation. It cannot be done with a simple object-relational "plugin module" or web "applet". On the other hand, as we show below it is possible to provide online processing with clean, modular modifications to a database system or application.

In the remainder of this paper we highlight areas where truly online processing is becoming critical, and then describe some basic implementation techniques to make it possible.

2 Critical Applications for Online Processing

Batch-style behavior remains a problem in a number of applications. The result of this is either that the application is frustratingly slow (discouraging its use), or the user interface prevents the application from entering batch states (constraining its use.) The applications in this section are currently being handled with one or both of these approaches.

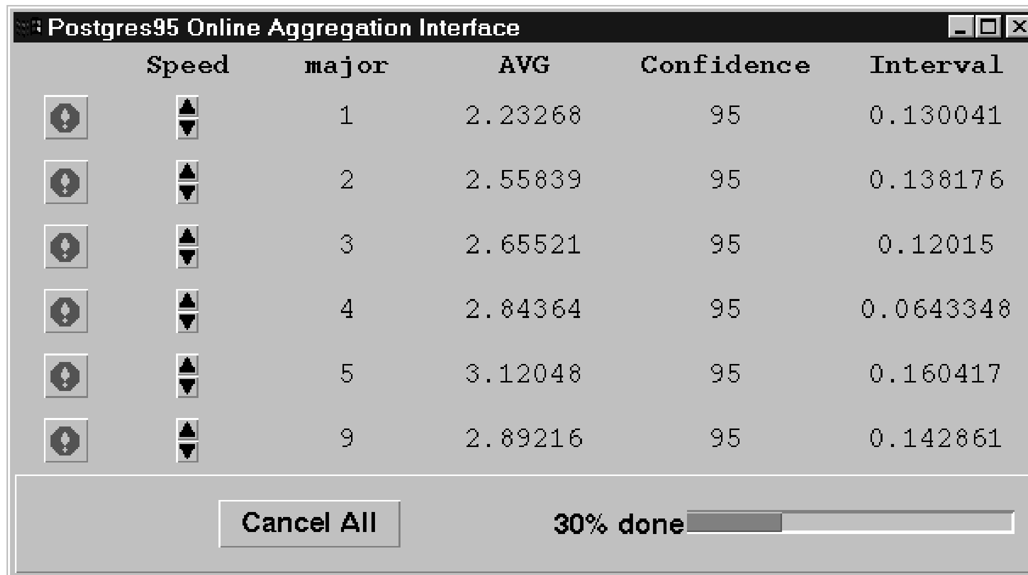

Figure 1: A Speed-Controllable Multi-Group Online Aggregation Interface

2.1 Online Aggregation

Aggregation queries in relational database systems often require scanning and analyzing a significant portion of a database. In current relational systems such queries have batch behavior, requiring a long wait for the user. In [HHW97] we detail an Online Aggregation system we are building at Berkeley, which performs relational aggregation queries in an online fashion.

Consider the following simple relational query:

```
SELECT AVG(final_grades) FROM grades
GROUP BY major;
```

The output of this query in an online aggregation system can be a set of interfaces, one per output group, as in Figure 1. For each output group, the user is given a current estimate of the final answer, and a "confidence interval", which says that with x% confidence, the current estimate is within an interval of size k from the final answer. A status bar at the bottom of the screen shows how much processing time remains. These interfaces expose salient features of the current state of processing, and predict the final outcome in a statistically accurate fashion. In addition, controls are provided to stop processing on a group, or to speed up or slow down the group relative to others. These interfaces require the support of significant modifications to a relational DBMS. Many of the implementation themes which support online aggregation are described in Section 3.

2.2 Online Data Mining

Data mining algorithms typically run for hours on large datasets without producing output. While running, they usually make at least one complete pass over the database. A particularly frustrating aspect of data mining algorithms is that they are not only slow, but also opaque; they are "black boxes" that do not allow users to tune them in significant ways without starting over.

Consider, for example, the common algorithms for finding "association rules" in market-basket data [AS94]. The goal of these algorithms is to find sets of items that people tend to buy together in a single trip to a store (e.g., rules of the form "people who buy diapers also tend to buy beer").

To use an association rule application, a user specifies values for two variables, one that sets a minimum threshhold on the amount of evidence required for a set of items to be produced (*minsupport*) and another which

sets a minimum threshhold on the correlation between the items in the set (*minconfidence*). The system begins its multi-hour undertaking, at the end of which it produces association rules which passed the threshholds of minimum support and confidence. Woe to the user who sets those threshholds incorrectly! Setting them too high means that few rules are returned and the process must be restarted. Setting them too low means that the system (a) runs even more slowly, and (b) returns an overwhelming amount of information, most of which is useless.

The association rules techniques can be easily "brought online", and a prototype implementation has been undertaken at Berkeley [ASY97]. While the algorithm is in its first phase producing individual items (1-itemsets) that have minimum support, it can provide running estimations of support for 1-itemsets in the same manner as an online aggregation query for `COUNT`. During this process, various items can be removed from consideration by the user, potentially speeding up computation of larger itemsets. Similarly, the user can modify minsupport and minconfidence as desired. Later phases of the algorithm are somewhat different from online aggregation processing, but still allow the user to add and remove itemsets from consideration, modify minsupport and minconfidence, and watch itemsets being generated. When a user changes the itemsets considered at any level (e.g., deletes a 2-itemset S from consideration), the system must spawn a new thread to "percolate" this change upwards through larger itemsets (e.g., delete all itemsets that are supersets of S). When all threads of the online data miner terminate, the user is still free to modify the support and confidence, or explicitly cause itemsets of any size to be added to or deleted from consideration — the system simply starts a new thread to differentially handle any newly added or deleted itemsets from that stage onward.

Most other data mining algorithms (clustering, classification, pattern-matching) are similarly time-consuming, and also relatively easily brought online. Note that interactive, online data mining is closer to data *browsing* — this should be particularly appealing to users who are skeptical that a computer can find all their answers automatically.

2.3 Online GUI Widgets

A common criticism of database systems is that they are much harder to use than desktop applications like spreadsheets. A common criticism of spreadsheets is that they do not gracefully handle large datasets. An inherent problem is that many spreadsheet behaviors are painfully slow on large datasets, despite impressive improvements from the commercial vendors. The problems typically have to do with the point-and-click nature of the GUI widgets used in spreadsheets.

Consider the common "list box" widget — it allows the user to bring up a list and scroll through it, or jump to particular points by typing prefixes of words in the list. Now imagine implementing a list box over gigabytes of unsorted, unindexed data resulting from an *ad hoc* query. This is likely to be rather unpleasant to use. Similarly unpleasant behavior arises when sorting, grouping, recalculating or cross-tabulating over large datasets — activites which are usually interactive in spreadsheets. Today's desktop applications carefully, almost imperceptibly discourage these behaviors over large unindexed datasets. There are many scenarios where such behavior is desirable, however; in such cases unwieldy database software (e.g., OLAP and Data Warehousing) is employed, requiring batch performance and lots of disk space to set up.

It should be possible to put online processing behind many of the widgets found in desktop applications so they continue to run smoothly over large datasets. Many development environments and toolkits today include libraries of GUI widgets. A data-intensive online GUI widget is a natural extension. In some instances, online GUI widgets could present different interfaces than standard widgets, in order to represent the state of ongoing processing — a trivial example is the "heartbeat" given by many web browsers as they present text and images online. In other instances status information might be unnecessary. The code underneath these widgets would need to process data using the kinds of techniques described in Section 3.

2.4 Online Data Visualization

Data visualization is an increasingly active research area, with some impressive prototypes under development (e.g. Tioga Datasplash [ACSW96], DEVise [LRB+97], Pad [PF93]). These systems are all interactive, allowing users to “pan” and “zoom” in datasets, and derive and view new visualizations quickly.

An inherent challenge in architecting a data visualization system is that it must present large volumes of information efficiently. This involves scanning, aggregating and rendering large datasets at point-and-click speeds. Typically these visualization systems do not draw a new screen until its image has been fully computed. Once again, this means batch-style performance for large datasets. This is particularly egregious for visualization systems that are expressly intended to support browsing of large datasets.

A natural solution is to extend a visualization system to draw objects as soon as they are fetched from the database. For example, as soon as a tuple is fetched, a corresponding point can be plotted on a map or graph. An more complex alternative we are exploring is to combine this incremental, sampling-like access with network data encoding. We model the final state of the screen as a *single aggregate object* to be estimated. After scanning a number of tuples, we use them as a sample to estimate the first few coefficients of a wavelet encoding of the final image. As more tuples are scanned, this estimate is refined. The user sees the picture improve much the way that images become refined during network transmission. This may be particularly useful when a user pans or zooms on a canvas, when the accuracy of what is seen is not as important as the rough outlines of the moving picture.

3 Basic Techniques for Online Processing

We believe that truly online processing can be built out of a number of relatively simple operators, much as traditional query processing engines are built. In this section we highlight some of the techniques we are studying. We make no claim to be exhaustive. However these techniques appear to be effective for applications we have studied, and are suggestively different from traditional algorithms for handling large datasets.

3.1 Sampling and Statistics

For an online estimation to be representative, it has to be built on some appropriate sample set of inputs. Olken proposed sampling access methods [Olk93], and these are quite applicable for applications which require guarantees of randomness during online processing. In many cases more traditional access methods may be used, as long as the order of access is expected to be uncorrelated with the estimation being made.

In the Online Aggregation project, we have exploited and developed results in statistics to provide estimators for common SQL aggregation functions (`COUNT, SUM, AVG, STDDEV`), and confidence intervals of the sort seen in Figure 1 [Haa96, Haa97a]. These kinds of estimations are obviously useful for Online Aggregation. They may also be useful in applications like online visualization and mining, where the system needs to decide when to modify previous estimations presented to users.

3.2 Striding Access Methods

Given a dataset which can be decomposed into groups, *striding* access methods provide delivery of tuples from the different groups at user-controllable relative rates. The Online Aggregation interface in Figure 1 bases its “speed” buttons on striding access methods.

The Index Stride access method was presented in [HHW97]. Given a B-tree index on the grouping columns,[1] on the first request for a tuple we open a scan on the leftmost edge of the index, where we find a key value k_1. We assign this scan a search key of the form $[= k_1]$. After fetching the first tuple with key value k_1, on a subsequent request for a tuple we open a second index scan with search key $[> k_1]$, in order to quickly find the next group in

[1] Index striding is naturally applicable to other types of indices as well, but we omit discussion here due to space constraints.

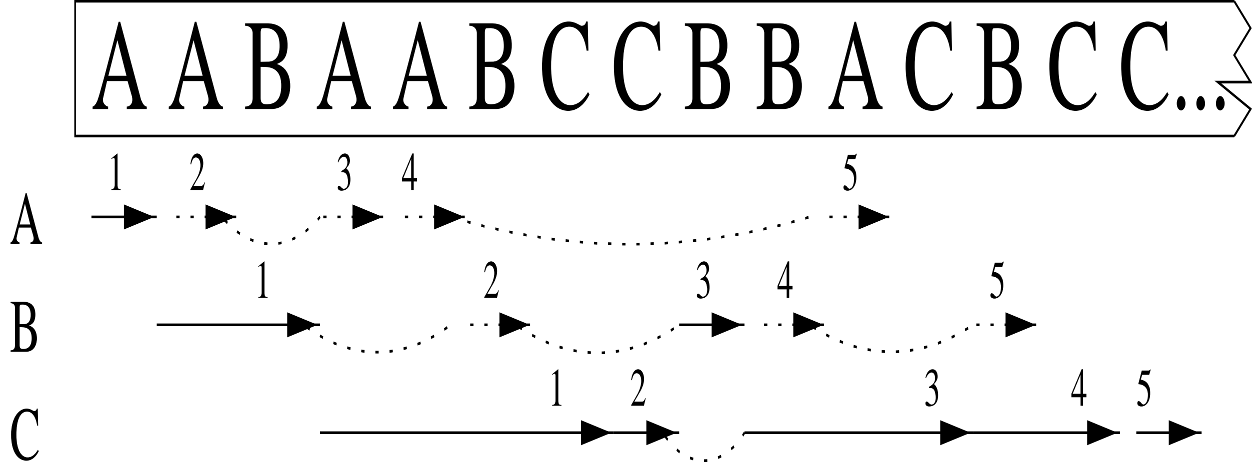

Figure 2: Heap Striding. The large letters represent the group attributes of tuples in a heap file. The three rows of arrows represent the accesses per group, with solid lines indicating initial I/Os and dotted lines indicating tuples which are revisited for delivery. The numbers represent the "strides" through the groups.

the table. When we find this value, k_2, we change the second scan's search key to be $[= k_2]$, and return the tuple that was found. We repeat this procedure for subsequent requests until we have a value k_n such that a search key $[> k_n]$ returns no tuples. At this point, we satisfy requests for tuples by fetching from the scans $[= k_1]$, ..., $[= k_n]$ in a (possibly weighted) round-robin fashion.

Striding can be done without an index as well. We are currently developing a Heap Stride access method, for which we briefly give intuitions here. Although heap striding scans a relation sequentially, it does not necessarily output tuples as soon as they are scanned. This is done because of upstream processing (joins, computation, etc.) that may need to occur before the tuple is delivered to the user — fairness dictates that time for such processing not be spent on a group until it is that group's turn.

Like index striding, heap striding opens a cursor per group as new groups are encountered. The distinction is that this is done while sequentially scanning a file (or "heap", in database terminology.) To begin, a cursor is opened at the start of the heap, and the first tuple is returned, say with group 'A'. The cursor is moved forward in the file *without returning any tuples* until a tuple from a different group is found, say group 'B'. The location of each additional 'A' tuple encountered while looking for a 'B' is enqueued on a "to-do" list associated with the 'A' cursor. This process continues until either (1) all groups have been encountered, or (2) it is considered beneficial to perform a round-robin tour of the existing groups before searching further for new groups. Decision (2) is made based on stored statistics, number of available buffers, upstream processing overheads, and other factors affecting the costs and benefits of returning tuples vs. further exploration of the heap. Round-robin processing is done by visiting each cursor and either fetching the next tuple in its to-do list, or (if the list is empty) exploring the heap file further as described above. A picture of fair round-robin processing is shown in Figure 2; the resulting output pattern would be "ABCABCABCABCABC...".

We believe we can realize a heap striding access method to provide efficient, index-free strided access. With intelligent buffering it can still be possible to perform a single I/O per block of a file during heap striding, so the postponement of tuple delivery may not produce associated I/O costs. Many design issues remain in determining policies for managing buffers and to-do lists, and in making tradeoffs between overall efficiency and meeting user goals on relative rates for different groups.

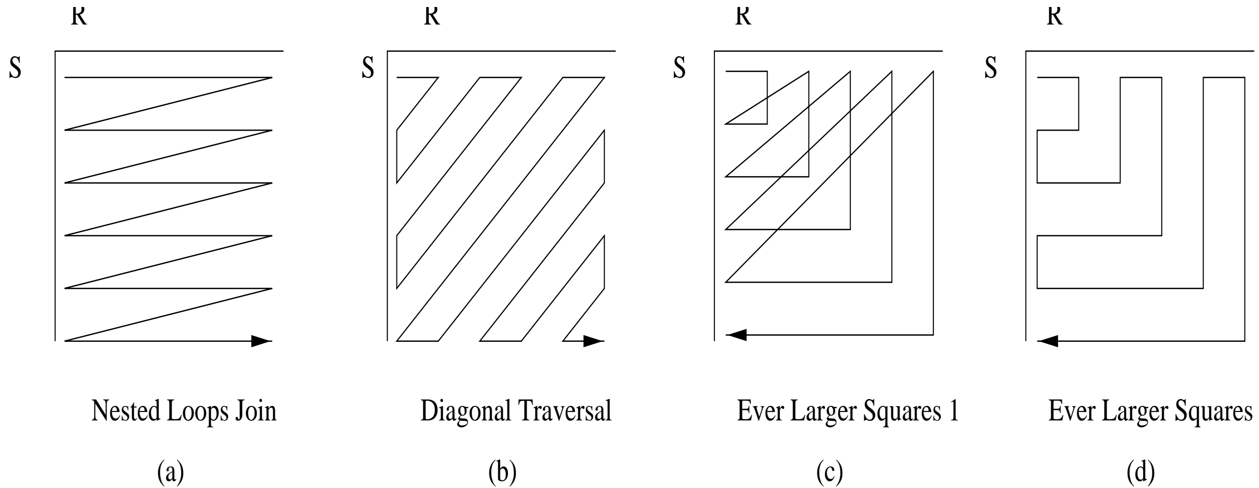

Figure 3: Join Matrix Traversals

3.3 Indexes and Compression

Indexes can be exploited in a variety of ways for online processing. We have seen how they can be used for striding access. They can also be used more directly to provide refining estimations of a dataset.

Consider the root of a B+-tree index. It contains a set of keys $k_1, \ldots, k_n$ which partition the dataset into $n + 1$ buckets of roughly equal size. This can be thought of as a representation of the distribution of values in the indexed column. In fact, any level of depth d in the tree is a rough equi-depth histogram of $(n+1)^d$ buckets. Thus a standard B+-tree can be traversed in *breadth-first* order to produce an iteratively refining estimate of a data set [Ant93].

Two-dimensional indexes like R-trees can be used to iteratively refine graphical or geographical datasets, and are perfectly suited for online data visualization. Compression techniques like wavelets are commonly used in online network delivery of images. It happens that the standard Haar wavelet encoding [Fal96] is extremely close to the traditional quad-tree [FB74], which is in turn isomorphic to the Z-ordering of Orenstein [Ore86]. This analogy motivates and justifies the use of multi-dimensional indexes for online delivery of data, and conversely suggests interesting directions for online indexing and search, using the wavelet estimation described above to model the eventual state of an n-dimensional search tree.

More flexible trees can be used for online processing yet more effectively. Ranked trees [Olk93] and pseudo-ranked trees [Ant92] allow more accurate “bucket depth” information to be included with the keys. Generalized Search Trees (GiSTs) allow essentially arbitrary flexibility in keys [HNP95]; Aoki has proposed extensions to the GiST framework [Aok97] to encompass statistical (non-restrictive) keys and arbitrary traversal patterns.

3.4 Plane-Sweep Joins

Haas has recently noted that nested loops join is not necessarily appropriate for online estimation; this is because one can tighten a confidence interval only once per tuple of the outer relation [Haa97b]. As an alternative, in joint work we are considering joins which draw on ever larger combinations of tuples from the two input relations. If we picture the tuples in the two relations being laid out as axes of a matrix, nested-loops join visits entries in the matrix in row-major fashion, and confidence intervals shrink at the end of each row (Figure 3a). As an alternative, we propose sweeping the matrix diagonally (Figure 3b), or via ever-larger squares (Figure 3c, 3d). For such traversals, confidence intervals can be shrunk every time a square submatrix containing (1,1) is completely traversed. We are currently investigating efficient versions of such joins.

3.5 Competition and Multi-Threading

Sometimes it can be difficult to predict the processing scheme that will produce the best behavior. Antoshenkov has proposed running multiple access methods in competition to resolve this problem in standard relational systems [Ant93, AZ96]. This approach is particularly important in online processing: one processing scheme may produce online results quickly, while another may produce final answers far more quickly than the first. In such a scenario it can make sense to execute both schemes, letting the first one produce estimates until the second is complete.

A similar theme of multi-threading is useful in a non-competitive manner. In a multi-phase algorithm like mining association rules, user modifications at one phase can often be propagated through later phases differentially. To make this work in an online setting, the original algorithm and the user modification should each represent a thread of processing — it is both correct and beneficial to execute many such threads simultaneously, to encourage online feedback to user input as well as to promote progress toward completion [ASY97]. Scheduling of the different threads should be handled carefully to maximize feedback throughput. It may make sense in some scenarios to allow users to adjust the thread scheduling via relative speed controls.

4 Conclusion

It is our belief that online processing is so natural as to require little further motivation. After a brief digression into the economics of computing and the art of benchmarking, we conclude with some side benefits of research into online processing.

4.1 Online Economics and Benchmarking

Typical cost/performance benchmarks (TPC benchmarks, Dollar Sort, etc.) compare the cost and speed of different hardware/software solutions. Analogies are sometimes made between these benchmarks and auto racing (e.g., the "Indy" and "Daytona" versions of sorting benchmarks [NBC+94]). If we apply the logic of such benchmarks to automobiles, we are led to buy *the fastest car we can purchase within our budget.*

Few of us use such logic in choosing a car. A more reasonable way to choose a car is to buy the most reliable car that matches our budget and functionality needs. Bringing the analogy back to computing, we really want a computing system that provides quick, reliable answers at a reasonable price. We should be willing to sacrifice some negligible reliability to save money. By this argument, a single PC running online software may be far more cost-effective than the price/performance winner of a TPC benchmark winner at a particular budget — the benchmark winner consumes enormous resources to compute a 100% accurate answer, whereas a 99% accurate answer is available to a slower system in the same amount of time. Most consumers will choose the Ford over the Mercedes.

4.2 Subsidiary Benefits of Online Processing

We have seen that by stressing interactivity over speed, online processing allows us to provide far cheaper, more usable solutions to computing challenges. This research has some additional benefits as well:

- Online processing is a natural and long-sought [BDD+89, SSU90, SAD+93] meeting point for research in databases and user interfaces.
- The current "database dinosaurs" have no clear advantage in developing online processing systems. This is an inherently lightweight systems domain, and the field is open.
- Online systems provide "crystal ball" rather than "black box" behavior, allowing users to predict and react to the system output rather than play time-consuming guessing games using trial-and-error or "relevance

feedback". This is especially important for IR and AI-based systems whose ill-defined semantics have to be guessed at by users.

- Online systems make impossible queries possible. This is no surprise to statisticians, census-takers and pollsters who can predict the outcome of events before they complete (or even before they start!). There is no reason we cannot do the same in software.

A variety of techniques beyond those described here have been emerging over the last few years; some of them are presented in companion articles in this bulletin. We expect online processing to become an important research and development theme over the next few years. We are currently implementing our online processing techniques and interfaces in the context of the Informix Universal Server and Informix MetaCube Explorer.

Acknowledgments

Thanks to Paul Aoki, Ron Avnur, Marti Hearst, and Allison Woodruff for feedback on an early draft of this paper. This work is funded by a grant from Informix Corporation and a University of California MICRO award.

References

[ACSW96] Alexander Aiken, Jolly Chen, Michael Stonebraker, and Allison Woodruff. Tioga-2: A direct-manipulation database visualization environment. In *Proc. 12th IEEE International Conference on Data Engineering*, pages 208–217, New Orleans, February 1996.

[Ant92] Gennady Antoshenkov. Random Sampling from Pseudo-Ranked B+ Trees. In *Proc. 18th International Conference on Very Large Data Bases*, pages 375–382, Vancouver, August 1992.

[Ant93] Gennady Antoshenkov. Query Processing in DEC Rdb: Major Issues and Future Challenges. *IEEE Data Engineering Bulletin*, 16(4):42–52, 1993.

[Aok97] Paul M. Aoki. Generalizing "Search" in Generalized Search Trees. Submitted for publication, June 1997.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proc. 20th International Conference on Very Large Data Bases*, pages 487–499, Santiago, September 1994.

[ASY97] Ron Avnur, Jonathan Spier, and Shu-Yuen Didi Yao. GOLD (GUI OnLine Data) Mining for Association Rules. Class project, CS286, U. C. Berkeley, May 1997.

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

[BDD+89] Philip A. Bernstein, Umeshwar Dayal, David J. DeWitt, Dieter Gawlick, Jim Gray, Matthias Jarke, Bruce G. Lindsay, Peter C. Lockemann, David Maier, Erich J. Neuhold, Andreas Reuter, Lawrence A. Rowe, Hans-Jorg Schek, Joachim W. Schmidt, Michael Schrefl, and Michael Stonebraker. Future Directions in DBMS Research — The Laguna Beach Report. *ACM SIGMOD Record*, 18(1):17–26, 1989.

[Fal96] Christos Faloutsos. *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, Boston, 1996.

[FB74] R. A. Finkel and J. L. Bentley. Quad-Trees: A Data Structure For Retrieval On Composite Keys. *ACTA Informatica*, 4(1):1–9, 1974.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Technical Report MSR-TR-95-22, Microsoft Research, 1996.

[Haa96] P. J. Haas. Hoeffding Inequalities for Join-Selectivity Estimation and Online Aggregation. IBM Research Report RJ 10040, IBM Almaden Research Center, San Jose, CA, 1996.

[Haa97a] Peter J. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *Proc. 9th International Conference on Scientific and Statistical Database Management*, Olympia, WA, August 1997.

[Haa97b] Peter J. Haas. Personal communication. May 1997.

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

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[LRB+97] Miron Livny, Raghu Ramakrishnan, Kevin S. Beyer, Guangshun Chen, Donko Donjerkovic, Shilpa Lawande, and Jussi Myllymaki. DEVise: Integrated Querying and Visualization of Large Datasets. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997.

[NBC+94] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A RISC Machine Sort. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 233–242, Minneapolis, May 1994.

[Olk93] Frank Olken. *Random Sampling from Databases*. PhD thesis, University of California, Berkeley, 1993.

[Ore86] J. A. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 326–336, Washington, D.C., May 1986.

[PF93] K. Perlin and D. Fox. Pad: An Alternative Approach to the Computer Interface. In *Proc. ACM SIGGRAPH*, pages 57–64, Anaheim, 1993.

[SAD+93] Michael Stonebraker, Rakesh Agrawal, Umeshwar Dayal, Erich J. Neuhold, and Andreas Reuter. Database Research at the Crossroads: The Vienna Update. In *Proc. 19th International Conference on Very Large Data Bases*, pages 688–692, Dublin, August 1993.

[SSU90] Abraham Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman. Database Systems: Achievements and Opportunities — The "Lagunita" Report. *ACM SIGMOD Record*, 19(4):6–22, 1990.

[ZDN97] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997.

Dynamic Query Processing in DIOM †

Ling Liu
Department of Computing Science
University of Alberta
lingliu@cs.ualberta.ca

Calton Pu
Dept. of Computer Science and Engineering
Oregon Graduate Institute
calton@cse.ogi.edu

Abstract

In an open environment such as the Internet, query responsiveness involves both the capability of responding to a query within a reasonable time frame and the capability of dynamically incorporating the new information sources and the up to date information of existing data sources into the answer of a query. In this paper we present the dynamic distributed query processing framework, developed in the DIOM project, to demonstrate the feasibility and the benefit of improving responsiveness of querying across heterogeneous information sources, without relying on an integrated global view schema pre-defined over all the participating information sources. Then we discuss several issues with respect to improving query responsiveness in the context of DIOM, including how the information consumers' queries are dynamically processed and linked to the relevant information sources, and how the query scheduling framework scales up in an environment where information sources accessible to users are constantly and rapidly changing, in their content, numbers, identity, and connectivity.

1 Introduction

The continuous advancement of wide-area network technology has resulted in a rapid increase in both the number of data sources available on-line and the demand for information access from a diversity of clients over the Internet or intranets. The availability of various Internet information browsing tools further promotes information sharing across departmental, organizational, and national boundaries. One of the most distinct features of such open and loosely-coupled distributed environments is that information sources accessible to a user are not pre-defined *a priori*, contrary to the traditional database assumption of a closed universe in centralized or tightly-coupled distributed environments.

A serious problem that arises in open environments is that of *query responsiveness*: a potentially very expensive query evaluation process due to the large and evolving number of information sources and joins, and the unpredictability in network configuration. There are several reasons that query responsiveness poses new technical challenges: First, it is well known that accessing multiple remote data sources over networks is expensive, and of the large collection of available information sources, only a small subset may actually contribute to the answer of a specific query. Second, data sources in open environments by their nature are heterogeneous and

Copyright 1997 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

† The first author is partially supported by NSERC grant OGP-0172859 and NSERC grant STR-0181014. The second author is partially supported by NSF grant IRI-9510112, and grants from Intel and Hewlett-Packard.

dynamically evolving. As a consequence, the query optimization and execution processes have to deal with the dynamic changes in content, numbers, identity, and connectivity of individual data sources. Third, the number of data sources that are available to a specific query varies from time to time not only due to the dynamic arrival of new information sources but also due to the vulnerability of data servers and communication links to the congestion on networks and the contention at sources [1].

We have identified three key steps in the optimization of query performance and responsiveness in open distributed data delivery systems: (1) to identify relevant information sources that can contribute to a specific query [4] based on both compile-time analysis and run-time information, thus minimizing the number of information sources accessed to answer a query, (2) to provide adequate support for establishing a dynamic interconnection between information consumers and information producers [6], thus allowing seamless incorporation of changes in content, numbers, identity, and connectivity of information sources into the planning and processing of queries, and (3) to consolidate run-time query plan modification techniques in the presence of delay and slow delivery of data or unexpected unavailability of information sources (see [1, 3, 4] or the papers appearing in this special issue).

Our work in DIOM [6, 4] has primarily targeted on the issues of source identification and dynamic interconnection. First, we have developed a three-level progressive pruning model and a set of dynamic query routing algorithms to identify relevant information sources for a query, thus reducing the overhead of contacting the information sources that do not contribute to the answer of the query. Efficient query routing not only reduces the query response time and the overall processing cost, but also eliminates a lot of unnecessary communication overhead over the global networks and over the individual information sources. Second, we have combined two independent but complementary strategies in the development of dynamic distributed query services for achieving higher system scalability and better query responsiveness:

- An incremental approach to construction and organization of information access through a network of domain-specific application mediators.
- A collection of facilities to allow information consumers to pose their queries on the fly and to dynamically link a user query to the relevant information sources.

The first strategy supports seamless incorporation of new information sources into the DIOM system. The second strategy allows the distributed query services to be developed as source-independent middleware services which establish the interconnection between consumers and a variety of information producers' sources at the query processing stage. As a result, the addition of any new sources into the system only requires each new source to have a DIOM wrapper installed. The DIOM services can dynamically capture the newly available information sources and seamlessnessly incorporate them into the distributed query scheduling process.

The main features that distinguish DIOM from other mediator-based approaches is the systematic development of a dynamic query mediation framework and the collection of dynamic query processing techniques for scalable data delivery [6, 5]. In what follows, we first give a brief overview of the system architecture and the dynamic query processing framework currently being implemented in the DIOM project. Then we describe the two main components of DIOM dynamic query processing: query routing and query planning, and their roles in improving query responsiveness. We conclude the paper with a summary and a brief report on our ongoing research and development.

2 Dynamic Query Processing: System Architecture

The DIOM system architecture [5] is a two-tier architecture offering services at both the mediator level and the wrapper level as shown in Figure 1.

A *mediator* is a software component which provides services to facilitate the integration and access of heterogeneous information. DIOM mediators are application-specific, consisting of a consumer's query profile and

many information producer's source profiles, described in terms of the DIOM interface definition language (DIOM IDL) [5]. The consumer's query profile captures the querying interests of the consumer and the preferred query result representation. The producer's source profiles describe the content and query capabilities of individual information sources. A *wrapper* is a software module providing an appropriate interface for a component data

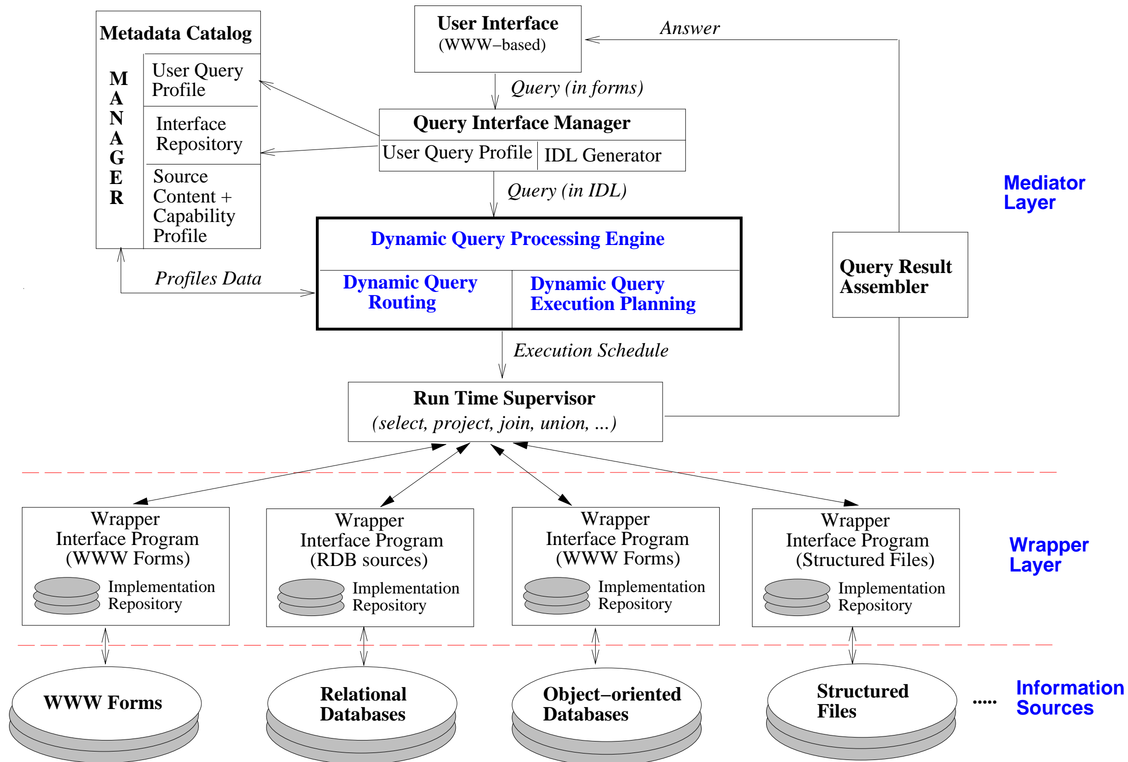

Figure 1: Dynamic Query Scheduling Framework

repository. The main task of a wrapper is to control and facilitate external access to the wrapped information repositories by using the local metadata maintained in the implementation repository through wrapper interface functions. *Information sources* at the bottom of the diagram fall into three categories: *well structured* (e.g., RDBMS, OODBMS), *semi-structured* (e.g., HTML files, text based records), or *unstructured* (e.g., technical reports). Each information source is treated as an autonomous unit. Information sources may make changes without requiring consent from any mediators. Currently DIOM also provides access to *unstructured* data (e.g., technical reports) through external WWW keyword based browsers such as Altavista, Infoseek, Yahoo. A DIOM mediator may access multiple and remote information sources through communication with their corresponding DIOM-to-source wrappers [5].

The DIOM dynamic query scheduling manager coordinates the communication and distribution of the processing of information consumer's query requests using the user query profiles and the description of the content and query capabilities of information sources. In addition, run-time information such as the intermediate results, the status of the networks and connections to information sources and the replication relationships among available information sources, is used for further refinements of the query routing result and the query execution plan. Figure 1 presents a sketch of the system architecture currently being implemented in the DIOM project. The main components of the DIOM distributed query scheduling manager are query interface manager, query routing, and query execution planning.

A user may submit his/her query on-the-fly, using WWW fill-out forms. The query interface manager in turn generates a user query profile which consists of (1) a form query in terms of the DIOM interface query lan-

guage (DIOM IQL) [5], (2) virtual interface classes, described in terms of the DIOM interface definition language (DIOM IDL) [5], which are used as template holders for receiving and representing the resulting objects of the query; and (3) user query annotation (optional), which uses an interactive interface program to allow the user to annotate the scope and context of what are to be expected in a specific query, including the input and output parameters of the query. Then the query is passed to the dynamic query processing engine for query routing and query planning.

- **Dynamic query routing:** The main task of dynamic query routing is to select relevant information sources from a large collection of available information sources which can contribute to answering a query. This is performed by dynamically relating the user query profile with the content and query capability descriptions of the sources.

- **Dynamic query execution planner:** The dynamic query execution planner consists of two steps: *query decomposition*, in charge of rewriting an IQL query, posed by the information consumer on the fly, into a group of subqueries, each targeted at a single data source, and *query scheduler*, which is responsible for generating a query execution schedule that is optimal in the sense that it utilizes the potential parallelism and the useful execution dependencies between subqueries to restrain the search space, minimize the overall response time, and reduce the total query processing cost.

Suppose we are interested in purchasing a book. Consider a query Q: "*Find title, authors, price, publisher, and reviews of science fiction that were published after 1996*". By using the IDL generator, we have all the parameters of interest generated, i.e., the category (science fiction, health, and so on), title, authors, price, publish-year, and publisher for `Book` objects and the book reviews for `Review` objects. By using the interactive interface program for user query profile generation, we have the query scope {`Book, Review`}, the list of input parameters {`category, publish-year`}, and the list of output parameters {`title, authors, price, publisher, reviews`}. We can further annotate, for example, whether we are interested in purchasing books from bookstores, bookclubs or publishers, what is our preferred currency unit for the price information, and so forth. An information source profile contains the content description (such as names of the relations and classes accessible at the source) and the query capability description (such as the list of mandatory or optional input parameters and the list of allowed output parameters). The user query profile and the producers' data source profiles play an important role in the query routing module to restrain the search space of the query by isolating the query within a subset of information sources that are relevant to the query.

Once the relevant information sources are selected, the query execution planner first decomposes the query into a group of subqueries, each targeted at a selected information source, and then finds a relatively optimal schedule that makes use of the parallel processing potential and the useful dependencies between subqueries. The ultimate goal of query planning is to reduce the overall response time and the total query processing cost. The run-time supervisor executes the subqueries according to the execution schedule produced by the query planner. It communicates with wrappers which control and facilitate external access to the information sources. Each wrapper translates the subquery request received from the run-time supervisor into an executable query program (or a function call) that can run at the local source. The result assembly process involves two phases for resolving the semantic variations among the subquery results: packaging each individual subquery result into a DIOM object (done at wrapper level) and assembling results of the subqueries in terms of the consumers' original query statement (done at mediator level). The semantic attachment operations and the consumers' query profiles are the main techniques that we use for resolving semantic heterogeneity implied in the query results [5].

3 Dynamic Query Routing

Query routing is the first step that constrains the search space for a query in open environments, in preparation for query planning. One of the main goals of query routing is to identify relevant information sources for a query as

early as possible, thus reducing the overhead of contacting the information sources that do not contribute to the answer of the query. Query routing is particularly critical in open environments that contain large and growing collections of heterogeneous information sources.

Given a user query Q, a user query profile of Q, and a set of source content and capability descriptions, we design the query routing service as a three-step process, which determines the relevance of the sources in answering a query by incorporating compile-time analysis with run-time information.

Step 1: Level-one relevance pruning prunes the information sources whose content descriptions are obviously irrelevant to the scope of the query Q (e.g., in terms of substring matching or ontology mapping used in the DIOM implementation). For level-one relevance pruning we use the user query scope description of Q and the content and category description of the sources. At level-one routing stage, the redundancy or replication of the sources will be detected and removed, based on the source replication description. Other factors such as unavailability of the sources or affordability of the sources should be considered at this step too. We refer to the set of sources selected by this step as *target* information sources of *level-one relevance*.

Step 2: Level-two relevance pruning prunes the information sources that have level-one relevance but do not offer enough query capability to contribute to the answer of Q, either due to the restriction on the scope of query parameters of Q, or the restriction on the list of input or output arguments of the sources, or due to the conflict of query interest with the access constraints associated with the sources. The user query capacity description of Q and the source capability profiles are used in the level-two relevance pruning. We call the set of sources selected by Step 2 as *target* information sources of *level two relevance*.

Step 3: Level-three relevance pruning explores run-time information to further prune irrelevant information sources to answer Q. For the atoms in Q that are of form $a\theta v$ where a is an attribute and v is a constant, if there are sources which take a or its alternative as one of the input arguments, then it is in general worthwhile to ask some subqueries to these information sources and use the run-time information returned (intermediate results) to continue to prune the number of information sources selected. This step is especially helpful when the subset of information sources resulting from level-two relevance pruning is not significantly small comparing with the large collection of available information sources. We call the set of sources selected by this step as *target* information sources of *level three relevance*. The level-three relevance pruning is built as a plug-and-play component and can be tuned to play any time when the need arises.

4 Dynamic Query Planning

Dynamic query planning component generates a parallel access plan for a group of subqueries, which is a relatively optimal schedule that makes use of the parallel processing potentials and the useful execution dependencies between subqueries. Two-phase reduction approach is used to reduce the solution search space based on built-in heuristics and cost estimation. A sketch of the two-phase query planning architecture is shown in Figure 2. The first processing phase is based on heuristic query optimization techniques. The second processing phase is the cost estimation and access plan scheduling.

The purpose of applying the heuristics first is to restrict the solution search space well enough so that the cost-based search is feasible and beneficial in the smaller solution space. The common heuristics used in the initial implementation of DIOM include (1) performing selections first; (2) performing joins that produce the smallest result earliest; and (3) performing *Cartesian product* last. The other heuristics observed in DIOM include (4) both union and Cartesian product should always be performed at the site where the result is expected because the communication cost of transferring the result is greater than the communication cost of transferring any one of the operands; (5) if the expected size of join result is smaller than any of its inputs, then it is beneficial to rewrite $\bowtie(\cup(Q_{11}, \ldots Q_{1n}), \cup(Q_{21}, \ldots Q_{2m}))$ into $\cup(\bowtie(Q_{11}, Q_{21}), \ldots, \bowtie(Q_{1n}, Q_{2m}))$; and (6) if the distribution of

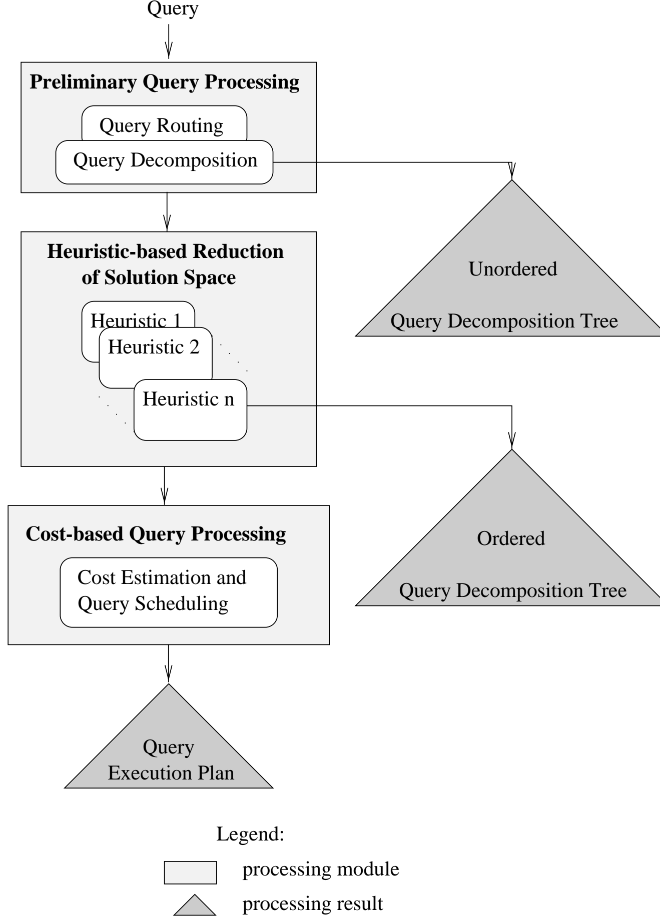

Figure 2: Two-phase query planning architecture

join over union does not lead to an increase in the number of inter-site joins, then such interchange is performed, otherwise, the order of operations remains unchanged. For detailed algorithms in heuristic-based optimization or cost-dirven optimization refer to [7].

To cater to the needs of various database users, in DIOM we provide a flexible framework for dynamic query planning, which allows the plug-in of query optimization components on demand, thereby providing user-driven and customizable query optimization. For example, in general, the cost of query execution consists of three independent factors: communication cost, local processing cost, and total response time cost. They may be combined additively into a generic goal formula shown in Equation 1.

$$Cost = a_{cc} \cdot C + a_{lqp} \cdot L + a_{rt} \cdot R = A^T \cdot \begin{pmatrix} C \\ L \\ R \end{pmatrix}, \text{ where} \tag{1}$$

C is the total amount of communications over the network spanning the distributed database expressed in time units;
L is the total amount of local query processing, also expressed in time units;
R is the total response time of the query.

For a given query Q, a tree will be generated after the query routing process (see Figure 2). Let us denote the left subtree Q_{left}, the right subtree Q_{right}, and the binary operator that takes Q_{left} and Q_{right} as inputs Q_{this}.

The communication cost $C(Q)$, the local processing cost $L(Q)$, and the total response time $R(Q)$ are computed recursively as follows:

$$\begin{aligned} C(Q) &= C(Q_{left}) + C(Q_{right}) + comm_cost(Q_{this}); \\ L(Q) &= L(Q_{left}) + L(Q_{right}) + loc_cost(Q_{this}); \\ R(Q) &= \max\left[R(Q_{left}), R(Q_{right})\right] + comm_cost(Q_{this}) + loc_cost(Q_{this}). \end{aligned} \tag{2}$$

where $comm_cost(Q_{this})$ is the cost of shipping data to the site where the operator Q_{this} will be executed, and $loc_cost(Q_{this})$ is the local processing cost of executing the operator Q_{this} at the chosen site. The coefficients associated with each of these components are the indicators of the desired optimization profile. They can be controlled by the user of the database by setting the profile via the components of vector A^T. The general cost estimation formula 1 serves as the goal function of the optimization process. For example, if the user's primary concern in finding an optimal query execution plan is the response time, then A^T is set to $(\,0 \quad 0 \quad 1\,)$. Vector $A^T = (\,0.3 \quad 0 \quad 0.7\,)$ would be specified by a user who would like to tune the dynamic query scheduler with respect to his particular requirement by allocating 30% of the total cost to the communication cost and 70% to the response time.

The first prototype implementation of DIOM dynamic query processing system (DQS) was developed and tested on Solaris platform using SunJDK version 1.1. The byte-code has been tested on the following platforms: Windows NT v.3.5.1 using Netscape Navigator v.2.01, and Sun OS v.4.1.4 using Netscape Navigator v.3.0. In the current prototype, the local processing cost estimation is based on the statistic information provided in the source query profiles. The communication cost estimation is based on the communication unit costs of connections, given by the mediator adminstrator. For the sources that have no statistic information available, or the network connections that have no communication unit cost, the system-defined default values will be used. Three main features that distinguish the DIOM dynamic query scheduling framework from conventional distributed query processing work: First, the multi-level progressive query routing process is a new query processing step that is not addressed in conventional distributed query processing paradigm. Second, in DIOM we not only provide the support for commonly used query optimization tactics such as the first three heuristics, but also introduce several semantic-based heuristics to enhance the quality of distributed query optimization solutions generated in DIOM. Third, we provide user-driven trace functions (capabilities) to allow the performance of distributed query processing to be tuned as necessary, such as the coefficients associated with the Equation 1, the decision of communication unit cost and local processing unit cost, etc.

5 Summary

We discussed the DIOM dynamic query processing framework and the strategies used for improving query responsiveness. The most interesting features that distinguish the dynamic query processing in DIOM from other approaches, such as Carnot, Garlic [2], TSIMMIS [8], are summarized as follows: First, we allow users to pose queries on the fly, without relying on a pre-defined view that integrates all available information sources. Second, we identify the importance of query routing step in building efficient query scheduling framework for distributed open environments. Third, we develop a three-tier approach (i.e., query routing, heuristic-based optimization, and cost-based planning) to eliminate the worst schedules as early as possible. Last but not least, we delay the hard semantic heterogeneity problems to the result assembly stage through semantic attachments rather than semantic enforcement.

Our ongoing research towards improving query responsiveness continues in several directions. We are currently evaluating query routing algorithms using more than a hundred of data sources, and designing algorithms for query result packaging and assembly. We are working on the generation of optimal parallel access plan that takes into account the efficient processing of aggregate functions such as SUM, COUNT, MAX and MIN. We are also interested in exploring possibilities to adapt and incorporate the state of art research results in improving

query responsiveness with the DIOM system, such as the query scrambling approach for dynamic query plan modification [1] and the online aggregation for fast delivery of aggregate queries by continuous sampling [3].

For additional information, including access to the demo and the more detailed technical reports and references about DIOM cited in this paper, the interested readers are invited to browse our WWW page: `http://www.cs.ualb`

References

[1] L. Amsaleg, M. J. Franklin, A. Tomasic, and T.Urhan. Scrambling query plans to cope with unexpected delays. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.

[2] L. Haas, D. Kossmann, E. Wimmers, and J. Yan. Optimizing queries across diverse data sources. In *The International Conference on Very Large Data Bases*, 1997.

[3] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[4] L. Liu. Query routing in structured open environments. Technical report, TR97-10, Department of Computing Science, University of Alberta, Edmonton, Alberta, Feb. 1997.

[5] L. Liu and C. Pu. An adaptive object-oriented approach to integration and access of heterogeneous information sources. *DISTRIBUTED AND PARALLEL DATABASES: An International Journal*, 5(2), 1997.

[6] L. Liu and C. Pu. A dynamic query scheduling framework for distributed and evolving information systems. In *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, Baltimore, May 27-30 1997.

[7] K. Richine. Distributed query scheduling in the context of diom: An experiment. MSc. Thesis, TR97-03, Department of Computer Science, University of Alberta.

[8] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *VLDB 96*, Bombay, India, Sept. 1996.

Dynamic Query Optimization in Multidatabases *

Fatma Ozcan Sena Nural Pinar Koksal Cem Evrendilek
Asuman Dogac
Software Research and Development Center
Middle East Technical University (METU)
06531 Ankara Turkey
{fatma,nural,pinar,cem,asuman}@srdc.metu.edu.tr

Abstract

In this paper, we describe a dynamic query optimization technique for a multidatabase system, namely MIND, implemented on a DOM environment. A Distributed Object Management (DOM) architecture, when used as the infrastructure of a multidatabase system, not only enables easy and flexible interoperation of DBMSs, but also facilitates interoperation of the multidatabase system with other repositories that do not have DBMS capabilities. This is an important advantage, since most data still resides on repositories that do not have DBMS capabilities. Dynamic query optimization, which schedules intersite operations at run-time, fits better to such an environment since it benefits from location transparency provided by the DOM framework. In this way, the dynamic changes in the configuration of system resources such as a relocated DBMS or a new mirror to an existing DBMS, do not affect the optimized query execution in the system. Furthermore, the uncertainty in estimating the appearance times (i.e., the execution time of the global subquery at a local DBMS) of partial results are avoided because there is no need for the dynamic optimizer to know the logical cost parameters of the underlying local DBMS.

In scheduling the intersite operations a statistical decision mechanism is used. The proposed scheme tries to exploit the inherent parallelism in the system as much as possible.

The performance of the developed method is compared with two other most related techniques and the results of the experiments indicate that the dynamic query optimization technique presented in this paper has better performance.

1 Introduction

A common characteristic of today's information systems is the distribution of data among a number of autonomous and heterogeneous repositories. Increasingly, these repositories are database management systems (DBMSs), but there is still a large volume of data that is stored in file systems, spreadsheets and others. An accepted approach

Copyright 1997 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

* This work is partially being supported by the Turkish State Planning Organization, Project Number: AFP-03-12DPT.95K120500, by the Scientific and Technical Research Council of Turkey, Project Number: EEEAG-Yazilim5, by Motorola (USA), by Sevgi Holding (Turkey) and by the Middle East Technical University, Project Number: AFP-97-07-02-08.

for achieving interoperability among DBMSs is the multidatabase approach. One restriction of multidatabase systems has been that they are able to provide interoperability only among DBMSs and can not handle repositories which do not have DBMS capabilities. The Distributed Object Management (DOM) approach has lifted this restriction [2, 9]. When a DOM architecture is used as the infrastructure of a multidatabase system, it easily becomes possible to provide interoperability of a multidatabase system with other repositories that do not have DBMS capabilities. What makes this possible is the encapsulation and abstraction capabilities of object-orientation that enable the development of wrappers which encapsulate a particular repository and provide a common DBMS-like interface to the rest of the system.

Yet, in order to exploit the location transparency feature of a DOM architecture, changes in multidatabase query processing are necessary. In this paper, we describe a dynamic query optimization technique implemented for a multidatabase system, namely MIND [5, 7, 8], realized on a DOM environment, namely CORBA (Common Object Request Broker Architecture) [22], which is a standard developed by OMG (Object Management Group).

In a multidatabase system when a global query is decomposed, two types of queries [17] are produced. One is the queries against the export schemas that are derived by translating the local schemas of the underlying databases into a canonical data model, the other is the post-processing queries [3] that combine the results returned by the LDBMSs (i.e., the partial results). Most of the previous work on global query processing has concentrated on producing a static execution plan for post-processing queries based on cost estimations. Some systems like Garlic [16], however, have looked at optimization for the entire query, not just the post-processing part. Due to local loads and autonomy of the local database systems, it is very difficult, if not impossible, to do a detailed cost-based query optimization. However, since the difference between a near-optimum and a bad execution plan may be enormous, finding an effective technique for optimizing post-processing queries is essential. Dynamic query optimization technique described in this paper is an attempt in this respect [19].

The technique proposed uses the available partial results at run-time without dictating an execution site. In scheduling the intersite operations between the partial results, the conditions between these results should be considered. Intersite operations such as join, outerjoin or union are required to combine the results returned by the local sites. Additionally, a decision mechanism is required to find out the execution order. We have used a statistical decision mechanism in the dynamic query optimization to handle this decision problem. In doing this another observation is the following: such a decision can not be solely based on costs of performing the intersite operations since selectivity of the intersite operation also matters. Thus, we define a weight function which considers both cost and selectivity.

The paper is organized as follows: In Section 2, the related work is summarized. Section 3 describes the implementation framework of the dynamic query optimizer, namely the MIND architecture. The dynamic query optimization algorithm that utilizes the query graphs is given in Section 4. In Section 5, we present the performance evaluation results of the proposed scheme, in comparison to two other related static optimization techniques. Finally, Section 6 contains the conclusions and future work.

2 Related Work

As we have stated, most of the previous work done in multidatabase query processing has concentrated in static optimization of post-processing queries. An exception is the work presented in [1] which modifies execution plans on-the-fly in response to unexpected delays that can arise due to communication problems in obtaining partial results. The algorithm proposed in [1] modifies the execution plan when a significant idling arise so that progress can be made on other parts of the plan.

In [24], a distributed query optimization algorithm is presented for multidatabases. The optimization algorithm is organized as a sequence of steps and at each step all local DBMSs work in parallel to evaluate the cost of execution plans for partial queries of increasing sizes, and send their cost estimates to the other local DBMSs that need them for the next step. No duplicate computation is performed and the execution plans that may not

lead to an optimal solution are discarded as soon as possible.

Although the query optimization is performed in parallel, this algorithm produces only serial query execution plans. Furthermore, the algorithm does not consider the plans with bushy inners.

In [11], a technique is suggested to reduce query response time in multidatabase systems. In this paper, the authors propose an algorithm that first produces an optimal left deep join tree and then reduces the response time using simple tree transformations. Three algorithms namely, top-down, bottom-up and hybrid are proposed to balance the left deep join trees. The overhead incurred by these algorithms are O(n^2), n being the number of nodes in the query graph. The algorithms do not guarantee that the resulting tree is optimal with respect to the total response time [11].

The algorithm that produces left deep join trees does not consider the bushy inners. The tree transformation algorithms suggested in [11] try to reduce the response time by finding cost reducing bushy inners. However, since they start with a left deep join tree, they may miss the least costly bushy inners.

In [12, 13, 14], a cost based query optimization strategy is given. In this strategy, the optimization algorithm tries to maximize the parallelism in execution while taking the federated nature of the problem into account. In optimizing intersite joins, a two-step heuristic algorithm is proposed which incorporates parameters stemming from the nature of multidatabases such as the time taken by global subqueries at the local DBMSs, transmission overhead incurred while communicating partial results during intersite join processing and the cost of converting global subquery results into the canonical data model. In this technique, the appearance times of the partial results are estimated by applying synthetic database calibration [10] and an execution plan is generated before sending the subqueries to the local DBMSs.

3 MIND Architecture

The dynamic query optimization scheme described in this paper has been implemented within the scope of the MIND project [5, 7, 8]. MIND is a multidatabase system prototype whose architecture is based on OMG's Object Management Architecture (OMA). The components of the MIND are designed as CORBA objects communicating with each other through an ORB.

In MIND, local DBMSs are encapsulated in a generic database object. MIND defines the generic database object in CORBA IDL with multiple implementations, one for each local DBMSs which are called Local Database Agents (LDA). There is one implementation for each of the local DBMSs; currently supported systems include Oracle7[1], Sybase[2], Adabas D[3] and MOOD (METU Object-Oriented Database System) [4, 6].

An overall view of MIND is provided in Figure 1. The basic components of the global layer in MIND consist of the following classes: Global Database Agent (GDA) class, Schema Information Manager (SIM) class and the Query Processor (QP) class. Global Database Agent objects are responsible for parsing and decomposing the queries according to the information obtained from the Schema Information Manager. They are also responsible for global transaction management.

Query processing in MIND is realized by the GDA object as follows: After a query is decomposed, the global subqueries are sent to the involved LDA objects. The optimization process starts after the first partial result from a LDA object becomes available. A QP Object performs intersite operations between two partial results that are available at run-time according to the dynamic query optimization scheme presented in this paper. These two results may be either partial results computed at the LDA objects or results of previous intersite operations available from QP objects. There could be as many QP objects running in parallel as needed to process the partial results. The GDA object controls and schedules all the intersite operations required for the execution of the global query.

[1] Oracle7 is a trademark of Oracle Corp.

[2] Sybase is a trademark of Sybase Corp.

[3] Adabas D is a trademark of Software AG Corp.

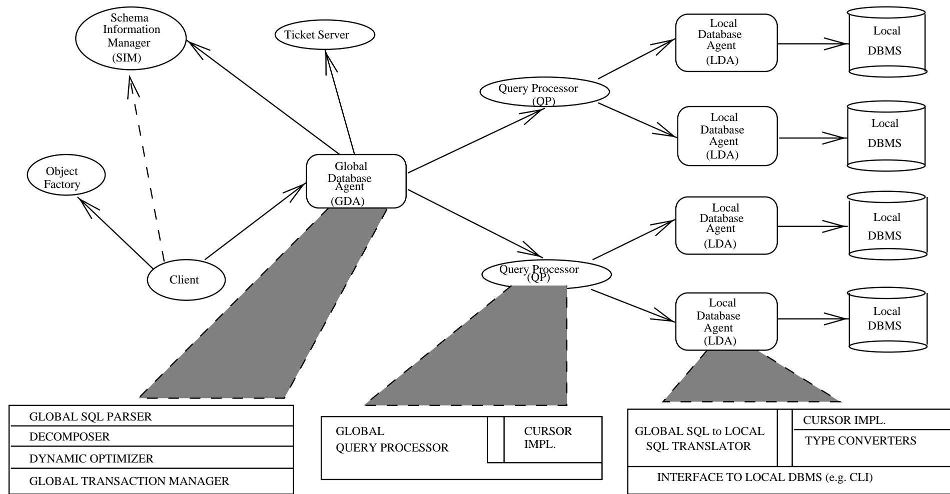

Figure 1: An Overview of the MIND Architecture

4 Dynamic Query Optimization

Since DOM architectures provide location transparent access to the objects, it is necessary to develop query optimization strategies based on this fact. The dynamic query optimization scheme [19], which is a step in this direction, uses the available partial results at run-time, instead of producing an execution plan based on the estimated execution time of subqueries. Thus, this scheme reduces the uncertainty in cost calculations, since it does not need to know the logical cost model of the underlying systems which may not be available due to local autonomy. Also, due to the local processing loads of the LDBMSs accurate estimations of the appearance times are very difficult.

In scheduling intersite operations between the partial results, the query graphs containing the conditions between these results are used. A decision mechanism is necessary to find out the execution order. A sample case is provided to point out the issues that should be considered: Assume that two partial results R_1 and R_2, with an intersite operation between them, are available from local databases, LDBMS1 and LDBMS2. Although it is possible to schedule this intersite operation immediately, it may be the case that a third partial result R_3, which appears at LDBMS3 at a later time, when processed with R_1, may reduce the partial result later to be processed with R_2 so much that the optimum execution time is obtained. So, delaying the processing of some partial results may be more efficient. However, the actual appearance time of R_3 is a required parameter in order to decide whether it pays off to wait for it. Moreover, this decision cannot be solely based on the cost of the intersite operation, since the selectivity of the operation also affects the optimum execution order. To make a knowledgeable guess on when to delay processing of a partial result, we have used a statistical decision mechanism in order to set a threshold value. In threshold calculations, a weight function is employed which considers the cost of an intersite operation, its selectivity and the transmission costs incurred. An intersite operation is scheduled unless the value of its weight function exceeds the threshold. Furthermore, the proposed scheme tries to exploit the inherent parallelism in the system as much as possible.

4.1 Cost Functions

In minimizing the total execution time of global queries, the critical part of the process is the scheduling of the intersite operations. The costs and selectivities of an intersite operation directly affect their order in an optimum solution. In other words, the less costly and more selective intersite operation must be favored as early as possible among the available alternatives. Also in providing the cost estimations in the dynamic optimization scheme,

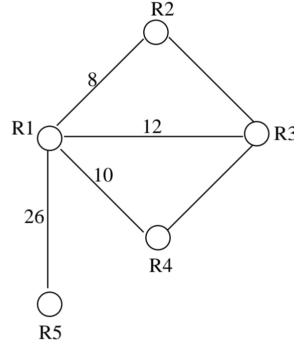

Figure 2: Determining the threshold

there is no need to consider appearance time and conversion costs since optimization process starts after the partial results become available in the canonical data model. In the light of this discussion, we provide the definition of the weight function for intersite operations.

Definition 1 The weight function, ψ(R$_i$,R$_j$), is defined as

$$\psi(R_i, R_j) = \frac{(IP_{cost}(R_i, R_j) + T_c(R_i) + T_c(R_j))}{(1 - \delta(R_i, R_j))}$$

where δ (R$_i$,R$_j$) is the selectivity of the intersite operation between R$_i$ and R$_j$, IP$_{cost}$(R$_i$,R$_j$) is the cost of this operation provided that the least costly method available for performing the intersite operation is chosen and T$_c$(R$_i$) is the cost of transferring R$_i$ from one CORBA object to another. If one of the objects, whose results are to be combined, is a QP object then one of the transmission costs T$_c$(R$_i$) or T$_c$(R$_j$) will be zero. For the estimation of $IP_{cost}(R_i, R_j)$, if the operation is join, three join methods, namely, hash-partition, sort-merge and nested loops with hashing are used and the cost calculations to select the least costly of these three join methods are performed as in [23]. If the operation is outerjoin then nested loops method is used.

In the next section, we discuss how statistics are employed in the scheduling of intersite operations by using the given weight function.

4.2 Determining the Threshold Value

In order to determine whether it pays off to delay an intersite operation, we propose setting a threshold value associated with the available partial result. An intersite operation involving this partial result is scheduled if the value of its weight function does not exceed the threshold.

The problem at hand is the following: There is a collection of weight function values calculated for every edge connected to an available partial result in the query graph. In this population, there may be values which are close to the mean and there may be values with large deviations from the mean. We need to find out those values with large positive deviation. We can determine such values by using the mean and the standard deviation of this population [15]. When a partial result, R$_i$ appears, the following formulas are used to calculate the mean, the standard deviation and the threshold respectively.

$$\mu = (\textstyle\sum_{j=1}^{n} \psi(R_i, R_j))/n, \qquad \sigma^2 = (\textstyle\sum_{j=1}^{n} (\psi(R_i, R_j) - \mu)^2)/n, \qquad \Theta = \mu + a\,\sigma$$

In the above formulas, n denotes the number of nodes connected to node n$_i$ (partial result R$_i$) and a is a constant which determines the distance from the mean. If this distance is short, then more of the weight function values will be rejected, implying more selectivity. Currently, a is taken as 1, i.e, the weight function values which are one or more σ far from the mean are rejected.

The following example is provided to clarify the use of threshold value.

Example 1: Consider the query graph given in Figure 2. Assume that the partial result R_1 appears. In order to calculate a threshold value for R_1, the weights of all edges connected to R_1 are calculated according to the weight function given in Definition 1. As can be seen from the query graph, the weights of the edges e_{12}, e_{13} and e_{14} are very close to each other, whereas the weight of the edge e_{15} is a lot larger. The aim is avoiding the scheduling of this intersite operation. For this population, the mean and the standard deviation are found as μ = 14 and σ = 7.071. By using these values, the threshold is calculated as Θ = 21.071 for which all edges, but e_{15}, can be scheduled.

The scheduling algorithm implementing the dynamic query optimization is given in [20].

5 Performance Evaluation

In this section, performance comparisons of the proposed optimization technique with the two most related techniques are presented. The first algorithm [11] (Hybrid Algorithm) tries to reduce the response time by tree balancing. And the second algorithm [12, 13, 14] tries to maximize the parallelism in execution by considering the parameters stemming from the nature of multidatabases. Both of these techniques perform static query optimization, that is, the execution plan is decided at compile-time. Therefore, the query execution times of the two algorithms with the proposed technique is examined.

In order to compare the performance of the algorithms, an execution environment is simulated. The simulation has two phases : In the first phase, an execution plan is generated based on the estimations of appearance times, and execution time of intersite operations for the two static techniques. In the second phase, the run-time environment is simulated. For the static techniques, the plans generated in the first phase are used in calculating the actual execution times. In the first phase cost estimations are used. For the estimations, the global query graphs are formed with 3 to 15 partial results. For each fixed sized query graph, the cardinalities, unique cardinalities and record widths are randomly chosen with different variances for a total of 10 test cases and the results are averaged. The sizes of the partial results changes from 4K to 1M and the selectivities are calculated with respect to cardinality and unique cardinality values are calculated assuming uniform distribution for each partial result. The calculation of the cardinality and unique cardinality values for intermediate results are performed as in [21]. Initial appearance times for the partial results are also randomly chosen assuming that the sizes of the partial results are uncorrelated with the appearance times but correlated with the type of the global subquery. Communication cost and conversion cost for intermediate results are taken to be proportional to the sizes of intermediate results. The left deep tree used as the initial input for the Hybrid Algorithm is obtained as in R* [18].

In the second phase of the experiments the effect of the deviation of the estimated costs from their actual values is explored. For this purpose, programs are run for five different deviations from the estimations. The actual appearance time is assumed to deviate at most 10%, 20%, 30%, 40% and 50% from the estimated time and the execution times of the intersite operations are assumed to deviate at most 20% from their estimations. The deviation of the execution times of the intersite operations is assumed not to exceed 20%, since these operations are performed by QPs whose logical cost parameters are known.

Figure 3 depicts the results of the experiments. As shown in Figure 3, the difference between the Hybrid Algorithm and the dynamic technique increases as the number of partial results increase.

Dynamic query optimization technique proposed in this paper is slightly better than the technique proposed in [13, 14] which requires database calibration [10] which is a costly process. Moreover, the technique proposed is more suitable to DOM platforms and also its implementation is straightforward.

6 Conclusions and Future Work

Contrary to static optimization, which decides on the execution sites and the execution plan apriori, the dynamic query optimization technique proposed in this paper schedules the intersite operations at run-time without dic-

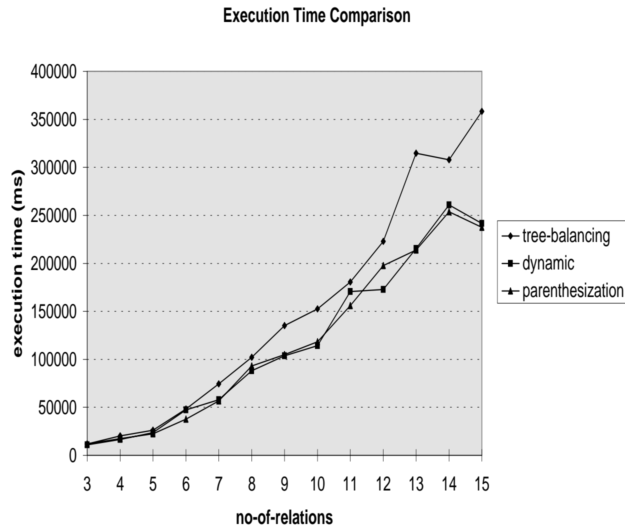

Figure 3: Execution time comparison of the three algorithms

tating an execution site. First, it has been observed that dynamic optimization fits better to a DOM platform by exploiting the location transparency property of the DOM environment. Second, using the actual appearance times of global subqueries at run time instead of their estimations is beneficial since it is difficult to estimate the appearance times due to local autonomy and local workloads of the underlying databases. However, a decision mechanism is required in scheduling intersite operations in order to perform optimization. For this purpose, a threshold value is used. In the threshold calculations, a weight function is employed which considers the selectivity of the intersite operation, in addition to its cost. An intersite operation is scheduled if the value of its weight function does not exceed the threshold.

The performance results show that the proposed technique is an effective one and performs better than the algorithm given in [11]. Although the proposed technique performs almost the same as the technique given in [13, 14], it does not require costly operations such as database calibration.

References

[1] L. Amsaleg, M. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *Proc. of the Int. Conf. on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, December 1996.

[2] M. Brodie and S. Ceri, "On Intelligent and Cooperative Information Systems: A Workshop Summary", Int. Journal of Intelligent and Cooperative Information Systems, Vol. 1, No:2, 1992

[3] U. Dayal, "Processing Queries over Generalization Hierarchies in a Multidatabase System", International Conf. on Very Large Data Bases, 1983.

[4] A. Dogac, et. al, "METU Object-Oriented DBMS", Demo Description, in Proc. of ACM Sigmod Intl. Conf. on Management of Data, Minneapolis, May 1994

[5] A. Dogac, et. al, "METU Interoperable Database System", ACM SIGMOD Record, 24(3), September, 1995.

[6] A. Dogac, et. al, "METU Object-Oriented DBMS Kernel", in Proc. of Intl. Conf on Database and Expert Systems Applications, London, September 1995 (Lecture Notes in Computer Science, Springer-Verlag, 1995).

[7] A. Dogac, et. al, "A Multidatabase System Implementation on CORBA", 6th Intl. Workshop on Research Issues in Data Engineering (RIDE-NDS '96), New Orleans, February 1996.

[8] A. Dogac, et. al, "METU Interoperable Database System", Demo Description, in Proc. of ACM Sigmod Intl. Conf. on Management of Data, Montreal, June 1996

[9] A. Dogac, C. Dengi and T. Ozsu, "Building Interoperable Database Management Systems on Distributed Object Management Platforms", Communications of the ACM (to appear).

[10] W. Du, R. Krishnamurthy and M-C. Shan, "Query Optimization in Heterogeneous DBMS", Proc. of the 18th Int. Conf. on VLDB, August 1992.

[11] W. Du, M-C Shan and U. Dayal, "Reducing Multidatabase Query Response Time by Tree Balancing", In ACM SIGMOD Intl. Conf. on Management of Data, 1995.

[12] C. Evrendilek, "Multidatabase Query Processing and Optimization", Ph.D. Thesis, Dept. of Computer Engineering, METU, November, 1996.

[13] C. Evrendilek, A. Dogac, S. Nural and F. Ozcan, "Query Decomposition, Optimization and Processing in Multidatabase Systems", in Proc. of Workshop on Next Generation Information Technologies and Systems, Naharia, Israel, June, 1995.

[14] C. Evrendilek, A. Dogac, S. Nural and F. Ozcan, "Multidatabase Query Optimization", Journal of Distributed and Parallel Databases, Vol.5, No.1, January, 1997.

[15] John E. Freund, *Modern Elementary Statistics*, Prentice-Hall, 1988.

[16] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, Jun Yang Optimizing Queries Across Diverse Data Sources. VLDB Conf. Athens, August 1997.

[17] W. Kim, *Modern Database Systems*, Adison-Wesley, 1995.

[18] G.M. Lohman, et. al, "Query Processing in R^*", Query Processing in Database Systems, Springer-Verlag, 1985.

[19] F. Ozcan, "Dynamic Query Optimization on a Distributed Object Management Platform", M.Sc. Thesis, Dept. of Computer Engineering, METU, February 1996.

[20] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, A. Dogac. "Dynamic Query Optimization on a Distributed Object Management Platform", in Proc. of the Intl. Conf. on Information and Knowledge Management (CIKM'96), Maryland, November 1996.

[21] C. Ozkan, A. Dogac, M. Altinel, "A Cost Model for Path Expressions in Object-Oriented Queries", Journal of Database Management, Vol.7, No.3, June 1996.

[22] Object Management Group, *The Common Object Services Specification*, Volume 1, OMG Document Number 94.1.1, January 1994.

[23] B. Salzberg. File Structures: An Analytical Approach, Prentice Hall Inc., 1988.

[24] S. Salza, G. Barone and T. Morzy, "Distributed Query Optimization in Loosely Coupled Multidatabase Systems", Proc. of Intl. Conf. on Database Theory, 1995.

Execution Reordering for Tertiary Memory Access

Sunita Sarawagi [§]
IBM Almaden Research Center
sunita@almaden.ibm.com

Abstract

In this article we investigate methods of dynamically reordering execution of a query plan. In existing systems, once a query is optimized, it is executed in a fixed order, with the result that data requests are made in a fixed order. Only limited forms of runtime reordering can be provided by low-level device managers. More aggressive reordering strategies are essential in scenarios where the latency of access to data objects varies widely and dynamically, as in tertiary devices, wide area distributed systems and broadcast disks in mobile computing. In this article we focus on methods of dynamically reordering different parts of a plan tree to match execution order to the optimal data fetch order. These techniques were developed in the context of a tertiary memory database system but are applicable to other cases as well. Our prototype implementation based on Postgres yields significant performance improvements with these techniques.

1 Introduction

We discuss reordering strategies for speeding up queries. In conventional systems, once a plan is optimized, execution of the plan proceeds in a fixed manner with the result that data pages are demanded in a fixed order that was optimized for a given data layout. The only form of dynamic reordering available during execution is through low-level I/O device schedulers or in some cases by asynchronous prefetching. The reordering that existing schedulers can achieve is limited to I/O requests from multiple users or to batch prefetching from processes doing asynchronous I/O. These existing schedulers may have greater opportunities for optimization if prefetching is done in larger batches; however, prefetching in large amounts can adversely affect caching performance [CFKL95]. In this work, we show that an effective way around this problem is to dynamically reorder execution to match the optimal data fetch order. If data in some part of the plan tree is "near by" now and will get "further away" later, it is advantageous to process the "near by" data first instead of waiting for the data "far away". This paper describes the implementation and evaluation of this simple idea in practical settings.

The key features of our framework for reordering execution are:

1. Relations are comprised of **chunks** that are available together.

Copyright 1997 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[§] Work done while the author was at the University of California, Berkeley

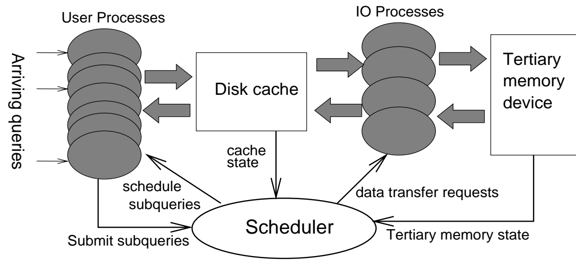

Figure 1: The architecture of the tertiary memory database system with the centralized scheduler

2. Each query plan tree is divided into parts (called **subqueries**[1] here) that can be executed independently in arbitrary order.

3. A **scheduling unit** collects subqueries from many users and decides at runtime the order in which they are executed.

4. A **reorderable executor** communicates with the scheduler to process the query plan in the order dictated by the scheduler.

Section 2 gives an overview of our architecture and reviews the design of the scheduling unit. Section 3 presents the design of the new executor which extracts the list of subqueries and executes them in arbitrary order. Section 4 presents related work and finally Section 5 makes concluding remarks. This article is an abridged version of the work presented in [Sar96, SS96, Sar95].

2 Architecture Overview

Figure 1 sketches the architecture of our tertiary memory database system introduced in [Sar95, Sar96]. We assume a process-per-user architecture where each user-session has a separate process serving its queries. An arriving query is first compiled by the user process. The user-process then extracts the list of subqueries from the query and submits the list to the scheduling unit. The scheduling unit is centralized in that it receives subqueries from all user processes and decides when they are executed. It maintains a set of I/O processes that transfer data between the disk cache and tertiary memory. As soon as all the data accessed by a subquery are available in the disk cache, the scheduler marks the subquery as "ready" for execution. The user processes contact the scheduler to collect ready subqueries and block until some subqueries are ready. After finishing execution of these ready subqueries, they send a notification to the scheduler, which can then decide to evict the cached data used by that subquery when desirable.

Each relation consists of a number of **fragments**. A fragment is the part of a relation that lies contiguously on a storage medium. The fragments of a relation correspond to the data **chunks** of our framework introduced in Section 1 (item 1). We further restrict the size of each fragment based on the size of the cache, the latency of access on tertiary memory, the data transfer rate and the number of concurrent users as discussed in [Sar96].

The scheduler (1) co-ordinates data movement between the disk cache and tertiary memory, (2) schedules query execution for each user process, and (3) decides what data is cached to or evicted from the disk cache. The scheduler makes these decisions based on system-wide information about pending subqueries from all users, the state of the disk cache and the tertiary memory, e.g, what platter is currently loaded. Details of how these decisions are made is given in [Sar96]. When deciding on the order of executing subqueries, the scheduler's objective is to maximize the overall system throughput. Hence, for a given user-process, one or more subqueries could be scheduled for execution together in an arbitrarily interleaved fashion with those of other users.

[1]The term *subqueries* is not to be confused with the SQL notion of subqueries. We use the subqueries to refer to parts of query.

3 Execution Engine

In this section we describe the design of the execution engine of the user processes. We first list the requirements needed by an execution engine to support reordering. Then we present the mechanism for supporting these requirements.

3.1 Specifications

1. *Submit to the scheduler a list of subqueries to be executed*
   The scheduler does not need to know all the details of the subquery, only which fragments are needed *together* in executing the subquery. Consider a nest-loop join between relations R and S where R has three fragments R_1, R_2 and R_3, and S has two fragments S_1 and S_2. The list of subqueries submitted to the scheduler, called the `SQ-list`, is:

   $$\{(R_1, S_1), (R_1, S_2), (R_2, S_1), (R_2, S_2), (R_3, S_1), (R_3, S_2)\}$$

2. *Execute subqueries out-of-order*
   There should be no **ordering constraints** between the subqueries submitted to the scheduler. For the two way join example above, the subquery (R_2, S_1), for instance, might be scheduled before the subqueries (R_1, S_1) and (R_1, S_2). Thus when the operators of a plan tree have precedence constraints on them, the subqueries must be submitted in multiple stages. For instance, for hash-join queries the inner fragments have to be fetched and the hash-table built, before processing any fragments of the outer relation.

3. *Execute multiple subqueries together*
   The scheduler could have more than one subquery ready for execution. We require that the executor be able to process multiple subqueries together. Executing one subquery at a time can lead to redundant computation for joins, since the scans on the outer relation cannot be shared across multiple fragments of the inner relation. For instance in the two-way join example, if S_1, S_2 and R_3 are cached, the scheduler will "ready" both the subqueries (R_3, S_1) and (R_3, S_2). The executor must be able to join R_3 with both S_1 and S_2 in one scan of R_3. Hence, although executing each subquery separately would allow for easy implementation, we must provide a means of executing multiple subqueries together.

3.2 Design

In this section, we describe the design of an executor that meets the specifications of Section 3.1. We base our discussion on the Postgres execution engine, in which each query plan is a tree of operators. All operators are implemented as iterators and support a simple start-next-end interface. Most relational database systems have analogous operator-based execution engines and can be extended similarly.

The optimized plan tree is then processed to extract the list of subqueries as discussed in Section 3.2.1. In Section 3.2.2 we discuss how execution proceeds out of order.

3.2.1 Extracting subquery lists

This proceeds in two phases: the fragmentation phase and the extraction phase.

Fragmentation phase: In this phase, each scan node on each base relation is replaced by a *combine node* that contains a list of scan nodes on the fragments of the base relation. The type of scan (sequential scan or index scan) on the fragments is the same as on the base relation. We assume that all the fragments of a relation have the same set of indices. For example, in Figure 2(a) we show the plan-tree of a 3-way join with three sequential scan nodes on base relations S, U and T. In Figure 2(b) we show the plan-tree after fragmentation.

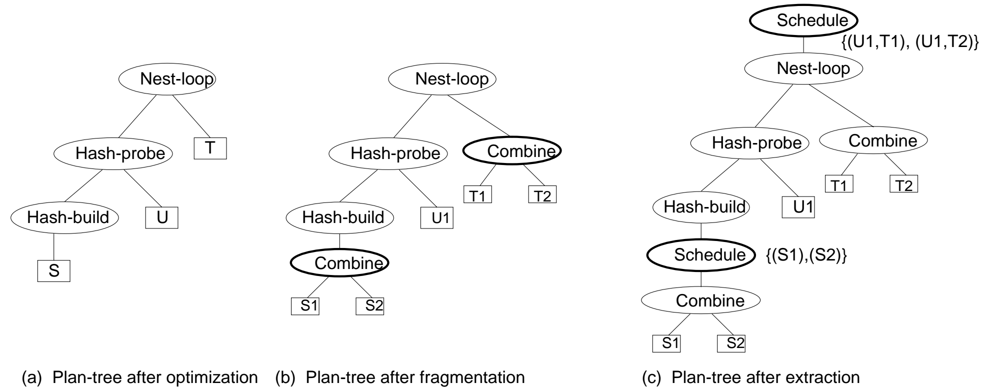

Figure 2: Example of a three-way join. The scan nodes are not shown for clarity; all of them are sequential.

Extraction phase: In this phase we extract the `SQ-lists` and insert special nodes called *schedule nodes* that are responsible for communicating with the scheduler and keeping synchronization information during execution. Each schedule node has an associated `SQ-list`. Because of precedence constraints between operators (Section 3.1(item 2)), we could have multiple `SQ-lists` in a plan-tree. For example, the plan-tree in Figure 2(b) has ordering constraints between the hash-build and hash-probe nodes. Hence, we added a schedule node before the hash-build node since the hash-build stage has to complete before starting processing on any nodes above it. We add a second schedule node at the top for the rest of plan-tree.

For inserting such schedule nodes and for constructing the `SQ-lists` we define a "Find-Sub-Query" call for each plan-tree node. This call returns the list of subqueries necessary to process the node. We give below the "Find-Sub-Query" routine for a few common nodes.

Find-Sub-Query for various nodes
Combine node:
  return list of fragments under the combine node
Hash-build, Aggregate or Sort node:
  `query-list` = Find-Sub-Query(subtree under node)
  if `query-list` non-empty
    add schedule node with `query-list` below node
  return `empty-list`
Join node:
  `listL` = Find-Sub-Query (left subtree)
  `listR` = Find-Sub-Query (right subtree)
  `query-list` = cross product of `listR` and `listL`
  If `listR` is empty, query-list = `listL`
  If `listL` is empty, query-list = `listR`
  return `query-list`

Finally, we add a schedule node above the root node of a plan tree when the "Find-Sub-Query" call on the root node returns a non-empty `query-list`. Also, note that the lists "`listL`" and "`listR`" could either be a flat list of fragments or a cartesian product obtained from another join node. For instance, when `listL` is $\{(R_1, S_1), (R_1, S_2)\}$ and `listR` is $\{T_1, T_2\}$, the cross product of these lists in the case of nested-loop join will yield a `query-list` $\{(R_1, S_1, T_1), (R_1, S_2, T_1), (R_1, S_1, T_2), (R_1, S_2, T_2)\}$.

In Figure 2, the "Find-Sub-Query" call on the Hash-build node adds a schedule node with the list $\{(S_1), (S_2)\}$

and returns the empty-list. The "Find-Sub-Query" call on the Hash-probe node returns $\{(U_1)\}$ and on the right branch of the Nest-loop node returns the list $\{(T_1),(T_2)\}$. The "Find-Sub-Query" call on the Nest-loop node returns the cross product $\{(U_1,T_1),(U_1,T_2)\}$ that is stored in a schedule node at the top of the tree.

3.2.2 Executing queries out-of-order

Our goal during the design of the execution engine was to follow the normal mode of processing as far as possible except for occasional communication between the execution engine and the scheduler for passing subquery information, collecting ready subqueries and notifying subquery completion. We show here how minor modifications in the scan nodes and the newly introduced schedule and combine nodes enable us to achieve this goal.

For efficiency reasons (discussed in Section 3.1,item 3) we want to execute all subqueries of a plan-tree from a single plan-tree instead of building a separate plan-tree for each subquery. This requires us to keep track of what subquery of the plan-tree is currently being executed. We do so by marking the scan nodes of the subqueries currently being executed as `available` and all other scan-nodes `suspended`. The plan-tree is then processed as usual: starting from the root of the plan tree, successive "next" calls are made to each node of the tree. When a "next" call is made on a combine node it submits the "next" call to a scan node underneath it that is marked `available`. A "next" call on a `suspended` scan node returns no tuple. Thus, only scan-nodes of currently scheduled subqueries participate in execution.

We next discuss how and when the schedule nodes are used for exchanging subquery information. Note that there could be multiple schedule nodes in the plan tree. It is critical to ensure proper interaction between these nodes to prevent deadlocks during execution by (1) submitting the `SQ-list` of a schedule node before the `SQ-list` of any schedule node above it and (2) processing subqueries of one schedule node and notifying the scheduler of their completion before submitting a `SQ-list` of some other schedule node. We want all these operations to be seamlessly integrated with the normal processing of the plan-tree. We achieve this goal by localizing all communication control into a "next" call of a schedule node consisting of the following steps:

1. Make a "next" call on the node underneath the schedule node to get the next tuple, t
2. If t is valid, return t
3. Else, if first time empty tuple returned
    - Submit the stored `SQ-list` to the scheduler
4. Else, /* stored list of subqueries already submitted */
    - Inform the scheduler of the completion of the last batch of scheduled subqueries, if any, and mark the scan nodes of those subqueries as `suspended`.
5. Make a blocking call to the scheduler to get the next collection of subqueries. Let Q be the collection of "ready" subqueries returned by the scheduler.
6. If Q is empty, then all subqueries have been executed, therefore return EOF.
7. Else, enable Q for execution by marking all the scan nodes appearing in Q as `available`.
8. Finally, make a "next" call on the node underneath and return the tuple obtained.

Note that, we submit the `SQ-list` of a schedule node only when no valid tuple is returned from the subtree underneath it. (steps 1 to 3 above). This ensures that the `SQ-list` of a schedule node is submitted only after all the `SQ-lists` in the schedule nodes underneath this node have been submitted and processed. If we had

submitted the `SQ-list` as soon as a schedule node is opened this ordering would not be preserved and could lead to deadlocks.

We will illustrate the above steps with the plan-tree in Figure 2. Initially, all the fragments are marked `suspended`. The first "next" call results in the submission of the list $\{(S_1), (S_2)\}$. Assume the scheduler makes (S_2) available first. As a result, the hash-build operation is partially completed. The scheduler is informed of the completion of subquery (S_2) (so it can uncache S_2 if needed) and a blocking request is made to get the next subquery. When the scheduler makes (S_1) ready, the rest of the hash-build operation is completed and the scheduler is informed of its completion. Next, the `SQ-list` $\{(U_1, T_1), (U_1, T_2)\}$ on the topmost schedule node is submitted. Assume both the subqueries are scheduled together. All data required by the plan-tree is now available. Hence, execution of the query is completed by pipelining the hash-probe and nest-loop operations.

The above scheme requires certain caution when scheduling *multiple* join subqueries together to avoid repetition of the following form: Consider the $R \bowtie S$ example of Section 3.1. Following the above scheme we first submit the `SQ-list` $\{(R_1, S_1), (R_1, S_2), (R_2, S_1), (R_2, S_2), (R_3, S_1), (R_3, S_2)\}$ to the scheduler. Suppose the scheduler next makes (R_1, S_1) ready, the executor finishes processing (R_1, S_1) and asks for the next set of ready subqueries. Suppose the next set of scheduled subqueries is $\{(R_1, S_2), (R_2, S_1), (R_2, S_2)\}$. To execute these three subqueries, scan-nodes of fragments R_1, R_2, S_1 and S_2 will be marked `available`, and the plan-tree will be processed as usual. But, by doing so, we have *repeated* the execution of subquery (R_1, S_1). To avoid such repetitions, the scheduler keeps track of subqueries already executed and uses this information for scheduling subqueries.

3.3 Handling Dependencies

Sometimes, it is not possible to know before execution what subqueries are needed because there is *dependency* between fragments. To determine what fragments are needed, some other fragments have to be processed. For example, with index scans, the data blocks required can be determined only after partial processing on the index trees. Similarly, with tuples pointing to large objects, the large objects to be fetched can be determined only after selecting the required tuples. To handle dependencies, two changes are needed:

1. First, we augment the plan-tree structure further with a special schedule node called the *resolve node*. The resolve node is added during the extraction phase immediately above the plan-tree node that introduces dependency between fragments. The resolve node, like the schedule node, contains a list of subqueries (`SQ-list`) that need to be executed first to resolve the dependencies. For instance, for an index scan, the resolve node is added immediately above the corresponding combine node and the `SQ-list` is the list of index trees on the indexed fragments. The `SQ-list` of the first schedule node above this resolve node cannot be established and hence is marked `unresolved`.

2. Next, we process nodes that introduce dependency in two stages: in the first stage a "ResolveDependency" call is made to compute the dependant list of subqueries and in the second stage after the subqueries are scheduled the rest of the node is processed. For instance, for the index scan node in the "ResolveDependency" stage the index tree is scanned and the list of matching TIDs is sorted to get the list of blocks that needs to be fetched. In the second stage, after these blocks are fetched we complete the rest of index scan.

With these modifications we can handle dependencies during execution as follows: when it is time to process a schedule node, s marked "unresolved", we make a "resolve-sub-query" call on the node below. The resolve-sub-query call behaves like the "find-sub-query" call for each node of the plan-tree until a resolve node is reached. The resolve node submits its stored `SQ-list` to the scheduler, and as subqueries from this list get scheduled, we make ResolveDependency calls on the node below to get the new `SQ-list`. The final `SQ-list` is then returned and execution proceeds as usual. We illustrate details of this method with the three normal cases of dependencies in relational engines: index scans, joins with index scans on inner relation and large object access in [SS96, Sar96]

4 Related Work

Our technique is reminiscent of the way multiple query optimizers combine queries with common subexpressions [SG90]. [MSD93] discusses policies for scheduling a batch of select and hash-join queries for sharing in-memory hash-tables. Queries are thus scheduled for execution in a data-driven manner the way we do. However, such optimizers typically schedule at whole relation level and do not consider reordering within a scan unlike our scheme.

Dynamic query optimization [Ant93, CG94] is another technique that involves plan tree modification at run-time. However, in contrast to our work, the emphasis in that area is on choosing dynamically from some fixed set of execution plans. Once the choice is made, execution proceeds in a fixed order.

Query scrambling [AFTU96], is a closely related and concurrently developed technique used for reordering operators in a plan tree to deal with unexpected delays in distributed systems. Apart from differences in the details of implementation of reordering, a significant difference in functionality is that, [AFTU96] does not consider reordering amongst different parts of the same relation unlike us. On the other hand, they allow the join order in a plan tree to be modified unlike us.

In object oriented databases, the navigational nature of queries can lead to bad I/O performance making it important to do prefetching [GK94] and batching [KGM91]. [KGM91] presents ways of modifying the plan-tree to replace object-at-a-time references with an assembly operator that collects multiple object references first and then reorders them to optimize I/O accesses. However the main difference between their scheme and ours is that, they cannot handle reordering across different operators of a plan-tree or across data references of different users.

Another concurrent work on modifying query plans to reorder I/O access on tape is reported in [YD96]. They propose a scheme for pre-executing functions that access large objects so as to allow I/O requests of different large objects in the same tuple stream and across multiple users to be reordered. However, they do not allow the order of processing tuples to be modified unlike in our case.

5 Discussion

In this paper, we have explored a simple, yet powerful, idea of reordering execution to tune to the optimal data fetch order. Existing methods of query execution provide but a limited flexibility of reordering data fetches during execution. Our proposal is based on the premise that in a multi-user environment when access latency of data varies widely, significant performance advantage can be gained by dynamically reordering execution.

We proposed a general framework for reordering all parts of the plan tree. For building a reorderable execution engine, we extended the plan tree data-structure with three new meta-nodes that are added in an extra phase between optimization and execution of the plan tree. These operators enable the executor to communicate and synchronize with the scheduler for ordering the execution of subqueries. Our changes are restricted only to these new operators and the extra phase and thus enable modular extension of existing execution engines. We extended the Postgres execution engine and used it for building a prototype of a tertiary memory database.

Our prototype, described in detail in [Sar96], yields almost a factor of three improvement over schemes that use prefetching and almost a factor of twenty improvement over schemes that do not, even for simple index scan queries. Further experiments demonstrate that either (1) when the platter switch and seek costs are high, or (2) when the cache is small and there is overlap between data accesses of concurrent queries, our reordering scheme will enable better scheduling of I/O requests and more effective reuse of cached data than conventional schemes. The overhead of reordering is measured to be small compared to the total query execution time (less than 1%). Thus, at least for tertiary memory databases the penalty of reordering is so negligible that reordering can almost always be used to advantage.

Our proposed general framework is applicable to other situations where tuning data to some external order

of arrival is important, e.g., a broadcast disk-based mobile computing client. The data chunks can be determined by the pages broadcast together. The size of the data chunks is important for limiting the overhead of reordering. For our prototype, typical overhead per subquery was 30 milliseconds. Hence, as long as the processing time per subquery is much larger than this reordering can be used profitably. The scheduling unit would be responsible for watching the broadcast data stream, caching relevant data when appropriate and scheduling ready subqueries for execution.

Future work in the area should consider the impact of execution reordering on query optimization: executing queries in parts invalidates some of the assumptions and cost functions used by the optimizer. In this paper, index scans posed one such scenario. Another topic for future work is providing support for cancelling submitted subqueries to the scheduler when a restrict or a join node yields an empty result.

References

[AFTU96] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling query plans to cope with unexpected delays. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, Dec 1996.

[Ant93] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proc. International Conference on Data Engineering*, pages 538–547, 1993.

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Application-controlled caching, prefetching and disk scheduling. Technical Report TR-493-95, Princeton University, 1995.

[CG94] R.L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. *Proc. ACM SIGMOD International Conference on Management of Data*, 23(2):150–160, 1994.

[GK94] C.A. Gerlhof and A. Kemper. A multi-threaded architecture for prefetching in object bases. In *Advances in database technology*, pages 351–364, March 1994.

[KGM91] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. *Proc. ACM SIGMOD International Conference on Management of Data*, 20(2):148–57, 1991.

[MSD93] M. Mehta, V. Soloviev, and D.J. Dewitt. Batch scheduling in parallel database systems. In *Proc. International Conference on Data Engineering*, pages 400–410, 1993.

[Sar95] S. Sarawagi. Query processing and caching in tertiary memory databases. In *Twenty first conference on Very Large Databases*, Sep 1995.

[Sar96] S. Sarawagi. *Query processing in tertiary memory databases*. PhD thesis, University of california at Berkeley, Dec 1996.

[SG90] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, 1990.

[SS96] S. Sarawagi and M. Stonebraker. Reordering query execution in tertiary memory databases. In *Twenty second conference on Very Large Databases*, Sep 1996.

[YD96] Y.Yu and D. Dewitt. Query pre-execution and batching in paradise. In *Proc. International Conference on Very Large Databases*, 1996.

ERICH NEUHOLD

Prof. Dr. Erich J. Neuhold
Director, GMD-IPSI
Dolivostrasse 15
D-64293 Darmstadt Germany
email: neuhold@darmstadt.gmd.de

Position Statement

The world of databases has been changing fundamentally by moving away from business transaction processing into areas like data mining, multimedia, and extensible databases. Quite a number of products are available already, but the spread of these technologies into database application industries has been rather slow. As one of the consequences, databases play a quite minor role in the WEB scenarios.

If elected, I would concentrate on furthering the acceptance of the new database technologies in old and new application fields. I am sure that we can build on our very successful two activities, the International Conference on Data Engineering and the Data Engineering Bulletin. By working together with the Steering Committee of ICDE and the Editorial Board of the Bulletin I would like to add features which should help especially smaller companies to rapidly move into utilizing the new database technologies.

However, this will not be enough. I would like to use an active WEB presence of our Technical Committee and state-of-the-art seminars and tutorials to help the above objective. Here, it will be important to closely work together with SIGMOD in the USA, but also with the many, sometimes quite large, national database communities. Many of the new database applications will be in fields where language, culture and local practices will require a more national approach than in the past. Using the Internet and the WEB will allow a much easier and faster collaboration between those organizations.

BETTY J. SALZBERG

Betty Salzberg
Professor, College of Computer Science
Northeastern University
Boston MA 02115 USA
salzberg@ ccs.neu.edu

Position Statement

Data engineering is a discipline which is closely tied to commercial products. For example, universities teach SQL as the only query language for relational database management systems because SQL is the standard commercial language, not because it is the most elegant, the most expressive or the easiest to use. In this sense, data engineering is different from more theoretical areas of computer science such as algorithms or programming languages.

Being tied to commercial products is both a blessing and a curse. It is a curse because many ideas which are intellectually interesting standing alone are not valuable to our community. This would include algorithms which are less efficient than those already in use or languages other than SQL for querying relational data, for example. On the other hand, it is a blessing because being forced to work within the constraints of practicality can produce a clarity of vision.

As chair of the Technical Committee on Data Engineering, I would consider it my mission to help enable academics to attain that clarity of vision by involving practitioners more closely in all the interchanges of information sponsored by the Committee. I would like to see more technical talks and papers from industry workers. Since their livelihood does not depend on publishing papers and giving talks, these talks and papers would have to be invited and vigorously encouraged. I would like to see this in the TKDE journal and the ICDE conference as well as the *Data Engineering Bulletin*. In addition, I would also like to organize some small workshops where industry technical people exchange information with academics. I am currently proposing such a workshop to the National Science Foundation.

David Lomet recently asked you to send suggestions and ideas to the Data Engineering Bulletin and I would also like to urge you to do so. If elected, I promise to pick up your ideas for further consideration and implementation in a possibly international framework. But the first step to have an active Executive Committee is to become active yourself and the first step of that is to take part in the upcoming election.

Biography

Erich J. Neuhold is currently professor at the Darmstadt University of Technology and director of the GMD Institute for Integrated Publication and Information Systems (IPSI). He has spent more than 10 years in industry working for IBM and HP, and the remainder of his career, as a professor at various European and US universities.

He is currently European coordinator of our Executive Committee and has been active for many years with ICDE including PC chair, general chair, and member of the ICDE steering committee. He has held similar positions with VLDB, the VLDB endowment, and many other national and international conferences. He has also been involved with various IFIP activities and is currently chairman of the IFIP working group on databases which organizes the Data Semantics and Visual Database working conference series.

He is a member of 9 editorial boards of professional journals, has published or edited about 15 books, coordinated special journal issues, and published about 100 papers.

In this way, academics will understand in a deeper way what is already available in commercial products and what stumbling blocks still exist. They will be able to do research which is both intellectually interesting and valuable to our entire community. Practitioners may benefit from the ideas in the new research and university students may be able to understand better the principles behind current practice.

Biography

Betty Salzberg received her Ph.D. in mathematics from the University of Michigan. She is now a Professor in the College of Computer Science at Northeastern University. Her areas of interest include access methods and online reorganization. Her work in these areas has been funded by the National Science Foundation for the past nine years.

Salzberg´s access method algorithms include spatial indexing, temporal indexing and concurrency and recovery in tree indexes. She wrote the chapter on access methods in the recent *CRC Handbook on Computer Science.* Much of Salzberg´s work has been in collaboration with Dr. David Lomet, Dr. Georgios Evangelidis and Dr. Chendong Zou. Salzberg is the author of two textbooks, *File Structures: An Analytic Approach* and *An Introduction to Database Design*. She is the coauthor with Prof. Vassilis Tsotras of a survey article on temporal indexing to appear in *Computing Surveys*.

Salzberg has served numerous times on the program committees of the SIGMOD, VLDB and ICDE conferences. She is currently an associate editor of the *Data Engineering Bulletin*.

ELECTION BALLOT

TECHNICAL COMMITTEE ON
DATA ENGINEERING

The Technical Committee on Data Engineering (TCDE) is holding an election for Chair. The term of Rakesh Agrawal has expired. Please email or fax in your vote.

BALLOT FOR ELECTION OF CHAIR
Term: (January, 1998 - December, 1999)

Please vote for one candidate.

☐ Erich Neuhold

☐ Betty Salzberg

☐ ____________
(write in)

Your Signature:____

Your Name:____

IEEE CS Membership No.:________________________________
(Note: You must provide your member number. Only TCDE members who are Computer Society members are eligible to vote.)

Please fax or email* the ballot to arrive by November 25, 1997 to:

twoods@computer.org
Fax: +1-202-728-0884

*Email may be sent by filling out a form at http://lsdis.cs.uga.edu/activities
or mail to
IEEE Computer Society
Attn: Tracy Woods
1730 Massachusetts Avenue, NW
Washington, DC 20036-1992

RETURN BY November 25, 1997

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

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