Query Scrambling for Bursty Data Arrival*

Laurent Amsaleg
University of Maryland
amsaleg@cs.umd.edu

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

Anthony Tomasic
INRIA
Anthony.Tomasic@inria.fr

Technical Report CS-TR-3714 and UMIACS-TR-96-84

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 may vary widely due to network congestion, link failure, and other problems. In this paper we examine a new class of methods, called *query scrambling*, for dealing with unpredictable response times. Query scrambling dynamically modifies query execution plans on-the-fly in reaction to unexpected delays in data access. We explore various choices in the implementation of these methods 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 it focuses on bursty arrivals of data. We identify and study a number of the basic trade-offs that arise when designing scrambling policies for the bursty environment. Our performance results show that query scrambling is effective in hiding the impact of delays on query response time for a number of different delay scenarios.

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 execution plans are typically generated statically, based on a set of assumptions about the costs of performing

*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 was supported in part by an INRIA Fellowship.

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 be the 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 paper, we extend the scope of query scrambling by investigating approaches to cope with 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 6.

The 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 Paper

Because scrambling introduces both benefits and costs, it must be regulated in an effective way. Thus, the key questions for implementing scrambling are: 1) when should scrambling start; 2) what should be scrambled; and 3) when should scrambling stop. In this paper we explore these key questions by examining several sets of *policies* to control scrambling, 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 paper, 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.

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

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.

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 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

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

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

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.

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 can not 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.

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

[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.

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.

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

In this paper, we assume that the query execution tree shape is fixed during execution, i.e. join ordering is not changed, and we assume that the physical network topology is 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 order have much less impact than delays which appear late.[4]

Consider the impact of physical network topology. If a network delay affects only a single remote source,

[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.

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.

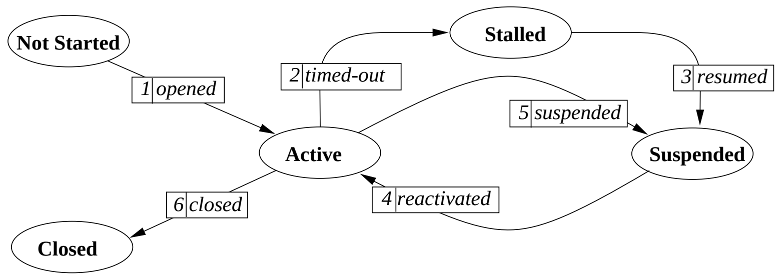

Figure 1: State Diagram for Query Operators

These states are:

- **Not Started**. State of an operator before being opened.
- **Active**. State of the operators that can be scheduled by the OS for execution. The actual order in which the OS schedules the **Active** operators 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 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 and also the main settings for configuring the query scrambling scheduler.

4.1 Simulation Environment

To study the performance of query scrambling in a bursty environment, we implemented the scheduling engine and several policies 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 connected to eight servers. Each site, except the query execution site, stores one base relation.

In this study, the disk at the query execution site (i.e., client) is used to store temporary results. Disks are modeled using a detailed characterization that was 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 and the client's disk cache are pages of size *DskPageSize*. The unit of transfer between sites are pages of size *NetPageSize*. The network is modeled

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)
NetPageSize	8192	size of a network page (bytes)
Compare	4	instr. to apply a predicate
HashInst	25	instr. to hash a tuple
Move	2	instr. to copy 4 bytes
Memory	2048	size of memory (disk pages)

Table 2: Simulation Parameters and Main Settings

simply as a FIFO queue with a bandwidth set as a parameter of the experiments described in the next section. The details of a particular technology (Ethernet, ATM) are not modeled. The cost of sending messages, however, is modeled and 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. The CPU 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.

We extended this simulator by adding a query scrambling scheduler that follows the description given Section 3. We also implemented several query scrambling policies that behave differently when operators of the query time-out or resume. 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 it has to delay this message. 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 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. We evaluate the performance of scrambling with three different network speeds: a fast network (100Mbits per second), a slow network (0.1Mbps) and also a network speed that is roughly equal to the observed bandwidth of the disk local to the client (5Mbps). This bandwidth is determined by the average performance of the local disk when it alternates between period of sequential and random I/Os as it does during the execution of a query.

In the model, all processing sites share a single communication link. This configuration enables us to model increasing latencies on the wire when the network is over-utilized. However, we do not model network congestion and the dropping of packets that could happen in this case. Congestion, however, is unlikely in our environment because we use a page-at-a-time approach to obtaining data from remote sources, rather than a streaming approach. Thus, when an operator requests a data page, this operator has to wait until the page is received before requesting the next one.

A way to tune the sensitivity of query scrambling is to decide when to stop scrambling, as described in Section 2. In all the experiments described in the next section, we force query scrambling to *suspend* when the left-most operator resumes and to return control to the normal iterator-based scheduling of operators. The left-most operator is favored over the others because it corresponds to the last operator opened by the normal iterator-based execution of operators and not by the scrambling scheduler. If no other delay is experienced by the query, than the query will complete its execution with no other extra costs than the ones charged while the left-most operator was stalled. Section 5.1.3 demonstrates the importance of this regulation mechanism.

4.2 Workload

In this section we present the workload used for all the experiments described in Section 5.

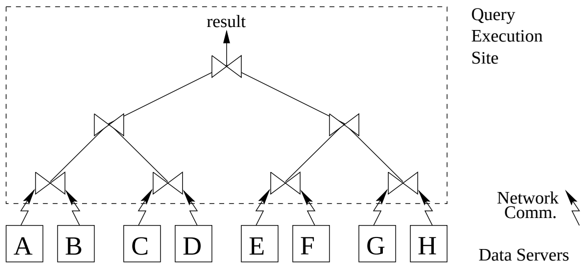

Figure 2: Query Tree Used for the Experiments

For all the experiments described in the next section, we use the query tree represented in Figure 2. The query tree is an 8-way join structured as a balanced bushy tree. For this query tree, all base relations (A through H) are stored on remote sites and their tuples are sent a page-at-a-time when requested to the query execution site through the network. All other operators, i.e., joins, are executed at the query execution site. This 8-way join query tree is used to explore fundamental tradeoffs of query scrambling policies. This tree is big enough to provide sufficient latitude for the policies to possibly perform multiple rounds of scrambling, highlighting different behaviors and their costs. This tree, which is more general than left- or right-deep trees, offers opportunities to reschedule many base relations scans and also subtrees of various sizes, each incurring different costs.

Each base relation used in the study has 10,000 tuples of 100 bytes each, which represents 250 pages of 4Kbytes. In our experiments, all the joins in the query tree produce the same number of tuples, i.e., 10,000 tuples also of 100 bytes each. The main memory needed by one operator is requested and reserved when this operator opens. In the case of a join, the size of its hash-table is of 250 pages. The memory used by an operator is freed when this operator closes.

Query scrambling needs memory to reschedule and materialize operators. Before rescheduling an operator, the scheduler computes the maximum amount of main memory this operator (and its child operators) will need for their execution. This amount depends on the length of the longest branch of the subtree rooted at the rescheduled operator. If there is not enough memory (because other operators already reserved some), then the scheduler does not reschedule the operator nor pre-reserve the memory this subtree needs. Later, when some memory will be freed by other closing operators, the scheduler may try to reschedule again this operator. Note that in the current implementation of the simulator, the memory of a stalled operator is not freed.

5 Experiments and Results

5.1 Non-Pipelined Performance

In the first set of experiments, we investigate the performance of query scrambling for an execution model in which no pipelining is performed. That is, the result of each join operator (excluding the root) is materialized to a temporary file on disk before it is consumed by its parent operator. A non-pipelined execution model is used initially in order to allow us to explore several fundamental tradeoffs of scrambling in the absence of complications due to memory management. In Section 5.2 we study the impact of limited memory.

5.1.1 Policies

Three different query scrambling policies are evaluated in this section. We compare these three policies to the case where no scrambling is performed, that is, the processing of the query incurs the cost of every delay. The performance of non-scrambled query is what would arise under traditional, iterator-based scheduling with no scrambling facilities. All policies perform the same under normal execution. That is, they are not applied until scrambling is invoked. The three policies are:

Materialize-All. When scrambling is initiated, this policy simultaneously materializes all base relations. When both sides of a join are entirely materialized, then this join becomes eligible to run under scrambling. Although only a single join can be materialized at a time, this join is allowed to run concurrently with the on-going materializations of base relations.

All-Builds. Rather than opening all base relations when scrambling starts, this policy opens all of those base relations that are on the build side (i.e., the left input) of joins. When a build relation is entirely materialized, then the corresponding join becomes eligible to run under scrambling. Again, such joins are allowed to run concurrently with the on-going materializations of other base relations. Note that in this policy, probe base relations (i.e., those that are the right input of a join) are never materialized but are always obtained directly from remote servers through the network.

Next-Build. This policy is similar to *All-Builds* but it only opens one build relation at a time. If a probe relation or an on-going materialization of a build relation times-out, then the scheduler then opens the next build relation.

5.1.2 Multiple-Delay Experiments

In the first set of experiments we examine the performance of the policies when some or 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.[5] In these experiments the delay period is fixed at 0.78 seconds, or three times the value of the timer used by the query processor to detect problems with a remote source. Because of the fixed value for the delay period, 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 (0.26 seconds).

Figure 3 shows the percentage slowdown of the query as the probability of delay for each page transmission is increased along the x-axis. In this first experiment, random delay is applied to the page transmissions of all eight base relations. The slowdown is computed by subtracting the normal running time for the query (in this case, 45.8 seconds) from the observed running time in the delayed case, and dividing by the normal running time. As can be seen in the figure, the slowdown for all policies shown increases linearly with the delay probability, but there are dramatic differences in the slopes of the lines.

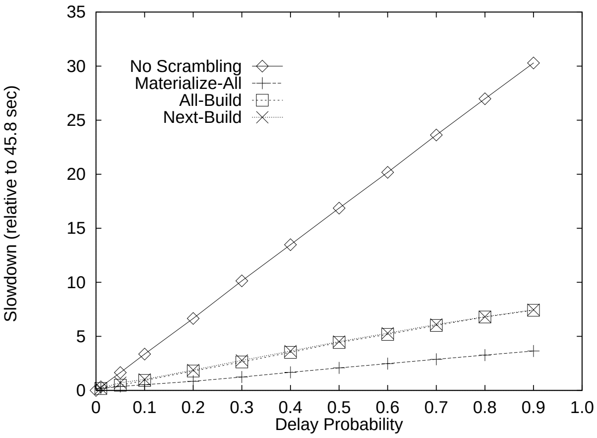

Figure 3: Slowdown, Delay on All Relations
Net: 5 Mbps, Delay: 0.78 sec (3x Timer)

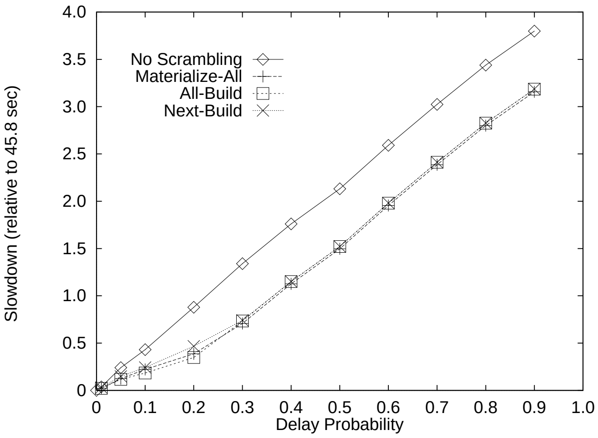

Figure 4: Slowdown, Delay on A Only
Net: 5 Mbps, Delay: 0.78 sec (3x Timer)

The top line of the figure shows the performance of the query if no scrambling is done; that is, it is the performance that would arise under traditional, iterator-based scheduling. When using the iterator model with no scrambling, query processing incurs the cost of every delay — in this experiment at 90% delay probability the query runs 30 times slower than when there are no delays. In this case, the response time of the query is 1449.8 seconds. Such a result is to be expected. The normal scheduler is unable to overlap any delays, so query execution time is increased by the sum of the delays experienced from all of the remote sources. At 90% delay probability, there are 1800 delays of 0.78 seconds each, so the total delay is 1404 seconds, compared to a normal query execution time of only 45.8 seconds. The slowdown for no scrambling at 90% delay probability is $(1449.8 - 45.8)/45.8 = 30.7$.

[5] 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.

Turning to the scrambling 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 no scrambling approach. The best policy for coping with delay in this case is *Materialize-All*. *Materialize-All* is the most aggressive policy; it opens all base relations simultaneously. As a result, it has the potential to overlap the most delay. That is, by requesting data from multiple sources, it can tolerate delays of a subset of those sources. In this experiment, since all sources experience delay, the aggressive approach of *Materialize-All* is beneficial. At 90% delay probability, the query is slowed down by a factor of 3.65 or 167 seconds. Since the total delay in this case is 1404 seconds, *Materialize-All* is able to *hide* 1237 seconds of delay by overlapping it with other useful work, such as retrieval of other base relations and join processing. The other two scrambling policies, *All-Builds* and *Next-Build* are also able to overlap delay, but to a lesser extent than *Materialize-All*. In this case, *Next-Build* performs the same as *All-Builds* because although it initially tries to materialize only one build relation at-a-time, as soon as it detects a delay on such a relation, it proceeds to open the next build. Thus, under any significant delay probability, it will quickly reach the state where it has all build relations open simultaneously. Both policies hide less delay than *Materialize-All* because they do not open a probe-side base relation until the entire corresponding build relation has been materialized. Thus, they have less opportunity to overlap delays on the probe relations.

One lesson from the preceding discussion is that if multiple sources are likely to have multiple delays, then policies that open more remote sources have a better opportunity to hide delay. Figure 4 shows the performance of the non-pipelined policies when only relation A (the leftmost relation) is subject to delays. In this case, beyond a delay probability of 30% the three policies perform roughly the same. This is because at this point, the policies are all able to complete their scrambling work within the delay of A (30% delay for A is 58.5 seconds) so their performance becomes roughly identical as delay is further increased. Prior to this point, *All-Builds* has a slight advantage over *Materialize-All* because it does not materializes any probe relations, so it does not incur disk contention.

5.1.3 Single-Delay Experiments

The multiple-delay experiments described in the previous section showed that the opening of multiple remote sources can improve the performance of scrambling policies by maximizing the potential to overlap delay. In this set of experiments 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 relation A. The x-axis on the graphs shows this 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.

Balanced Network Figure 5 shows the performance of the non-pipelined policies using the same network speed as used in the previous experiments (i.e., 5 Mbps, roughly equivalent to the observed speed of the disk). In this case we see that all three scrambling policies are able to hide much of the delay up to 80%

of the normal execution time (36.6 seconds) at which point they increase linearly with the delay. Prior to 80%, the *All-Builds* policy performs the best. Its performance has three stages: up until a delay of 20% its response time increases, it then remains flat until 60%, after which it increases linearly with the delay. During the first stage, it is materializing the non-delayed build relations (C, E, and G). If at anytime during this materialization the arrival of tuples from A resumes, then scrambling is suspended, and control returns to normal operation. Once all the scrambling materializations have been performed, the slowdown flattens out for higher delays because the joins can be run, and the work to perform the joins effectively hides the delay of A at that point.[6] If the delay of A is large enough so that all the joins that do not involve A are completed, then beyond that point the slowdown grows linearly with the delay, as there is no more work for scrambling to do.

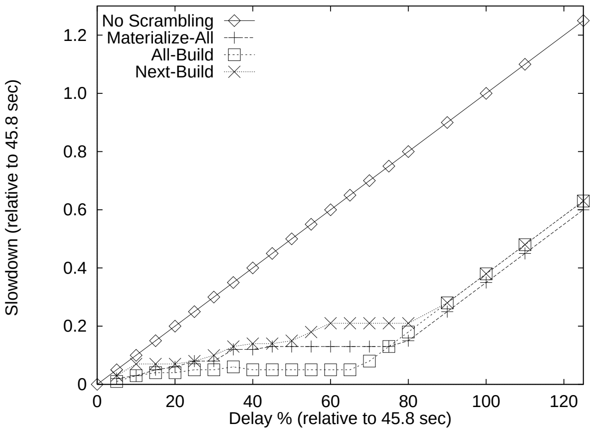

Figure 5: Slowdown, Initial Delay on A
Net: 5 Mbps, Delay % of 45.8 sec

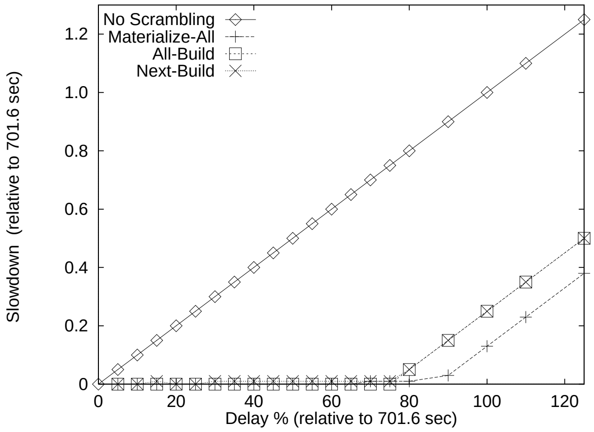

Figure 6: Slowdown, Initial Delay on A
Net: 0.1 Mbps, Delay % of 701.6 sec

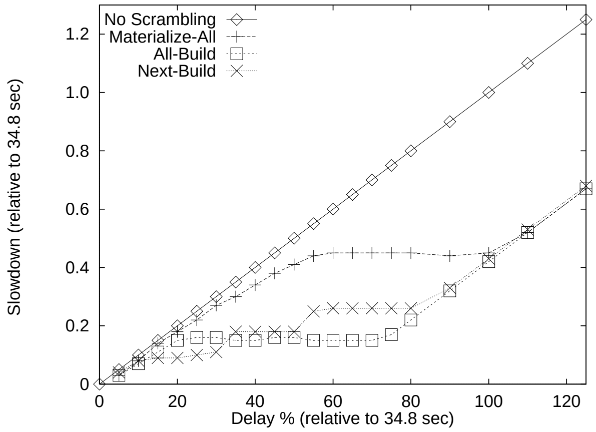

Figure 7: Slowdown, Initial Delay on A
Net: 100 Mbps, Delay % of 34.8 sec

[6] In these experiments, for *All-Builds* the build relations tend to become completely materialized all at about the same time, because they are the same size, and they are not delayed, and they are materialized in parallel. In a real environment, it is expected that they would arrive at different times.

The *Materialize-All* policy has similar behavior to *All-Builds*, but it performs worse because it materializes probe relations in addition to build relations, even though none of the probe relations incur delay. As seen in the previous experiments, the reading of the probe relations from disk introduces disk contention with the writes of the join results, so the overall performance suffers. It is interesting to note in the linear portion of the graph (beyond 80%) *Materialize-All* has a slight advantage over the others because it has already materialized relation B at that point, while the others wait until all tuples of A have arrived before requesting tuples from B. Finally, the *Next-Build* policy performs slightly worse than the others because it requests all remote relations one-at-a-time. Materializing relations in parallel provides a small benefit even if they are not abnormally delayed because the parallel requests to the remote data sources allows them to do some work (e.g., local I/O) in parallel with each other.

Varying Network Speed Many of the results in the previous experiments were driven to some extent by the fact that the network speed and observed disk speed were roughly equivalent. This balance makes scrambling materialization a fairly inexpensive technique, as the materialized data can be later read from the disk at about the same speed it could be obtained from the network (in the absence of delays). Thus, trading predictable local disk I/O for avoiding possible future delays from accessing remote sources is a good deal. In this section we show results that were obtained using other network speeds, in order to examine this tradeoff more closely.

Figure 6 shows the performance of the policies in the case of a single, initial delay of relation A, when the network is significantly slower than the disk. In this case, the network speed is 0.1 Mbps.[7] With the slow network, the normal execution time of the query climbs to 701.6 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 (as in the balanced network case), they run out of scrambling work and the slowdown increases linearly with the delay. As in the previous case, *Materialize-All* has a (now larger) advantage beyond that point because it has already materialized relation B.

If a slow network makes local disk I/O virtually free, then a faster network makes local I/O more expensive. Figure 7 shows the performance of the polices in the same workload as Figures 5 and 6, but with the network speed increased to 100 Mbps. In this case, the normal execution time drops to 34.8 seconds, much of which is due to the cost of local work at the query processing site. When disk I/O is expensive, the relative performance of the polices changes dramatically. First, the *Next-Build* policy, which usually had the worst performance of the scrambling policies the cases examined up to now, actually has the best performance with shorter delays (e.g., up to 30%). It performs better because it materializes less data. The step-shape of the curve for *Next-Build* clearly shows when each of the three non-delayed build relations is materialized and then joined with its corresponding probe. More importantly, the *Materialize-All* policy,

[7] This is about 12K Bytes per second, the same order of magnitude as the what can be currently obtained while surfing the Internet with a good connection.

which was typically best up to this point, is in this case relatively expensive. As usual, it materializes the largest amount of data, including both build and probe relations, to the local disk. These materializations, however, combined with the interference of joins results and probe relations result in significant overhead for *Materialize-All* here.

Controlling Overhead The previous experiment (shown in Figure 7) demonstrated that as networks become faster relative to local disk processing query scrambling will have to be more intelligent about its use of local disk resources. Up to this point, however, none of the scrambling policies were seen to perform worse than not scrambling. For example, in Figure 7 the *Materialize-All* policy approaches, but does not exceed the slowdown of the no scrambling policy when the delay is below 40%. The reason that *Materialize-All* does not become worse than no scrambling is because the scrambling policies all *suspend* (cf. Section 2). That is, if the left-most delayed relation resumes sending tuples, scrambling is suspended and control is returned back to the resumed operator. This policy limits the extent of the damage that can be caused by scrambling too aggressively.

Figure 8 shows the performance of the policies if the *suspend* decision is replaced by the *ignore* decision. In this case, once the initial delay of A is detected, query scrambling is initiated and run to completion regardless of when A returns. As a result, if the delay of A is short, then the aggressive initiation of materializations as performed by *Materialize-All* (and to a lesser extent by the others) can result in worse performance than simply waiting for A to return.

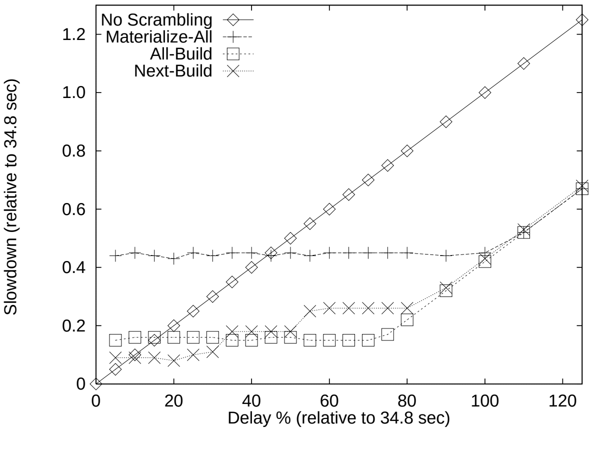

Figure 8: Slowdown, Initial Delay on A
Net: 100 Mbps, *ignore*, Timer: 10x

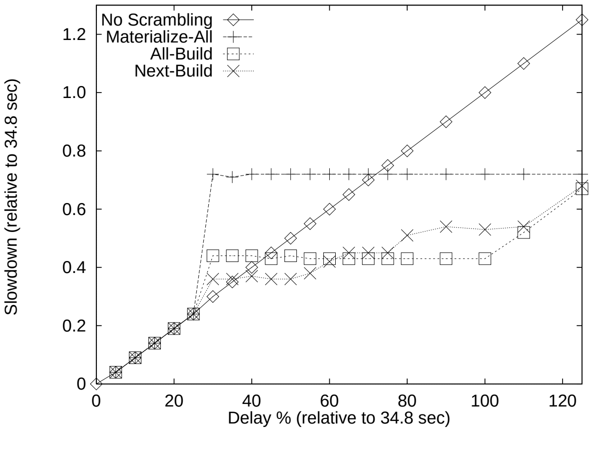

Figure 9: Slowdown, Initial Delay on A
Net: 100 Mbps, *ignore*, Timer: 500x

At first, it might appear that the problems encountered by the scrambling algorithms were caused by having a timer that was set at too short an interval. Figure 9 shows the performance impact of increasing the timer by a factor of 50. The net result of increasing the timer is that there is a longer amount of delay for which scrambling remains inactive, after which it is subject to the problems identified in the Figure 8. Thus, a more effective approach to scrambling is to use a fairly short timer, in order to allow scrambling to hide more delay, but to introduce regulation mechanisms such as *suspend*, in order to ensure that scrambling does not harm performance. Such considerations will become increasingly important as high-speed network access becomes more prevalent.

5.2 Pipelined Performance

In this set of experiments, we evaluate the performance of query scrambling for an execution model that pipelines hash joins. On an *open()* call, the hash join materializes its left child operator into a hash table. On each *next()*, the hash join continues to access tuples from its right child operator and joining them with the hash table until a matching tuple is generated. The matching tuple is returned as the result of the *next()* call. Pipelining is a typical way to perform joins. Pipeline execution requires more memory than non-pipelined execution and more choices are available for scrambling. Each time an operator is scrambled, sufficient memory must be available for the scrambled operator. If no memory is available, in our experiments, the operator is not scrambled. In this way we avoid the thrashing of memory.

5.2.1 Policies

We use three different policies to evaluate the performance of scrambling in a pipelined environment. These policies are *Materialize-Leaves*, *Next-Leaf* and *Pipe-MRS*. The last policy uses a notion developed in [AFTU96], that is, the notion of a *Maximal Runnable Subtree* (MRS). A runnable subtree is a subtree in which all the operators are in the **Not Started** state. A runnable subtree is maximal if its root is the direct descendant of a **Stalled** operator.

As in the previous set of experiments, we compare those policies to the case where no scrambling is performed on the query. The policies are defined as follows:

Materialize-Leaves. This policy starts simultaneously the materialization of all base relations (the leaves of the query tree) when the first time-out is observed. No other materializations are performed, i.e., no joins are never rescheduled. This policy terminates scrambling once all base relations have been entirely materialized.

Next-Leaf. This policy starts the materialization of the next relation when another relation times-out for the first time. Once a relation is entirely materialized, and if some operators are still stalled, then the policy initiates the materialization of another leaf. As for *Materialize-Leaves*, this policy does not scramble any joins.

Pipe-MRS. This policy initiates the materialization of the root of the next maximal runnable subtree (MRS) when a relation times-out. This policy does not materialize any base relations, but instead, materializes only joins.

5.2.2 Impact of Limited Memory

The general principles of performance demonstrated in the non-pipeline section also apply to pipelined execution. Instead of repeating those results for pipelined execution, we focus instead on the impact of limited memory. Each experiment varies the amount of memory available at the client. The minimum memory required is 3 hash tables of 250 pages each (750 pages) plus an addition 10 pages for miscellaneous staging buffers. No query can execute, regardless of its policy, with less memory. We increase the amount of memory for each experiment by 260 pages. Additional memory permits more pipelines to execute simultaneously. At a memory size of 1820 pages, the behavior of pipelined scrambling is no longer limited by memory, so any experiment with more memory will perform exactly the same way.

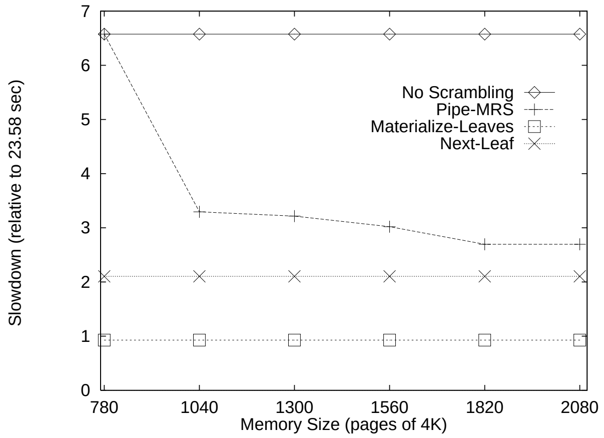

Figure 10: Slowdown, Delay 10% on All Relations
Net: 5 Mbps, Delay: 0.78 sec (3x Timer)

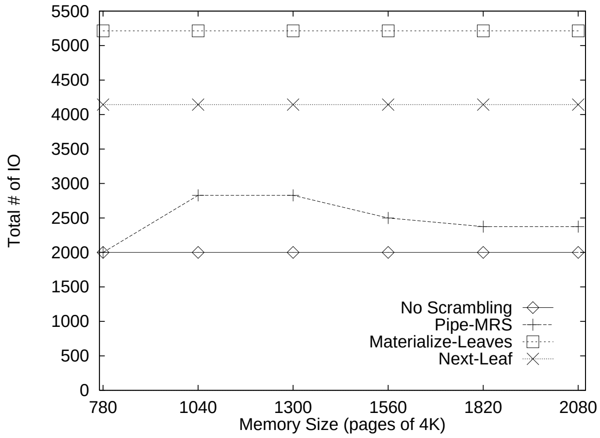

Figure 11: Total IO, Delay 10% on All Relations
Net: 5 Mbps, Delay: 0.78 sec (3x Timer)

Figure 10 shows the relative performance of the three policies and the non-scrambled query when all base relations are subject to random delays at a 10% probability throughout the entire execution of a query. In this case, the response time of the unscrambled query is independent of the memory size, because in this experiment there is always enough memory to execute the original query.

In addition to the memory needed to complete the query, *Materialize-Leaves* and *Next-Leaf* need the same amount of memory as the original query, plus a few additional pages for staging tuples from materialized base relations out to disk. Consequently, they are able to execute in all cases of memory size with the same performance and always consume a fixed amount of memory. Not surprisingly, the scrambling policy that maximizes the overlap between delays (i.e., policy *Materialize-Leaves*) performs the best. *Next-Leaf* overlaps less delay, since relations are materialized one at a time. As such, it performs slightly worse than *Materialize-Leaves*. The performance of *Materialize-Leaves* and *Next-Leaf* are to be expected: these policies are similar in spirit to the ones evaluated during the no-pipeline experiments, and as such highlight the same tradeoffs.

In contrast, the performance of *Pipe-MRS* improves with additional memory. The fundamental reason is that more concurrent MRSs can be rescheduled as the amount of available memory increases. With a memory size of 780 pages, the non-scrambled query and *Pipe-MRS* have the same performance. With this setting, the scheduler is never able to reschedule a subtree when a relation times out because there is not enough memory for any additional hash-tables. With 1040 pages, the scheduler is able to initiate the materialization of the join of C and D while the query tree stalls on relations A and B, and with 1820 pages, *Pipe-MRS* is able to activate all MRSs in the tree (three in this case). As expected, increasing the number of concurrent MRSs tends to augment the overlapping of delays, thus reducing the response time.

Figure 11 shows the I/O performance for the same experiment. Comparing this figure to the previous figure, we see that better policies also perform additional I/O, which is expected. The surprising aspect of this graph is the rise and *drop* in I/Os for *Pipe-MRS*. From 780 page to 1040 pages, the rise is due to the additional simultaneous execution of the join between C and D. The result of this join is written to disk. As memory increases again, the next maximal runnable subtree is the parent of the join between E and F. Now, two effects combine to slow the execution of each pipeline. First, since they execute in parallel, they share resources. Second, a larger subtrees takes more time to materialize data on the disk since a larger part of the entire tree must execute before the materialization operator starts issuing I/O. Since the pipelines are slowed by these effects, the normal, unscrambled execution of the query *catches up* with the scrambled pipelines. When normal execution catches up, the scrambling materializations are *stopped*, thus saving I/O!

6 Related Work

Network congestion, network link failure, server load, and temporary server unavailability all introduce unexpected delay in the accessing of remote sources. The techniques that attack this problem fall into two broad categories: the *proactive* and *reactive*.

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. At query start-up time, the plan is fixed, and query execution corresponds exactly to the plan.

The Volcano optimizer [CG94, Gra93] provides a framework for the application of proactive techniques for distributed query processing. During optimization, if a cost comparison returns *incomparable*, the choice for that part of the search space is encoded in a *choose-plan* operator. All decisions regarding query execution are then made final at query start-up time.

HERMES [ACPS96] uses a proactive technique for recording the costs of previous calls to remote sources (in addition to caching the results) and can use resulting history of costs to estimate the cost of new calls. As in Volcano, this system optimizes a query both at query compile and query start-up time, but does not change the query plan during query run-time.

Mariposa [SAL+96] bases the optimization of distributed queries on an economic paradigm. Although the query optimizer of [SAL+96] adopts a radical approach since it is first based on negotiation and second it is not based on costs, optimization still builds a plan that is fixed for the duration of the execution of the query.

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.

In the bursty data arrival environment, such as the Internet, existing techniques have several problems. Proactive techniques are limited because the history of query execution poorly predicts future query performance. The primary problem with existing reactive techniques is the *weight* of monitoring and operations. We classify reactive techniques as *heavyweight* if the unit of monitoring or operation is large, e.g. a join. Heavyweight reactive techniques also perform poorly in our environment since delays are not quickly detected.

In [AFTU96], we developed a collection of reactive techniques that both rescheduled operators and incrementally reorganized the query execution plan. In this paper, we extend query scrambling to deal with the bursty arrival environment by exploring *lightweight* reactive techniques, namely where the unit of monitoring and operation is small, e.g. less than a join. Lightweight query scrambling constantly monitors the execution of the query with a granularity a single data page access and it also monitors the behavior of any changes introduced into execution. Additionally, if necessary, only small changes in computation may occur with query scrambling, again at the level of a single data page access. Thus, query scrambling adapts quickly to the changing properties of the environment.

7 Conclusion

To address the issue of unpredictable delays in the wide-area environment, we have developed a class of techniques for query execution, called *query scrambling*. Query scrambling monitors query execution and reacts to unexpected delays by *on-the-fly* rescheduling the operations of a query during its execution.

In this paper we explored the key questions to query scrambling: when should scrambling start, what should be scrambled, and when should scrambling stop. We examined several sets of *policies* to control scrambling and described the architecture of a run-time scheduler that is capable of implementing these policies. We then used a detailed simulation of a run-time system in order to examine the performance tradeoffs of the various scrambling policies under both pipelined and non-pipelined execution models.

Our results show that query scrambling can in most cases *hide* a significant portion of the delay experienced by a query, i.e. the user does not experience any delay in the processing of a query. In some cases all the delay can be hidden at essentially negligible additional cost to query execution. In addition, since query scrambling introduces parallel access to remote sources into query execution, all the performance gains thereof also occur.

For example, we show that if many sources exhibit bursty arrival, then the overlap of delay is the most important consideration, regardless of network speed, policies or join operator. If network speed is slow relative to the disk and delays are long relative to disk, then materializing the results of remote source to local disk is very effective. If network speed is fast relative to disk (either because of point-to-point network or gigabit technology), then the bottleneck moves from the network to the disk, and thus materializing the results performs less well since it aggravates the bottleneck.

In some cases, improper use of scrambling can introduce network congestion and thrashing during query processing. We carefully documented the cases where these situations occur and show, both in terms of architecture and in terms of implementation policies, how to avoid them.

We showed that scrambling under pipelined execution models works well, however pipelined execution requires the reservation of memory to perform well. Multiple scrambled pipelines compete for memory resources, potentially introducing thrashing. We showed how to avoid this behavior by limiting the number of simultaneously scrambling pipelines.

For future work, we intend to build a run-time system which *continuously* scrambles and *throttles* the behavior of query execution to balance trade-offs between performance gains and performance losses.

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 the draft version of this paper. We would also like to thank Dennis Shasha for discussions on query scrambling.

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.

[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 Distribution 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 Winsconsin, 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 *CIKM*, 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.

[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.