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

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.

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 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

[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.

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.

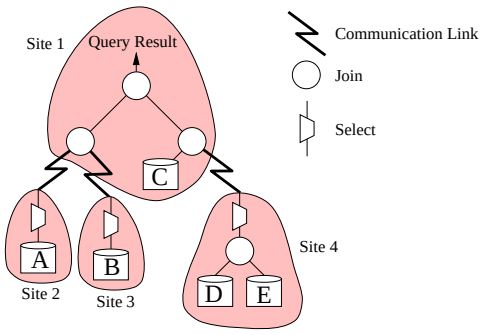

Figure 1: Example of a Complex Query Tree

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 run-time, 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 un-stall. 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:

$$\text{Qor} = \textit{select} * \textit{from } (Q1 \bowtie B) \bowtie Q2 \textit{ where } c_3$$

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.