University of Maryland Technical Report CS-TR-3811 and UMIACS-TR-97-54

Dynamic Query Operator Scheduling for Wide-Area Remote Access*

Laurent Amsaleg
IRISA & INRIA
Laurent.Amsaleg@irisa.fr

Michael J. Franklin
University of Maryland
franklin@cs.umd.edu

Anthony Tomasic
INRIA
Anthony.Tomasic@inria.fr

October 22, 1997

Abstract

Distributed databases operating over wide-area networks such as the Internet, must deal with the unpredictable nature of the performance of communication. The response times of accessing remote sources can vary widely due to network congestion, link failure, and other problems. In such an unpredictable environment, the traditional iterator-based query execution model performs poorly. We have developed a class of methods, called *query scrambling*, for dealing explicitly with the problem of unpredictable response times. Query scrambling dynamically modifies query execution plans on-the-fly in reaction to unexpected delays in data access. In this paper we focus on the dynamic scheduling of query operators in the context of query scrambling. We explore various choices for dynamic scheduling and examine, through a detailed simulation, the effects of these choices. Our experimental environment considers pipelined and non-pipelined join processing in a client with multiple remote data sources and delayed or possibly bursty arrivals of data. Our performance results show that scrambling rescheduling is effective in hiding the impact of delays on query response time for a number of different delay scenarios.

Keywords: distributed query processing, mediators, iterator execution model, performance analysis, query scrambling, dynamic query optimization

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. At present, such access is typically obtained through non-database facilities such as the World-Wide-Web. Advances in distributed heterogeneous databases (e.g., [Kim95, SAD+95, BE96, TRV96]) and other non-traditional approaches (e.g., WebSQL [MMM96]), however, aim to make the Internet a viable and important platform for distributed database technology.

The Internet environment presents many interesting problems for database systems. In addition to the issues of data models, resource discovery, and heterogeneity addressed by the work in the areas cited above, a major challenge that must be addressed for wide-area distributed information systems is that of *response-time unpredictability*. Data access over wide-area networks involves a large number of remote data sources, intermediate sites, and communications links, all of which are vulnerable to congestion and failures. Such problems can introduce significant and unpredictable *delays* in the access of information from remote sources.

Current distributed query processing technology performs poorly in the wide-area environment because unexpected delays encountered during a query execution *directly* increase the query response time. Query

*This work was partially supported by the NSF under grant IRI-94-09575, by the Office of Naval Research under contract number N66001-97-C8539 (DARPA order number F475), by Bellcore, and by an IBM Shared University Research award. Laurent Amsaleg contributed to this research while he was with the University of Maryland.

execution plans are typically generated statically, based on a set of assumptions about the costs of performing various operations and the costs of obtaining data. The execution of a statically optimized query plan is likely to be sub-optimal in the presence of unexpected response time problems that arise during the query *run-time*. In the worst case, a query execution may be blocked for an arbitrarily long time if needed data fail to arrive from remote data sources. The apparent randomness of such delays in the wide-area environment makes planning for them during query optimization nearly impossible.

To address the issue of unpredictable delays in the wide-area environment, we have developed a dynamic approach to query execution, called *query scrambling*. Query scrambling reacts to unexpected delays by *on-the-fly* rescheduling the operations of a query during its execution. Query scrambling attempts to hide delays encountered when obtaining data from remote sources by performing other useful work, such as transferring other needed data or performing query operations, such as joins, that would normally be scheduled for a later point in the execution. Query scrambling can be effective at hiding significant amounts of delay; in the best case, it can hide *all* of the delay experienced during a query execution. That is, a query can execute in the presence of certain delays with little or no response time penalty observable to the user.

1.1 Coping With Bursty Arrival

In a previous paper [AFTU96], we identified three types of delay that can arise when requesting data from remote sources:

Initial Delay There is an unexpected delay in the arrival of the *first* tuple from a particular remote source. This type of delay typically appears when there is difficulty connecting to a remote source, due to a failure or congestion at that source or along the path between the source and the destination.

Slow Delivery Data is arriving at a regular rate, but this rate is much slower than the expected rate. This problem can result, for example, from network congestion, resource contention at the remote source, or because a different (slower) communication path is being used (e.g., due to a network link failure).

Bursty Arrival Data is arriving at an unpredictable rate, typically with bursts of data followed by long periods of no arrivals. This problem can arise from fluctuating resource demands and the lack of a global scheduling mechanism in the wide-area environment.

The algorithm presented in [AFTU96] focused on the problem of Initial Delay. As such, it was assumed that once data started to arrive from a remote source, the remaining data from that source would arrive in an uninterrupted fashion. This assumption facilitated the development and study of an initial approach but limited the applicability of the resulting algorithm, as wide-area data access seldom fails in such a well-behaved manner. In this article, we extend the scope of query scrambling by investigating approaches to dynamically rescheduling query operations in the presence of the additional problem of bursty arrivals.

Bursty arrivals are more difficult to manage than initial delays for several reasons. First, the run-time system must constantly monitor the arrival of data from remote sources and must be able to react to delays that arise at any time. Such continuous monitoring of remote sources is not necessary in the initial delay environment. Second, due to the unpredictable nature of bursty arrivals, care must be taken to avoid initiating overly-expensive scrambling actions for short, transient delays, while remaining reactive enough to initiate scrambling without undue hesitation in situations where there is significant delay. Given the difficulty of predicting the future short-term behavior of remote access, scrambling for a bursty environment must be implemented such that it can be initiated, halted, and restarted in a lightweight manner.

1.2 A Reactive Approach

Query scrambling shares some common goals with other approaches to dynamic query processing. In general, methods that attack poor run-time performance for queries fall into two broad categories: *proactive* and *reactive*. *Proactive* methods (e.g., [ACPS96, CG94, SAL+96]) attempt at compile-time to predict the behavior of query execution and plan ahead for possible contingencies. These approaches use a form of late binding in order to postpone making certain execution choices until the state of the system can be assessed at run-time. Typically the binding is done immediately prior to executing the compiled plan, and remains fixed for the entire execution.

Reactive methods (e.g., [TTC+90, Ant93, ONK+96]) monitor the behavior of the run-time system during query execution. When a significant event is detected, the run-time system reacts to the event. Query scrambling is a reactive approach — the query execution is changed on-the-fly in response to run-time events. While other reactive approaches have been aimed towards adjusting to errors in query optimizer estimates (e.g., selectivities, cardinalities, etc.), query scrambling is focused on adjusting to the problems that arise due to the time-varying performance of loosely-coupled data sources in a wide-area network. Related work is discussed in more detail in Section 7.

One basic technique used by query scrambling is to change the scheduling of operators in a query plan if a delay is detected while accessing data from a remote site. Such rescheduling permits delays from different remote sources to overlap with each other and to overlap with useful work performed by the query processor. In order to implement this rescheduling, the run-time system must sometimes introduce additional materializations of intermediate results and base data into the query execution plan. For this and other reasons, query scrambling may increase the total *cost* of query execution in terms of network contention, memory usage, and/or disk I/O.

1.3 Overview of the Article

Because operator rescheduling introduces both benefits and costs, it must be regulated in an effective way. Thus, the key questions for implementing scrambling rescheduling are: 1) when should scrambling start; 2) what should be rescheduled; and 3) when should scrambling stop. We examine several sets of *policies* to control scrambling rescheduling, and we describe the architecture of a run-time scheduler that is capable of implementing these policies. We then use a detailed simulation of a run-time system based on the iterator query processing model [Gra93] in order to examine the tradeoffs of the various scrambling policies for both pipelined and non-pipelined execution.

In this article, we focus on query processing using a data-shipping or hybrid-shipping approach [FJK96], where data is ultimately collected from remote sources and integrated at the query source. This approach models remote data access and is also typical of mediated database systems that integrate data from distributed, heterogeneous sources, (e.g., [TRV96]). In this work, the remote sources are treated as black boxes, regardless of whether they provide raw data or the answers to subqueries. Only the query processing that is performed at the query source is subject to scrambling. Our results show that scrambling, if done correctly, can produce dramatic response time savings under a wide range of delay scenarios. It can in some cases, reduce the slowdown observed due to random delays by a factor proportional to the number of bursty remote sources. It can also, in some cases completely hide the delay from the user.

In summary, unpredictable behavior of remote sources during query execution is a problem that database technology *must* address if it will ever be successful on the Internet. We have investigated initial results for a new class of methods, query scrambling, that attempts to address this problem. This article describes the following contributions:

1. An examination of the weaknesses of the iterator model in this environment,
2. An architecture, which extends the iterator model, of a scrambling rescheduling run-time system,
3. Several policies for controlling the key implementation aspects of scrambling rescheduling,
4. Extensive simulation results that document the various performance trade-offs of the policies, and
5. Evidence that scrambling rescheduling is effective for a broad class of workloads in a bursty data arrival environment.

The article is organized as follows. Section 2 describes the basic trade-offs for query scrambling to cope with bursty arrivals. Section 3 provides a detailed model and architecture of a run-time scheduler for implementing scrambling rescheduling. Section 4 describes the policies which control rescheduling. Section 5 describes the experimental framework and Section 6 describes the experimental results for the non-pipelined and pipelined cases. Section 7 describes related work. Section 8 concludes the article.

2 Query Scrambling Overview

In this section we first discuss the behavior of a traditional iterator based run-time system and its behavior in the bursty environment. We then describe how scrambling can be applied to such a run-time system in order cope with unexpected delays. Finally, we discuss the basic tradeoffs and design decisions that arise in the development of a scrambling algorithm.

2.1 Query Scrambling for Iterator-Based Execution Engines

Rather than relying on the operating system, most database systems provide their own execution engine, which performs scheduling and memory management for the operators of compiled query plans. The *iterator* model is one way to structure such an execution engine [Gra93]. In this model, each node of the query tree is an iterator. Iterators support three different calls: *open()* to prepare an operator for producing data; *next()* to produce a single tuple, and *close()* to perform final housekeeping. To start the execution of a query, the DBMS initiates an *open()* call on the root operator of the query tree, and this call iteratively propagates down the query tree.

A key attribute of the iterator approach is that the scheduling of the query operators is, in some sense, compiled into the query tree itself. The scheduling of the operators in the tree is determined by the way in which operators make *open(),next()*, and *close()* calls on their children operators. The data flow among nodes in this model is demand-driven. A child node passes a tuple to its parent node in response to a *next()* call from the parent. As such, iterator-based plans allow for a natural form of *pipelining*. Each time an operator needs data, it calls its child operator(s) and waits until the requested data is delivered. The producer-consumer relationship allows the operators to work as co-routines, and avoids the need for storage of intermediate results, as long as the child operator produces tuples at about the same rate or slower than they can be consumed by its parent operator. This scheduling dependency can be avoided, however, if the child operator first *materializes* its result (e.g., as part of *open()* processing) either in memory or to disk. After materialization, the child can then provide tuples to the parent operator in the typical one-at-a-time fashion in response to *next()* requests. A completely *non-pipelined* schedule can be constructed by introducing materialization between each pair of operators in the tree.

This simple, static scheduling approach works well when the response times of operators and data sources can be predicted with some accuracy. When processing queries with data from remote sources, however, unpredictable delays in obtaining that data can arise. The effect of such unexpected delays on a precompiled schedule can be severe. When a remote source blocks, all of its *ancestors* in the query tree will also block. In addition to delaying the initiation of operators that are scheduled to execute later in the plan, such blocking can also block other operators that are already executing. For example, if a binary operator (e.g., a join) becomes blocked because one of its children blocks, then it will stop requesting tuples from its other child, thereby inducing blocking on the subtree rooted at that child as well. This blocking can propagate *down the subtree* to the leaves of the tree, unless a materialization (which breaks the producer-consumer dependency) is encountered.[1] With a static schedule, progress on the query can, in some cases, grind to a halt even if only a single data source becomes delayed.

In this article, query scrambling applies dynamic scheduling to query execution in order to avoid the problems caused by unexpected delays. It depends on two basic techniques: *rescheduling* and *materialization*. Simply stated, when a delay in obtaining data from a remote source is detected, scrambling changes the scheduling of operators in the query tree in order to allow other portions of the plan to execute. To perform this rescheduling, scrambling introduces any materializations that are required to allow the re-scheduled operators to run. Materializations can be added to the plan by placing a *materialization* operator between

[1] Note that this blocking phenomenon arises even if operators are ones that support intra-operator parallelism such the *exchange* operator of Volcano [CG94].

the re-scheduled operator and its parent.[2] A materialization operator is a unary operator, which when opened, obtains the entire input from its child and places it in storage (typically disk, unless there is sufficient memory). The materialization operator provides tuples in response to *next()* requests from its parent operator when the parent is eventually able to execute.

As stated in the introduction, there are three key policy questions for the implementation of a scrambling run-time system: (1) when to start scrambling, (2) what to scramble, and (3) when to stop scrambling. In the following three sections we describe the options and the basic tradeoffs that arise for each of these.

2.2 Initiating Scrambling

A fundamental principle of our approach to Query Scrambling is that the normal scheduling of a query execution should proceed unperturbed in the absence of unexpected delays. The assumption is that the execution plan generated by the optimizer is in fact, an efficient plan, and that re-scheduling and materialization can result in additional memory, disk I/O, and other costs. Thus, the original plan should be tampered with only if an unexpected problem arises during the execution.

In order to determine when a delay has occurred, the system associates a *timer* with each operator that directly accesses data from a remote site. This timer is started when the operator begins waiting for a chunk (i.e., a page or packet) of data to arrive from the remote site, and is reset when the data arrives. If the timer goes off before the data arrives, then the scrambling mechanism is informed that a significant delay has occurred.

Given such a timer mechanism, the main policy question is to determine at which point there are sufficient problems to warrant the initiation of re-scheduling. There is a knob that can be used to fine-tune such a policy. The *timeout-value* is the value with which the timer is initialized when an operator enters a waiting state. The length of this value determines how long the operator waits before a *timeout* alarm is raised.

The *timeout-value* limits the degree of response time variance that will be tolerated for any remote source. This knob allows the sensitivity of the scrambling policy to be adjusted across a range from *aggressive* (i.e., low settings for the knob) to *tolerant* (i.e., high setting). The tradeoffs between these two extremes are fairly straightforward: A *tolerant* policy runs the risk of allowing too much delay to accumulate before reacting, while an *aggressive* policy can potentially waste resources in an effort to solve non-existent (or minor) problems. The decisions covered in the next two sections, however, can help limit the extent of the damage caused by an overly aggressive approach.

2.3 What to Scramble

Once scrambling has been initiated, the next decision to be made is the extent of the scrambling action to be performed. As stated previously, scrambling involves the rescheduling of operations in the execution plan. There are two types of policy decisions that must be made with respect to the extent of scrambling: i) where in the tree to initiate scrambling; and ii) how many scrambling operations should be initiated.

[2] This notion of a materialization operator is not related to the operator for path expressions described in [BMG93].

For the first question, we consider two options: i) early initiation of a *non-leaf* operator in the plan; and ii) early retrieval of data from a *remote source*. The first case, initiating a non-leaf operator, requires the scrambling system to artificially call *open()* on that operator. The *open()* has the usual effect of initiating the sub-tree of the query rooted at that operator. It is relatively simple to execute a *non-pipelined* operator out-of-turn (i.e., before its parent operator) because such an operator simply writes its result to a temporary file (or to an allocated area in memory). On the other hand, rescheduling *pipelined* operators is more difficult; it requires the introduction of a materialization operator as a *surrogate parent*, in order to temporarily store the result of the operator. A surrogate parent is also needed in the case of early retrieval of data from a remote source. In that case, a materialization operator is inserted in the tree to pull tuples from the remote source and store them locally at the query execution site.

The tradeoffs between these two choices are as follows: Starting a non-leaf operator allows the entire subtree rooted at that operator to be initiated at the cost of at most, a single additional materialization. The downside of this approach is that sufficient memory must be allocated to allow the subtree to execute. In contrast, early retrieval from a remote source requires very little memory (e.g., one or two pages, for staging tuples to disk), however, an additional materialization is required for every remote source opened in this way.

The second decision that must be made is how many scrambling operations should be initiated. The fundamental tradeoff here is as follows. The more operations that are initiated, the more remote sources can be accessed in parallel, and hence, the greater the potential for overlapping the delays that might arise from those remote sources.[3] There are, however, significant dangers in starting too many operators. First, if care is not taken, the data arriving from multiple sources can cause contention in the network or at the query execution site. On the network, contention can result in the invocation of congestion avoidance mechanisms, which can force sources to send data at a low rate. At the query execution site, thrashing can arise if the speed of materializations to disk cannot keep up with the rate at which the remote sources are delivering data. These problems can be mitigated, to some extent, if the query execution site controls the arrival of data from remote sources. Such control can be achieved using a *page-at-a-time* protocol (as opposed to a *streaming* protocol) between the query execution site and the remote sources.

Another problem that can arise from initiating too many scrambling operations is the randomization of disk access. When multiple relations are placed on the disk of the query execution site, access to those relations may interfere with other disk I/O performed by the query. For example, in the case of a non-pipelined join, accessing the input relations from disk may interfere with the writing of the join result to disk, thereby turning both processes into random rather than sequential I/O. Such interference can slow disk access substantially. Note that this latter problem can arise regardless of whether a streaming or page-at-a-time protocol is used to obtain data from remote sources.

[3] In general, if n remote sources are subject to significant, independent delays, then by accessing those sources in parallel, scrambling has the potential to improve performance (over not scrambling) by as much as n times.

2.4 Stopping Scrambling

The third key decision for scrambling is that of when to stop scrambled operations once they have been initiated. There are two basic choices here. One option is to simply *suspend* all scrambled operations when the remote source that triggered scrambling resumes sending data. The other option is to *ignore* the status of the blocked remote source, and continue scrambling. Perhaps the most intuitive approach is to suspend scrambling and resume normal processing as soon as a blocked operator becomes unblocked. Since scrambling is a reaction to an unanticipated event, it makes sense to resume the original plan as soon as possible. In addition, scrambling has the potential to add costs to the execution of the query, so returning to the original schedule can help avoid such costs.

In cases where a remote source temporarily experiences delays but then performs smoothly, the approach of returning to the original plan is likely to work well. In other cases, however, going back too soon can carry its own costs. Recall that some scrambled operators (e.g., those higher in the query tree) may consume considerable amounts of memory. If the suspension of scrambling causes the scrambled operators to be swapped out then it is possible to encounter a thrashing condition if the remote source repeatedly delays and resumes. On the other hand, not swapping the scrambled operators out could result in a significant waste of memory and could hurt performance. Thus, for very unreliable remote sources, it could be beneficial to continue scrambling, even if the remote source resumes. A useful option in this case might be to materialize the delayed source in the background while continuing to complete the scrambling operations. Materializing an operator that was started normally, however, would require additional mechanism beyond what has been described above.

2.5 Discussion

The above sections described the main decisions that must be addressed when designing a query scrambling policy for the bursty environment. These decisions and their possible settings are summarized in Table 1. The settings allow the scrambling policy to be adjusted between tolerant and aggressive approaches towards dealing with delays. In general, tolerant policies favor sticking to the original query plan wherever possible, while aggressive policies are more willing to commit resources in order to hide potential delay. As stated above, it is possible to implement scrambling in a way that can reduce the potential for problems. For example, using a page-at-a-time protocol rather than a streaming one for obtaining data from remote sources can reduce the potential for network and local disk congestion.

In this article, query execution tree shape is fixed during execution, i.e. join ordering is not changed, and the physical network topology is also fixed. Both of these assumptions impact the performance of scrambling.

Consider the impact of tree shape on scrambling. If the first (left-most) remote source, say A, in the query execution order, has a long delay, then scrambling will perform very well. The rest of the query will execute during the time that A is delayed, effectively overlapping the delay of A with all other delays and work. However, suppose the *last* remote source, say Z, is delayed. Scrambling will be ineffective, since there is no work after Z and thus no work to scramble. In general, delays which appear early in query execution

Decision	**Values**	
	(tolerant)	(aggressive)
Start *timer-value*	high	low
Which Operators	*remote source*	*non-leaf*
How Many Operators	few	many
Stop	*suspend*	*ignore*

Table 1: Summary of Scrambling Options

order have much more impact than delays which appear late.[4]

Consider the impact of physical network topology. If a network delay affects only a single remote source, scrambling will perform as if the delay was due to the remote source itself. However, if a network delay affects all remote sources equally (e.g. a delay in the network link between the client and the local Internet router of the client), scrambling will be ineffective, because all remote sources are equally delayed and thus no work can be overlapped.

3 Architecture

In this section we describe the architecture of a scrambling run-time system. We first extend the iterator model with a scheduler. We then describe how materialization operators are inserted into the query tree.

3.1 The Query Scrambling Engine

We extend an iterator run-time system such that each operator has an independent internal process state. A *scheduler* dictates the state of each operator. Operators can be suspended, resumed, or terminated just like operating system threads. An operator can be in five possible states. Among these five states, six transitions are possible. Operator states and transitions are showed in Figure 1.

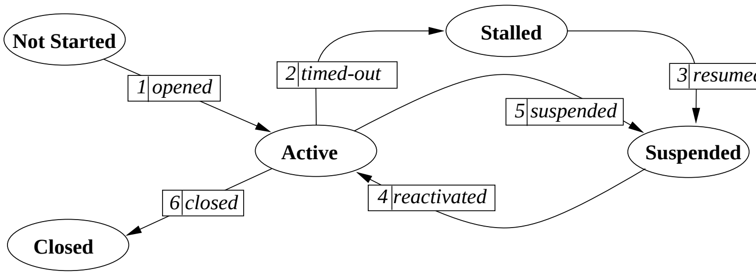


Figure 1: State Diagram for Query Operators

These states are:

- **Not Started**. State of an operator before being opened.

[4] Thus, a query optimizer for a run-time system that supports scrambling may favor query execution plans where historically unreliable remote sources appear early in the plan.

- **Active**. State of the operators that can be scheduled for execution. The actual order in which **Active** operators are scheduled is identical to the one that would normally be produced by the iterator model under traditional scheduling.
- **Suspended**. State of an operator explicitly suspended by the query scrambling scheduler.
- **Stalled**. State of an operator stalled due to the unavailability of the requested data.
- **Closed**. State of an operator once it has produced all its possible results.

The query scrambling scheduler moves one or more operators from one state to another via a transition in response to an external event. Three possible external events are defined:

- Time-Out. When the timer embedded in an operator goes-off, the operator informs the scheduler of the time-out. In turn, the scheduler then knows this operator can not be run.
- Resume. When pending data eventually arrive at the query execution site the scheduler determines the operator for which the data is intended. The scheduler then knows this operator can potentially be run again.
- End of Stream. An operator that produced all its possible results tells the scheduler it has reached the end of stream. Such an operator goes out of the scope of scrambling.

The reactions of the query scrambling scheduler to the occurrence of these events can be easily expressed in terms of transitions between states for the operators concerned by the events. The transitions between the states are:

1. *opened.* Every time an operator opens, the scheduler moves this operator from **Not Started** to **Active**.
2. *timed-out.* The scheduler moves an operator from **Active** to **Stalled** when the operator times-out (first external event). The scheduler also forces the ancestors of the stalled operator to go through this transition as well, indicating that a whole branch of the query tree is blocked and can not run.
3. *resumed.* When the pending data eventually arrives (second external event) the scheduler moves the corresponding operator, as well as its ancestors, from **Stalled** to **Suspended** indicating that they can potentially be run again.
4. *reactivated.* The scheduler moves an operator from **Suspended** to **Active** when it decides to reactivate it. Every time an operator is moved through the transitions *timed-out* or *resumed*, the query scrambling scheduler checks to see if one (or more) suspended operations need to be re-activated. For example, if no operators are **Active** because they are all timed-out, then the scheduler will try to reactivate the scrambling of **Suspended** operators.

5. *suspended.* The scheduler moves **Active** operators to the **Suspended** state when it decides to temporarily suspend their execution. This happens, for example, when the regulation mechanism of query scrambling decides to halt all materializations because the problem that triggered scrambling is resolved. Later, suspended materializations can be reactivated, for example in response to the time-out of one active operator.

6. *closed.* When an operator completes (end of stream, third external event), it closes and the scheduler moves it to the **Closed** state.

3.2 Modifying the Query Tree

After it has chosen an operator to reschedule, the query scrambling scheduler analyses the query tree to determine if it has to introduce a materialization operator as a *surrogate parent* to allow this operator to run. If not, then the scheduler simply starts a thread that opens the operator. In contrast, if a surrogate parent is required, then the scheduler creates a new materialization operator and inserts it between the rescheduled operator and its parent. Patching a query tree is fairly simple with iterators, since they interact through well defined, implementation independent, interfaces. As such, neither the parent nor the child operator needs to be aware of the patch.

Once the surrogate parent is placed in the tree, the scheduler opens it. After calling *open()* on its child, the materialization operator continuously calls *next()* and materializes the received tuples to disk. The child operator is closed when it produces its last tuple. At this point the materialization is *complete*.

Eventually, the original parent of the rescheduled operator will be scheduled to execute. Due to the patching of the query tree, when it calls *open()* on its child, it actually *re-opens* the materialization operator. In response to *next()* calls, the materialization operator returns the tuples that it previously materialized. If the materialization was *complete* then its child operator need never be called. On the other hand, if the materialization was incomplete, then once its supply of materialized tuples is exhausted, it simply *passes* any subsequent *next()* calls to its child, and passes each tuple obtained in this manner back to its parent.

4 Policies

We now present the scheduling and rescheduling policies that we study in the subsequent sections. Two of these policies are static while the two others are reactive. The static policies do not change the scheduling of operators even when delays are encountered (in fact, they are not aware that a delay has occurred). In contrast, the reactive policies change the original schedule once a delay is experienced. The two reactive policies differ by the operators that they are allowed to reschedule. Because of the memory problems that can arise when rescheduling subtrees, we focus on policies that have very manageable memory requirements. In particular, one policy only materializes relations obtained directly from remote sources and the other policy is able to in addition, reschedule a single join operator at a time. The four policies are:

Normal Iterator Execution (*ITR*). The first policy, which we use as a baseline, is a static, iterator-based execution as described in Section 2.1.

Materialize Always (*MA*). *MA* is also a static policy, but differs from *ITR* in that it it *immediately* initiates the materialization of all data sources at query startup time. When the query starts its execution, this policy inserts in the query tree materialization operators for all relations that are to be obtained from the remote sources. Once those operators have been inserted in the tree, the policy spawns threads to open them. Materializations continuously pull-over remote data and write this data on the local disk. In parallel to those materializations, the query continues its execution. When an operator (a join for example) needs data from a relation that is currently materialized, this join stops this particular materialization (others remain active), consumes the local data and requests the rest of this relation (if any) from the remote server. Of course, since *MA* is a static policy, it made aware of any delays that may be encountered during a query execution, but rather, the effected operators simply block. *MA* is used to show the impact of parallel fetching from remote sources in the absence of a reactive policy.

Reactive Materialize (*RM*). The simplest of the two reactive policies we study is *RM*. In the absence of delay, *RM* behaves identically to the static *ITR* policy. As soon as the query experiences a delay, it switches to a mode similar to *MA*, that is, all data sources are opened and their data materialized in parallel. Any delay experienced by on-going materializations do not trigger any special action. When the data source that caused this opening resumes, on-going materializations are *suspended* and the query returns to standard execution. If another delay is experienced, the suspended materializations are resumed, and they continue to bring data in parallel. The choice of suspending rescheduled operators was made because materializations consume little memory.

Reactive Materialize and Join (*RMJ*). This policy has the same basic behavior as *RM*, except that it also is able to reschedule the execution of single join at a time. As a result, this policy assumes that there is enough memory available to support the execution of this join. Because there is only enough memory for a single rescheduled join to execute, such a join is not initiated until both of its input relations have been fully materialized to the local disk; in this way it is known that the rescheduled join will not be blocked by any delayed data. Joins are elected for execution on a first-come first-served basis. Materialization of joins can run concurrently with on-going materializations of base relations. As in the previous policy, all on-going materializations (i.e., of base relations and/or joins) are suspended if delayed data begins to arrive. We chose to study *RMJ* because it allows for potentially more work to be done by scrambling rescheduling, but it also has very manageable memory requirements.

5 Experimental Framework

In this section we first describe the simulation environment used to evaluate several different policies for scrambling queries. We then present the workload used to perform these experiments.

Parameter	Value	Description
NumSites	9	number of sites
Mips	30	CPU speed (10^6 instr/sec)
NumDisks	1	number of disks per site
DskPageSize	4096	size of a disk page (bytes)
RequestSize	40	size of a data request (bytes)
TransferSize	8192	size of a data transfer (bytes)
Compare	4	instr. to apply a predicate
HashInst	25	instr. to hash a tuple
Move	2	instr. to copy 4 bytes
Memory	2048	memory size (in disk pages)
NetBw	0.1, 5, 20	network bandwidth (Mbits/sec)
MsgInst	20000	instructions to send or receive a message
PerSizeMI	3	instructions per byte sent
DiskInst	5000	instructions to read a page from disk

Table 2: Simulation Parameters and Main Settings

5.1 Simulation Environment

To study the performance of scrambling rescheduling, we implemented the scrambling architecture of Section 3 and the policies described in Section 4 on top an existing simulator that models a heterogeneous, peer-to-peer database system such as SHORE [CDF+94]. The simulator we used provides a detailed model of query processing costs in such a system. Here, we briefly describe the simulator, focusing on the aspects that are pertinent to our experiments. More detailed descriptions of the simulator can be found in [FJK96, DFJ+96].

Table 2 shows the main parameters for configuring the simulator, and the settings used for this study. Every site has a CPU whose speed is specified by the *Mips* parameter, *NumDisks* disks, and a main-memory buffer pool of size *Memory*. For the current study, the simulator was configured to model a client-server system consisting of a single client and eight servers. Each site, except the query execution site, stores one base relation. In all the experiments described in this paper, the servers were not performing any other work then servicing pages upon request, that is, the load on servers is minimum.

The CPU at each site is modeled as a FIFO queue and the simulator charges for all the functions performed by query operators like hashing, comparing, and moving tuples in memory, as well as for system costs such as disk I/O processing and network protocol overhead as described below.

Disks are modeled using a detailed characterization and settings adapted from the ZetaSim model [Bro92]. The disk model includes costs for random and sequential physical accesses and also charges for software operations implementing I/Os. The unit of disk I/O for the database is pages of size *DskPageSize*. The disks prefetch pages when reads are performed. In the current version of the simulator, 4 pages are obtained for each read access request made to the disk. In addition to the disk costs, there is always a charge of *DiskInst* instructions for each disk access. In our experiments, disks were seen to deliver data at an average rate of approximately 10 Mbits/sec with sequential I/Os, and a rate of approximately 3 Mbits/sec with random I/Os.

In this study, the disk at the query execution site (i.e., client) is used only to temporarily store intermediate results and base relations that are materialized during a query execution. The actual base relations are stored on disk at the servers (one relation per server, in this case). Although servers are configured with memory, the workload used in the experiments here is performed such that the server memory is not useful (i.e., there is no caching across queries and relations are accessed once per query) . Thus, in the experiments that follow, base relations are always read (sequentially) from the servers' disks for each query execution.

The network is modeled simply as a FIFO queue with a bandwidth dictated by the *NetBw* parameter; All processing sites share this single communication link. Three different bandwidth settings are used in the experiments that follow: slow (0.1 Mbit/sec), medium (5 Mbit/sec), and fast (20 Mbit/sec) in order to study cases where the system is network-bound, roughly balanced, and disk-bound at the query site respectively. The details of a particular technology (Ethernet, ATM) are not modeled. The cost of sending messages, however, is modeled as follows: the simulator charges for the time-on-the-wire (depending on the message size and the network bandwidth) as well as CPU instructions for networking protocol operations which consist of a fixed cost per message (*MsgInst*) and a per-byte cost based on the size of the message (*PerSizeMI*). The CPU costs for messages are paid both at the sender and the receiver.

The query execution model uses a synchronous (i.e., non-streaming) approach to remote data access. That is, when an operator running at the query site needs data from a remote source, it sends a request (of *RequestSize* bytes) to that source and waits for the reply (of course, other operators can run during this period). A source responds with with a block of *TransferSize* bytes of data. After the operator has consumed this data, it issues another request to the source.

Finally, we modeled a bursty environment by adding to each remote server a small piece of software. Every time a message is about to be sent by a site, the software checks to see if the message must be delayed. The duration of the delay as well as the moment when the delay is effectively enforced are fully configurable, and can range from a fixed duration enforced every time a given number of messages have been exchanged to a random duration and a random occurrence of delays using several probability distributions.

For all the experiments, we have set the value of the timer that actives the scheduler as a multiple of the expected round-trip time for requesting and obtaining a data page from an unloaded source in an unloaded network. In our experiments (except where noted) the timer is set to ten times the duration of this round-trip.

5.2 Workload

The workload used for all the experiments described in Section 6 consists of two versions of the query tree shown in Figure 2. The basic query is an 8-way join structured as a balanced bushy tree. As stated in Section 5.1, each base relation (A through H) is stored on a separate remote site, and scans of the base relations are executed at the remote servers. All other operators, i.e., joins (represented by circles in the figure), are executed at the query execution site. In the experiments we focus our study on hash-based joins.

The tuples of all base relations are 100 bytes each. As shown in Figure 2, there are two parameters for setting the (possibly different) cardinalities of the base relations. These parameters are indicated by the

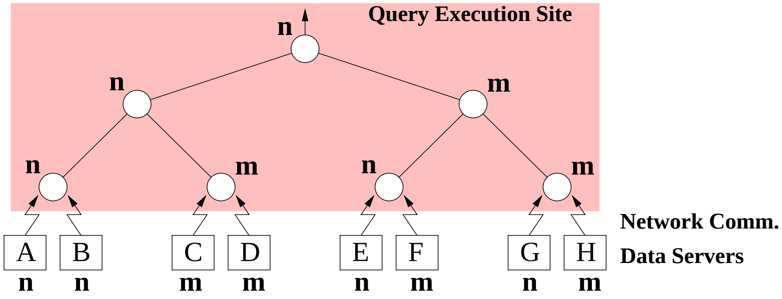


Figure 2: Query Tree Used for the Experiments

letters **n** and **m** in the figure. These same parameters are also used to set the cardinalities of the intermediate results produced by the various joins.

The two versions of the tree that are used in the study are called *uniform* and *non-uniform*; they differ in the settings of the cardinality parameters. For the uniform tree, **n** and **m** are set to be equal so that all base relations have the same size and all joins return a result that is the size of a single base relation. In this case, we set **n=m=10,000**, so that all base relations and join results consist of 1MB (250 disk pages) each. With this setting, all hash joins can be performed without partitioning.

For the non-uniform tree, **m** is set to be an order of magnitude greater than **n** (**n=10,000** and **m=100,000**). In this case we have base relations and intermediate results of either 1MB (250 pages) or 10MB (2,500 pages). The order of magnitude difference between **n** and **m** has two major consequences in our study. First, the hash join of relations C and D requires partitioning in this case, because neither of the relations can fit in memory. Second, the query execution makes better use of pipelining here than in the uniform query tree, as the right-hand sides of many of the joins are large. Recall that given sufficient memory, right-deep hash joins can be executed in a pipelined fashion, thereby avoiding materialization of the right-hand input (i.e., the *probe* relation). Thus, although many of the right-hand sides are relatively large in this query, they do not need to be staged to and from disk when the query executes normally.

These particular queries were chosen for the following reasons. First, an 8-way join query is complex enough to provide sufficient latitude for the scrambling policies and it allows us to investigate the differences and similarities among them. Second, the use of a bushy tree, which is more general than a left- or right-deep tree (i.e., it contains both left- and right-deep components), allows us to investigate scrambling behavior for both left- and right-deep plans. In addition, a bushy tree provides additional options for scrambling beyond those that arise with the more restrictive plans. Finally, we study both the uniform and non-uniform cases in order to compare scrambling in a situation where changes to the execution schedule are likely to have small effects on performance (i.e., the uniform case) and in a situation where it could conceivably have a large, negative impact on performance (i.e., the non-uniform case). Thus, these two queries, plus the ability to vary key system parameters such as the network speed, provide sufficient flexibility to allow us to cover a large area of the performance space for dynamic scheduling.

We also describe (in Section 6.2.4) a set of experiments designed to study the potential impact of scram-

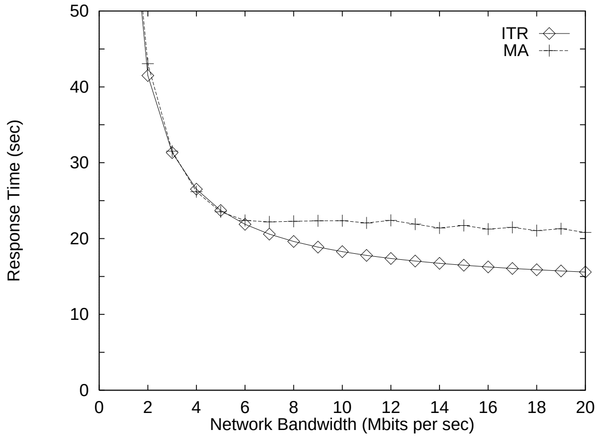


Figure 3: Response time, Uniform Tree

bling rescheduling on an application environment. In this section we use in a simplified version of a query of the TPC-D benchmark.

6 Experiments and Results

In this section we present experiments that analyze the trade-offs raised by scrambling rescheduling. We first investigate the impact of parallel materializations in the absence of delays. We then introduce delays in the execution of the queries to explore the potential benefits of overlapping delays with other work for various delays and network bandwidths.

6.1 Parallel Materializations and Network Speed

As stated in the introduction, the key technique that Query Scrambling rescheduling uses is the introduction of parallelism into the execution of a query in response to unexpected delays. Such parallelism is intended to hide delays by overlapping them with other useful work performed while waiting for missing data to arrive. Before investigating the performance of scrambling rescheduling policies in the presence of delays, however, we first examine the impact of parallelism in the absence of delays. By doing so, we are able to isolate the potential benefits and consequences of such parallelism on the normal execution of queries.

Figure 3 shows the response times of the Uniform query executed with the *ITR* and *MA* policies as the network bandwidth (*NetBw*) is increased from 2 Mbits/sec to 20 Mbits/sec.[5] As expected, the response time for both policies improves dramatically as the bandwidth is increased up to a point and then levels out. With very slow networks, the cost of query execution is dominated by the network costs and the policies have similar performance. As the network speed is increased (up to 5 Mbits/sec), the performance of the policies begins to diverge and *ITR* shows better performance than *MA*.

The performance of *ITR* is quite simple to explain. The main components of performance in this system

[5]Results for bandwidths lower than 2 Mbits/sec are not shown here. The response-time in this range is nearly totally dependent on the network speed, and thus, it increases proportionally with the slowdown of the network.

are the local (i.e. query site) processing and I/O, the remote (server) processing and I/O, and the network. With the *ITR* policy, very little of this work is overlapped. At low bandwidths, the portion of the response time that is due to network time-on-the wire costs is significant (e.g., 75% of the total at 2Mbit/sec). As the network speed is increased, the portion of the response time that is due to time-on-the-wire decreases and has smaller impact on the overall performance of *ITR*. Thus, as can be seen in Figure 3, improving the bandwidth for *ITR* beyond a certain point provides increasingly smaller gains.

In contrast to *ITR*, *MA* has a high degree of parallelism, so the explanation behind its performance here is slightly more subtle. At low bandwidths, the network can become a bottleneck when data are requested from multiple sources in parallel. When the network is the bottleneck, the performance of *MA* is almost completely dependent on it.[6] As the network bandwidth is increased, it no longer is the bottleneck, but the local disk (at the query site) soon becomes a bottleneck. Recall that *MA* obtains its high degree of parallelism by materializing data on the local disk. This materialization costs disk writes when the data is brought in, as well as disk reads when the data is eventually accessed by query processing.

Once the disk bottleneck is reached by *MA*, it actually has worse performance than *ITR*. This is because the *ITR* policy does no local I/O for the Uniform query. With a fast network, its performance is dictated by the local query processing and the (relatively fast) sequential I/Os done at the remote servers. The same general performance behavior, with larger response times, is observed for the two policies when using the Non-Uniform query.

The important lesson here is that materializing base relations in parallel with the query execution does not improve performance in the absence of delays. For slower networks, the performance of *ITR* and *MA* were roughly equivalent, and for faster networks, *MA* actually performed worse than *ITR*.

6.2 Rescheduling With Delays

We examined the performance of *ITR* and *MA* in the absence of delays across a range of network speeds, in order to gain an understanding of the performance tradeoffs of parallel materialization. In this section, we examine *ITR* and *MA* policies as well as two *reactive* ones (*RM* and *RMJ*) in the presence of various delays for slow (0.1 Mbits/sec), medium (5 Mbits/sec) and fast (20 Mbits/sec) network speeds. The slow network setting is intended to model speeds that are on the order of what could be obtained at a decently connected site with today's Internet technology. As shown in the previous section, with a slow network, little care needs to be taken when using the local resources at the query execution site, as they contributed at most a small portion to the total response time. The medium network speed was chosen so that the system would be roughly balanced between network bandwidth and local disk rates (under mixed random/sequential access) and the fast network is used to examine the performance of the policies when the local resources are the crucial factor in performance.

In the following, we examine the performance of the policies under different delay scenarios (e.g., bursty and initial delay) for the two query trees presented in Section 5.2. We first present the results for the Uniform

[6] Once all data has been downloaded by *MA*, there is a relatively small amount of additional work that must be performed at the query site in order to complete the query. The cost of this work is not impacted by the network bottleneck.

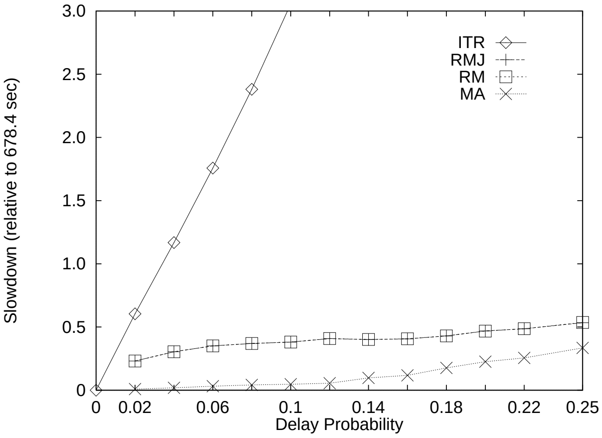


Figure 4: Slowdown - Bursty Delay, All Relations
Net: 0.1 Mbps, Delay: 10.52 sec (3x Timer), Uniform Tree

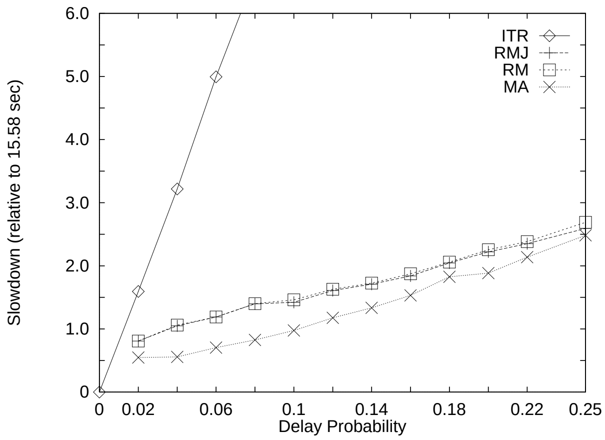


Figure 5: Slowdown- Bursty Delay, All Relations
Net: 20 Mbps, Delay: 0.63 sec (3x Timer), Uniform Tree

query and then for the Non-Uniform one.

6.2.1 Uniform Query Tree: Bursty Environment

We first examine the performance of the policies when all of the base relations are subject to random delays throughout the entire execution of a query. Delay is applied in the following way: Each remote source flips a weighted coin before sending a page of tuples to the query execution site. The outcome of the coin toss determines if the source should transmit the page normally, or if it should stall for a specified period before sending its page.[7] In all experiments, the timer used by the reactive policies to detect problems with a remote source is set to 10 times the expected round-trip delay time for a data request between the query site and a source (thus, the timer is different for each *NetBw* setting). In this experiment, the delay period (for each random delay) is set to three times the value of this timer. Because of the fixed value for the delay and timer, it is known that the query processor will *timeout* on a source each time that source delays. In this case, the timeout will be detected one-third of the way through the delay.

In the remainder of the performance section, all graphs show the percentage slowdown of the query (compared to the non-delayed case) as the probability of delay for each page transmission is increased along the x-axis. Figure 4 shows the slowdown for Uniform query under the various policies, using the slow network (0.1 Mbits/sec).[8] In this case, the duration of the delay is 10.52 sec (the is timer set to 3.5 seconds, here). Slowdown is computed by subtracting the normal response time for the query (in this case, 678.4 seconds) in the absence of delays, from the observed response time in the delayed case, and dividing by the normal response time.

As can be seen in the figure, the slowdown for all policies shown increases linearly with the delay proba-

[7] In those cases where random delays are used we ran each experiment 12 times and then averaged the results to get the final results presented here.

[8] Although we measured slowdowns for delay probabilities as high as 90%, we only show probabilities upto 25%, here, as lines remain linear beyond this point.

bility, but there are dramatic differences in the slopes of the lines. The *ITR* policy is the most sensitive to delay here. Since *ITR* accesses the base relations sequentially it incurs the full cost of *every delay on every source*. In this experiment, at 10% delay probability the query runs 3.1 times slower than when there are no delays. At 25% delay probability (not shown) the query runs 7.75 times slower.

This result is to be expected. The static, sequential scheduler is unable to overlap any delays, so query execution time is increased by the sum of the delays experienced by all of the remote sources. At 10% delay probability, there are 200 delays of 10.52 seconds each, so the total delay is 2104 seconds, compared to a normal query execution time of only 678.4 seconds. In this case, the slowdown for the standard query execution at 10% delay probability is $(2782.4 - 678.4)/678.4 = 3.1$. At 25% delay probability, there are 500 delays of 10.52 seconds each, so the total delay is 5260 seconds. The corresponding slowdown is $(5938.4 - 678.4)/678.4 = 7.75$.

Turning to the non-sequential policies, it can be seen that they too incur a linear slowdown as the delay probability is increased. The slopes of the increases, however, are much lower than for the sequential policy. By requesting data from multiple sources, the three policies can tolerate delays of a subset of those sources by overlapping them with other work.

The best policy for coping with delay in this experiment is *MA*. This policy is the most aggressive one, since it immediately initiates parallel materializations and continuously materializes data regardless of the potential delays. At 25% delay probability, *MA* executes in 905.45 seconds, that is, it is slowed by a factor of 33% with respect to the execution time of the query with no delays. Since the total delay in this case is 5260 seconds, this policy is able to *hide* 4354 seconds of delay by overlapping it with other useful work (e.g., the retrieval of other base relations) and other delays. Thus, while in the no-delay case with the slow network, *MA* and *ITR* displayed similar performance, in the presence of multiple delays (as may arise in a bursty environment), *MA* has a tremendous advantage over *ITR*.

The two reactive policies, *RM* and *RMJ* are also very beneficial here, but their performance is slightly worse than *MA*. The performance difference arises because the reactive policies must wait until the timer expires before resuming materializations when the left-most (i.e., non-scrambled) data source experiences a new delay. In contrast, *MA* does not rely on any timer mechanism. The performance difference seen in the figure, thus, is the sum of all the timer waits encountered by the reactive policies. In this scenario, with bursty delays on all relations, even a low probability of delay results in significant burstiness, so an aggressive policy will work well here.

Figure 5 shows the performance of the policies when the fast network speed (20 Mbits/sec) is used.[9] Here, even with a very fast network, the policies that hide delays using parallel materializations do well, and the more aggressive *MA* policy performs best here. This result is in contrast to the no delay case (Figure 3) where the performance of *MA* was worse than *ITR* for faster networks. The reason for this difference is that in this experiment, the large amount of delay overwhelms the cost of local processing, so even though *MA* performs much local I/O here, that I/O is more than paid for by the overlapping of delays.

[9] The results for the balanced network are similar, so are not shown here.

6.2.2 Uniform Query Tree: Initial Delay

One lesson from preceding experiments is that if multiple sources are likely to have multiple delays, even the most simple forms of parallelism offer a good opportunity to hide delays and that an aggressive policy can do well. In this section, we examine the potential negative impact of scrambling too aggressively by investigating a case where there is much less delay than in the previous cases. To accomplish this, we vary the length of a single, initial delay on the left-most relation of the query tree (i.e., relation A). As stated in Section 1, under the initial delay model, sources experience a single delay before transmitting their first tuple, but perform reliably after that. The x-axis on the graphs shows the magnitude of this initial delay as a percentage of the time required to execute the query in the absence of any delay. The y-axis shows, as before, the percent slowdown compared to normal execution.

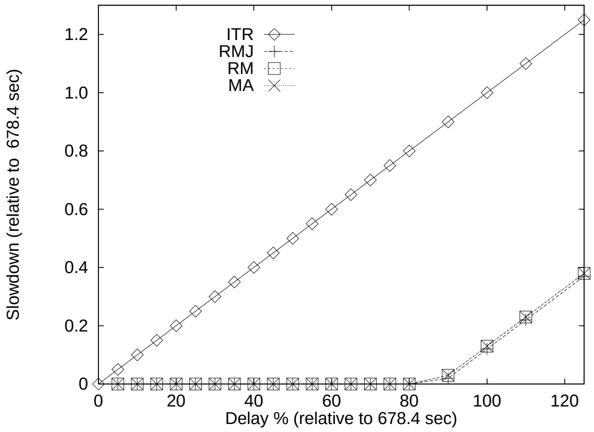


Figure 6: Slowdown, Initial Delay on A
Net: 0.1 Mbps, Delay % of 678.4 sec, Uniform Tree

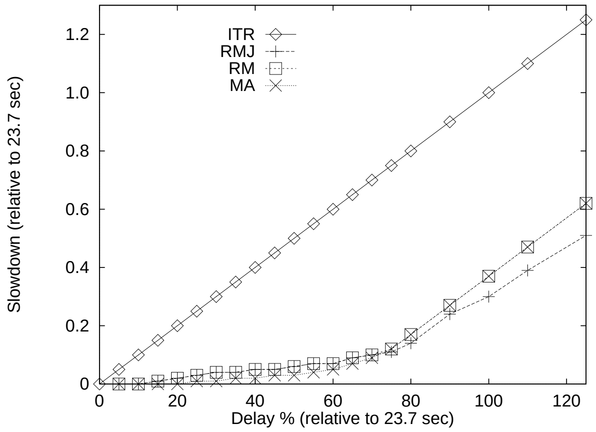


Figure 7: Slowdown, Initial Delay on A
Net: 5 Mbps, Delay % of 23.7 sec, Uniform Tree

Figure 6 shows the performance of the policies for the slow network. In this case, the execution time of the *ITR* policy is 678.4 seconds, and is completely dominated by the network cost. The result of this imbalance is that the use of local resources at the query processing site is effectively free, so all scrambling policies can hide virtually all of the delay up to 80%, after which they run out of work to perform and the slowdown increases linearly with the delay.

Figure 7 shows the performance of the policies with the balanced network. With this setting, the query execution time with no delays is 23.7 seconds and the overhead of materializations can have a somewhat larger impact. In this figure, all three parallel policies are able to hide most of the delay up to 70%, after which they increase linearly with the delay. Beyond 70%, *RMJ*, the reactive policy that can instigate join processing in addition to materializing base relations has a slight advantage over the other parallel policies because it performs some additional work (i.e., joins) whereas the other policies block after all base relations have been materialized if the tuples of A are still missing. As such, once the tuples of A have been received, the work that must be done by *RMJ* to complete the query is small and the query finishes relatively quickly. Although it is not shown in the graph, with higher delays (e.g., beyond 130%) *RMJ* eventually performs all the join processing it can without A (i.e. C ⋈ D, and E ⋈ F ⋈ G ⋈ H) at which point its response time curve becomes parallel to the others.

If a slow network makes local disk I/O virtually free, then a faster network makes local I/O relatively more expensive. Figure 8 shows the performance of the polices when the fast network is used. In this case, *MA*, the most aggressive policy, performs relatively poorly. *MA* always materializes all base relations concurrently with the normal query execution, so in the presence of short delays, *MA*, which is a static policy, commits to reading most of its data from the local disk using random I/O (3 Mbit/sec). In contrast, *ITR* is able to access its data over the high speed network in this case. (It is important to note, however, that even though the network bandwidth is 20 Mbits/sec here, *ITR* accesses remote sources one-at-a-time, and so is limited by the speed at which a remote source can provide data, i.e., 10 Mbits/sec.)

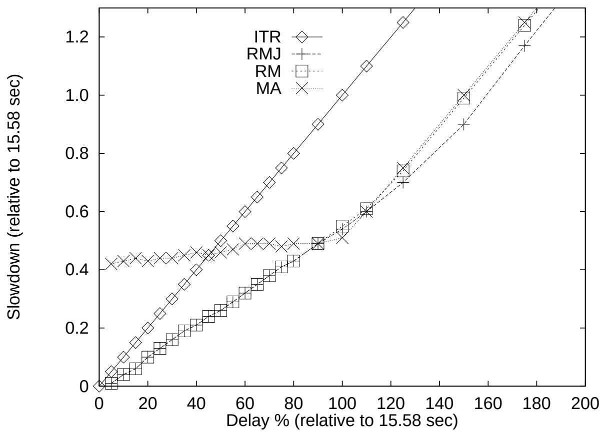


Figure 8: Slowdown, Initial Delay on A
Net: 20 Mbps, Delay % of 15.58 sec, Uniform Tree

The net effect is that in this case, the extra cost of the random, local I/O that *MA* performs in order to materialize and read base relations outweighs the benefit gained by hiding delay. Therefore, *MA* performs worse than *ITR* up until a delay of about 45%. *RM* and *RMJ* avoid the problems of *MA*, because both are able to stop the materialization of base relations when the delay of A is over. Because of this, the reactive policies are able to read their materialized data *sequentially* and thus, unlike *MA*, can obtain materialized data at the same speed (i.e., 10 Mbits/sec) that *ITR* can obtain data from the network. As a result, the reactive policies, unlike the static ones, are able to effectively hide delay by materializing base relations and then reading that materialized data for no penalty (compared to *ITR*) after the delay is over. As the delay is increased, the penalty that *MA* pays is erased, and at a delay of 95% and beyond, it performs similarly to *RM*. Finally, it should be noted that as seen in the balanced network case (Figure 7) *RMJ* performs slightly better than *RM* and *MA* at higher delays because it is able to overlap somewhat more delay by executing joins.

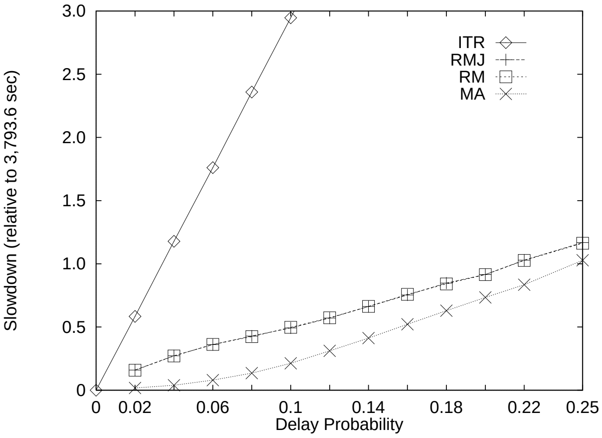


Figure 9: Slowdown, Bursty Delay on All Relations
Net: 0.1 Mbps, Delay: 10.52 sec (3x Timer), Non-Uniform Tree

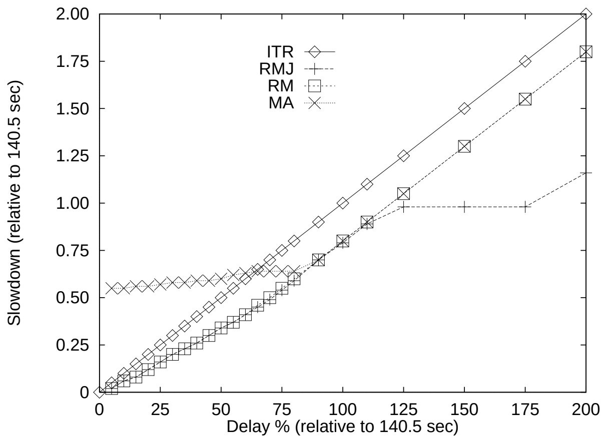


Figure 10: Slowdown, Initial Delay on A
Net: 20 Mbps, Delay % of 140.5 sec, Non-Uniform Tree

6.2.3 Non Uniform Query Tree

In this section, we briefly describe the performance of the policies with the non-uniform query tree as described in Section 5.2. This tree contains a mix of large and small relations, as well as high- and low-selectivity joins, and allows us to examine the performance of the policies in a situation where changes to the execution plan chosen by the optimizer could conceivably have a large, negative impact on performance. Recall that one impact of the non-uniform query is that one of its joins requires the hash join algorithm to use partitioning. We first investigate the performance of the policies in the bursty environment and then in the case of a single initial delay.

For the bursty delay cases, the results for the non-uniform query show the same behavior as was seen for the uniform query. That is, for all three network speeds, the parallel policies dramatically improve the performance of the query when it experiences many delays. Such a result is to be expected, since using local resources to support overlapping delay is virtually free compared to the amount of experienced delays. For the balanced and fast networks (not shown) the results are essentially the same as those for the Uniform query in Section 6.2.1. For the slow network (Figure 9), the results are also very similar to the Uniform case, except that with the mixed relation sizes of this tree, the parallel policies are slightly less effective in hiding delay than with the Uniform query tree.

Figure 10 shows the performance of the policies in the initial delay case for the fast network. As was seen for the uniform query (Figure 8). The performance here also quite similar to what was seen for the uniform query except for one aspect: At 125% delay, *RMJ* initiates the *partitioning* of the materialized base relation in order to perform the join of C and D. Between 125% and 175%, therefore, its curve is flat because this corresponds to the time required to partition the two relations before doing the join. This work is entirely beneficial to the query and does not incur any additional overhead because these two relations have to be partitioned anyway, either by the policy or by the query once the delay is over. Beyond 175%, *RMJ* has performed all the possible work and its performance increases linearly with the delay.

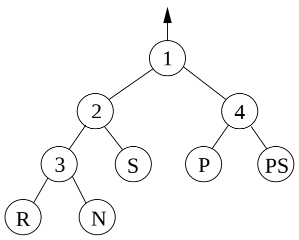
6.2.4 Impact of Rescheduling

In this section we describe a set of experiments to study the potential impact of scrambling rescheduling for a more application-oriented query than the uniform and non-uniform cases shown so far. The experiments use a simplification of the query Q2 of the TPC-D benchmark [Tra95]. We chose this query because it is relatively simple, yet processes a five-way join. The cardinalities of the relations involved in this query are as follows: PART: 200,00 tuples of 164 bytes, SUPPLIER (S): 10,000 tuples of 197 bytes, PARTSUPP (PS): 800,000 tuples of 219 bytes, NATION (N): 25 tuples of 185 bytes and REGION (R) 5 tuples of 181 bytes.

The query is:

```
SELECT S_ACCTBAL, S_NAME, N_NAME, P_PARTKEY, P_MFGR, S_ADDRESS, S_PHONE, S_COMMENT
FROM PART, SUPPLIER, PARTSUPP, NATION, REGION
WHERE P_PARTKEY = PS_PARTKEY AND S_SUPPKEY = PS_SUPPKEY AND P_SIZE = 15 AND P_TYPE LIKE ‘BRASS’
  AND S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY AND R_NAME = ‘EUROPE’
```

In this experiment, each base relation resides on a separate data source. Selects are performed at the data sources and the query execution site receives only the selected tuples. Figure 11 shows the query tree run at the query source site and the resulting cardinalities of the input relations and joins.



Result	Card.	Size	4k pages
R	1	4	1
N	25	8	1
3	10	4	1
S	10,000	197	500
2	5,000	193	239
P	100	29	1
PS	800,000	8	1,585
4	10,000	29	72
1	10,000	222	556

Figure 11: TPC-D Query Tree and Relation Sizes at Execution Site

We ran this query under various conditions of delays and network speed (network speed of 0.1, 5, and 100 Mbit/s). Here, we illustrate only the cases where initial delays are applied to the query. We delay one source per experiment and reschedule the query using the *RM* policy. The response times of the query in the absence of delays are: 724.8 seconds with a 0.1Mbit/s network, 28.02 seconds for 5Mbit/s and 17.3 seconds for 100Mbit/s. Three delay values were tried: 50%, 90%, and 150% of the execution time of the query in the non-delayed case. This totals to 45 experiments, plus 9 experiments with no delay. For each experiment, both the scrambling and non-scrambling versions were executed, and the relative performance improvement calculated. Several broad statements can be made about the behavior of scrambling for this query.

In all experiments, scrambling either improved performance significantly, or had negligible (under 0.01%) effect. The maximum performance improvement was 46.36% for an initial delay on the R relation of 90% the time required for the non-delayed query with a network speed of 0.1 Mbit/s using the MA policy. When the delayed relation is PS, the scrambling performance improvement is less than 0.03% for all policies. Since

Delay	No Scrambling	Scrambling	%
no delay	28.02		
on R	53.24	38.00	28.62
on N	53.24	38.17	28.29
on S	53.24	41.60	21.86
on P	53.24	41.66	21.74
on PS	53.24	53.23	0.01

Table 3: Response Times (sec), 5 Mbits/s, 90% initial delay (25.22 sec)

% Improvement	Total
0.00	3
0.01 to 10.00	6
10.01 to 20.00	11
20.01 to 30.00	10
30.01 to 40.00	12
> 40.00	3

Table 4: Histogram of Observed Improvements (*RM* Policy)

PS is the last relation to be processed, all other work has been performed and scrambling cannot overlap unfinished work with the delays in PS. Generally, as would be expected, performance improvement declines as the delay appears in relations later in the plan. Also, generally, performance improvement declines as the network speed increases. Table 3 shows a typical experimental result. We see that the performance improvement is above 20% for delays on all relations except for the delay of PS, as noted above. This is typical for scrambling in the initial delay environment. The general performance improvement is indicated by the delay on R, and performance declines (slightly) as the delay appears later into the tree, until a sharp drop at the delay of the last relation. Finally, we also categorized all 45 experiments for the *RM* policy by the size of improvement. This classification is given in Table 4.

6.2.5 Discussion

The experiments of this section showed that simple reactive polices such as *RM* and *RMJ* are fairly robust, even when some of the relations and intermediate results are scaled up by an order of magnitude. The main reason for this robustness is that these policies constantly monitor the execution of delays and enforce parallelism only when delays are experienced. As such, they are able to hide the delays with useful work without incurring a high additional cost. Even when base relations are big, as is the case for the non-uniform query tree, these policies bring a substantial improvement. One reason that materializing large relations does not hurt performance for these policies is that they suspend the rescheduled operations when delays are short so that extra work is not performed in the absence of delays. The overhead of materializations becomes significant only if most or all of the relations can be materialized and this can only happen when the delay is large. For the same reason, the joins materialized by *RMJ* do not typically hurt performance.

Another case that we studied (but do not present here) is for Cartesian products and joins whose results are significantly larger than the sum of their inputs. In such a case it conceivable that materializing such a result could hurt performance, but we did not see dramatic differences in our studies (for the reasons outlined above). Furthermore, query optimizers typically try to avoid such costly operations, making the occurrence of these cases less likely. Interestingly, it is fairly easy to protect query scrambling against such pathological cases. For example, we extended the policies to materialize only joins having a small ratio between the size of their result and the size of their input. This policy was able to avoid problems in the few cases where they arose.

7 Related Work

As stated in the introduction, techniques that try to adapt a query to a changing environment broadly fall in the proactive and reactive categories.

In the proactive category, the techniques gather as much information as possible to predict the state of the run-time system during query execution and use this information to construct the best query execution plan. Volcano [CG94, Gra93] introduces at optimization time *choose-plan* operators in the query tree that enable the selection of a particular query execution plan at runtime once enough information has been gathered. HERMES [ACPS96] records the costs of remote accesses into an history used to better estimate the costs of future accesses. Mariposa [SAL+96] builds query plans after having negotiated a price-performance trade-off with data providers. All these approaches ultimately decide at query start-up time the execution plan of a query, which however remains fixed for the whole duration of the query execution.

In contrast to the proactive category, techniques in reactive category monitor the progress of queries and modify query execution after execution has started. (Note that techniques in the proactive and reactive categories are generally complementary.) Monitoring determines if execution should deviate from the plan for some unforeseen reason. Reasons include inaccurate estimates for intermediate result sizes and direct considerations of problems with response times from remote sources are not accounted for.

[BRJ89] proposes a reactive technique in which the execution of a distributed query proceeds through three phases: (i) a monitoring phase observing the progress of the execution of the query; (ii) a decision making phase during which a new strategy for executing the query is computed; and (iii) a corrective phase in which the current execution is aborted and a new execution is initiated. A similar approach is used in Rdb/VMS [Ant93].

Both InterViso [TTC+90] and MOOD [ONK+96] are heterogeneous distributed databases that perform query optimization *while* the query is executing. Heterogeneous distributed database divide a query into a collection of subqueries and a composition query. There is one subquery for each remote source and a composition query than combines the results of the subqueries. These systems use a reactive technique that interleaves the execution of subqueries with the execution of the composition query by monitoring the arrival of the answer to subqueries and dynamically executing the composition query.

A technique similar in spirit to scrambling rescheduling is used to improve the access time to tertiary storage in [SS96]. This work divides queries into parts that can be executed independently in arbitrary order.

The order in which the parts are executed is dynamically chosen depending on the data each part needs to fetch, the state of the disk cache and the state of the the tertiary memory (i.e., the platter currently loaded). The scheduler's objective is to maximize the overall system throughput.

As stated in Section 1, the work described here builds on our initial definition of Query Scrambling [AFTU96]. Additional experimental results are also available in reference [AFT96, pages 11–21].

8 Conclusions

Query scrambling is a reactive technique for coping with unpredictable delays for wide-area remote data access. Query scrambling, in its most general sense, monitors query execution and reacts to delays by *on-the-fly* rescheduling query operators and possibly synthesizing new operators to run. This article, we focused on the tradeoffs that arise for the *rescheduling* portion of the query scrambling technique.

We first described the performance problems that arise from the iterator model, i.e., when executing a static query plan in the presence of unexpected delays. We then discussed alternatives for rescheduling and the tradeoffs among them. In particular, we focused on the way that memory management issues influence the feasibility of different rescheduling options. In general, memory management issues lead to rescheduling techniques that use minimal amounts of memory. Such techniques allow operators to be run "out-of-turn" by materializing their results to the local disk of the query execution site.

We studied two reactive policies: RM, which initiates the materialization of data from all remote sources when a delay is detected during normal query processing; and RMJ, which works similarly to RM, but in addition, has the ability to reschedule (and materialize) individual join operators, one-at-a-time. RMJ uses more memory to be reserved for rescheduling than RM but it has a greater opportunity to perform useful work when delays arise. The memory requirements for RMJ are much less than for a more general policy that would allow entire subtrees to be rescheduled at once. More importantly, RMJ avoids the potential problems that a more general policy would encounter if the rescheduled operations themselves became delayed.

The two reactive policies were compared to two static ones: ITR and MA. Policy ITR is an iterator-based execution policy, while MA augments such a policy by opening scans on all remote sources in parallel. Policy MA was used to investigate the impact of parallelism outside of a reactive policy. The polices were compared using a uniform and a non-uniform query tree. In addition, results using a simplified TPC-D query were also presented. The experiments were run using three network settings: one where the network was the dominant cost, one where the network and local disk were balanced, and one where the system was disk-bound at the query execution site. The slow setting is of the same order of what many current wide-area environments experience (even if the actual wires are somewhat faster). The balanced and fast networks show how the policies will change as deployed network technology continues to improve.

The performance studies showed that in the absence of delay, parallel materializations had little impact on performance for slow networks and were detrimental for fast networks. When delays were present, however, such parallelism provided substantial benefits; in a situation where all data sources are subject to delays, the performance improvement due to parallel materializations is a factor of the number of sites involved in the query. With the slow network, parallel materializations where seen to be always beneficial, and the reactive techniques were hurt slightly by their delay in initiating and resuming materializations. With the faster network, and less delay, however, the blind use of materializations as used by *MA* was seen to significantly hurt performance, while the reactive approaches were able to successfully hide delay in many cases. In terms of the reactive approaches, they were seen to have similar performance in most cases, but the ability to execute joins was seen to benefit *RMJ* in certain cases with long initial delays. Finally, using a query based on Q2 of TPC-D, we observed that the *RM* policy was effective at hiding delay across a range of delay scenarios.

Acknowledgments

We would like to thank Björn Jónsson and Tolga Urhan for providing invaluable assistance and information about the simulator used for this work. Thanks to Philippe Bonnet and Luc Bouganim for comments on earlier drafts of this paper.

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.

[AFT96] Laurent Amsaleg, Michael J. Franklin, and Anthony Tomasic. Query scrambling for bursty data arrival. Technical Report UMCP-CSD CS-TR-3714, University of Maryland, College Park, Maryland, November 1996.

[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.

[BE96] O. Bukhres and A. Elmagarmid. *Object-Oriented Multidatabase Systems.* Prentice Hall, 1996.

[BMG93] J. Blakeley, W. McKenna, and G. Graefe. Experiences building the open OODB query optimizer. In *Proc. of the ACM SIGMOD Int. Conf.*, page 287, Washington, DC, May 1993.

[BRJ89] P. Bodorik, J. Riordon, and C. Jacob. Dynamic distributed query processing techniques. In *Proc. of the 17th annual ACM Computer Science Conf.*, pages 348–357, Louisville, Kentucky, February 1989.

[Bro92] K. Brown. Prpl: A database workload specification language. Master's thesis, University of Winconsin, Madison, Winsconsin, 1992.

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Int. Conf.*, Minneapolis, Minnesota, May 1994.

[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.

[DFJ+96] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc. of the 22th VLDB Int. Conf*, Bombay, India, September 1996.

[FJK96] M. Franklin, B. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proc. of the ACM SIGMOD Int. Conf.*, Montréal, Canada, June 1996.

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

[Kim95] W. Kim. *Modern Database Systems: The Object Model, Interoperability, and Beyond.* ACM Press, New York, NY, 1995.

[MMM96] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *Proc. of the Int. Conf. on Parallel and Distribution Information Systems (PDIS)*, Miami Beach, Florida, December 1996.

[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.

[SAD+95] M. Shan, R. Ahmen, J. Davis, W. Du, and W. Kent. *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter Pegasus: A Heterogeneous Information Management System. ACM Press, 1995.

[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.

[SS96] S. Sarawagi and M. Stonebraker. Reordering execution in tertiary memory databases. In *VLDB*, Bombay, India, 1996.

[Tra95] Transaction Processing Performance Council (TPC). *TPC Benchmark D (Decision Support)*, May 1995. Standard Specification, Revision 1.0.

[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.