Scrambling Query Plans to Cope With Unexpected Delays

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

Tolga Urhan[†]
University of Maryland
urhan@cs.umd.edu

Technical Report CS-TR-3645 and UMIACS-TR-96-35

Abstract

Accessing numerous widely-distributed data sources poses significant new challenges for query optimization and execution. Congestion or failure in the network introduce highly-variable response times for wide-area data access. This paper is an initial exploration of solutions to this variability. We investigate a class of dynamic, run-time query plan modification techniques that we call *query plan scrambling*. We present an algorithm which modifies execution plans *on-the-fly* in response to unexpected delays in data access. The algorithm both *reschedules* operators and *introduces* new operators into the plan. We present simulation results that show how our technique effectively hides delays in receiving the initial requested tuples from remote data sources.

1 Introduction

Ongoing improvements in networking technology and infrastructure have resulted in a dramatic increase in the demand for accessing and collating data from disparate, remote data sources over wide-area networks such as the Internet and intranets. Query optimization and execution strategies have long been studied in centralized, parallel, and tightly-coupled distributed environments. Data access across widely-distributed sources, however, imposes significant new challenges for query optimization and execution for two reasons: First, there are semantic and performance problems that arise due to the *heterogeneous* nature of the data sources in a loosely-coupled environment. Second, 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. From the end user's point of view, congestion or failure in any of the components of the network are manifested as *highly-variable response time* — that is, the time required for obtaining data from remote

[*]Laurent Amsaleg is supported by a post-doctoral fellowship from INRIA Rocquencourt, France.
[†]Supported in part by NSF Grant IRI-94-09575, an IBM SUR award, and a grant from Bellcore.

sources can vary greatly depending on the specific data sources accessed and the current state of the network at the time that such access is attempted.

The query processing problems resulting from heterogeneity have been the subject of much attention in recent years (e.g., [SAD$^+$95, BE96, TRV96]). In contrast, the impact of unpredictable response time on wide-area query processing has received relatively little attention. The work presented here is an initial exploration into addressing problems of response-time variability for wide-area data access.

1.1 Response Time Variability

High variability makes efficient query processing difficult because query 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 (i.e., disk and/or network accesses). The causes of high-variability are typically failures and congestion, which are inherently runtime issues; they cannot be reliably predicted at query optimization time or even at query start-up time. As a result, the execution of a statically optimized query plan is likely to be sub-optimal in the presence of unexpected response time problems. 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 different types of response time problems that can be experienced in a loosely-coupled, wide-area environment can be categorized as follows:

- **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, of network congestion, resource contention at the source, or because a different (slower) communication path is being used (e.g., due to a 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.

Because these problems can arise unpredictably at runtime, they cannot be effectively addressed by static query optimization techniques. As a result, we have been investigating a class of dynamic, runtime query plan modification techniques that we call *query plan scrambling*. In this approach, a query is initially executed according to the plan and associated schedule generated by the query optimizer. If however, a significant unexpected problem arises during the execution, then query plan scrambling is invoked to modify the execution *on-the-fly*, so that progress to be made on other parts of the plan. In other words, rather than simply stalling for slowly arriving data, query plan scrambling attempts to *hide* unexpected delays by performing other useful work.

There are three possible ways that query plan scrambling can be used to help mask response time problems. First, scrambling allows useful work to be done in the hope that the cause of the problem is resolved in the meantime. This approach is useful for all three classes of problems described above. Second, if data are arriving, but at a rate that hampers query processing performance (e.g., in the Slow Delivery or Bursty Arrival cases), then scrambling allows useful work to be performed while the problematic data are obtained in a background fashion. Finally, in cases where data are simply not arriving, or are arriving far too slowly, then scrambling can be used to produce partial results that can then be returned to users and/or used in query processing at a later time [TRV96].

1.2 Tolerating Initial Delays

In this work, we present an initial approach to query plan scrambling that specifically addresses the problem of Initial Delay (i.e., delay in receiving the initial requested tuples from a remote data source). We describe and analyze a query plan scrambling algorithm that follows the first approach outlined above; namely, other useful work is performed in the hope that the problem will eventually be resolved, and the requested data will arrive at or near the expected rate from then on.

In order to allow us to clearly define the algorithm and to study its performance, this work assumes an execution environment with several properties. These are:

- Our algorithm addresses only response time delays in receiving the initial requested tuples from remote data sources. Once the initial delay is over, tuples are assumed to arrive at or near the originally expected rate. As stated previously, this type of delay models problems in connecting to remote data sources, as is often experienced in the Internet.

- We focus on query processing using a data-shipping or hybrid-shipping approach [FJK96], where data is collected from remote sources and integrated at the query source. That is, only query processing that is performed at the query source is subject to scrambling. This approach is typical of mediated database systems that integrate data from distributed, heterogeneous sources, e.g., [TRV96].

- Query execution is scheduled using an iterator model [Gra93]. In this model every run-time operator supports an *open()* call and a *get-next()* call. Query execution starts by calling open() on the top most operator of the query execution plan and proceeds by iteratively calling get-next() on the top most operator. These calls are propagated down the tree; each time an operator needs to consume data, it calls get-next() on its child (or children) operator(s). The iterator model imposes a schedule on the operators in the query plan.

Query plan scrambling is related to approaches for dynamic and/or parametric query optimization, and in contingency plan generation for real-time database systems. In Section 4 we outline related work and compare and contrast our approach to the literature. In short, our work has the following characteristics:

- It explicitly accounts for variations in response time of a source.
- It exploits, where possible, decisions made by the static query optimizer.
- It improves the response time of a broad range of plans and run-time scenarios.
- It imposes no performance overhead in the absence of unexpected delays.

Our work requires significant changes to the run-time system of a distributed heterogeneous database to allow for dynamic modification of query plans and the introduction of new operators. We believe these challenges are a fruitful area of future research.

The paper is organized as follows. Section 2 describes the algorithm and gives an extended example. Section 3 presents results from a simulation study that demonstrate the properties of the algorithm. Section 4 describes related work. Section 5 concludes with a summary of the results and a discussion of future work.

2 Scrambling Query Plans

This section describes our algorithm for scrambling queries to cope with initial delays in obtaining data from remote data sources. The algorithm consists of two phases — one that changes the execution order of operations in order to avoid idling, and one that synthesizes new operations to execute in the absence of other work to perform. We first provide a brief overview of the algorithm and then describe the two phases in detail using a running example. The algorithm is then summarized at the end of the section.

2.1 Algorithm Overview

Figure 1 shows an operator tree for a complex query plan. At the leaves of the tree are base relations. These are assumed to be stored at remote sites. The nodes of the tree are binary operators (i.e., joins) that are executed at the query source site.[1]

[1]Unary operators, such as selections, sorting, and partitioning are not shown in the figure.

As discussed previously, we describe the scrambling algorithm in the context of an iterator-based execution model. This model imposes a schedule on the operators of a query. In the figure, the joins are numbered according to the order in which they would be completed by an iterator-based scheduler. Typically, the plan for such a complicated query would be generated by a static query optimizer according to its cost model, statistics, and objective functions. Thus, the ordering of the operations in the initial plan is in some sense a "good" ordering, and should be preserved wherever possible.

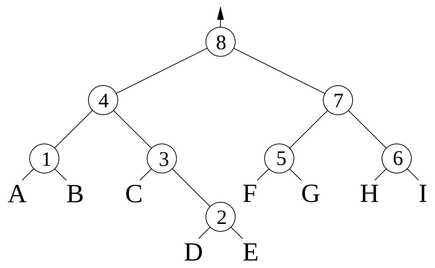

Figure 1: Initial Query Tree

The schedule implied by the tree in Figure 1 would begin by materializing the left subtree of the root node. Assuming that hash joins are used and that there is sufficient memory to hold the hash tables (so no partitioning is necessary), this materialization would consist of the following steps:

1. Scan relation A and build hash-table H_A;
2. In a pipelined fashion, probe H_A with tuples of B and build a hash-table containing the result of A⋈B (H_{AB});
3. Scan C and build hash-table H_C;
4. Scan D and build hash-table H_D;
5. In a pipelined fashion, probe H_D, H_C and H_{AB} with tuples of E and build a hash-table containing the result of (A⋈B)⋈(C⋈D⋈E).

The execution thus begins by requesting tuples of relation A from a remote site. If there is a delay in accessing that site, (say, due to that site being temporarily down), then the scan of A (i.e., step 1) is blocked until the site recovers. Under a traditional iterator-based scheduling discipline, this delay of A would result in the entire execution of the query being blocked, pending the recovery of the remote site.

Given that unexpected delays are highly probable in a wide-area environment, such sensitivity to delays is likely to result in unacceptable performance. Our scrambling algorithm addresses this problem by attempting to *hide* such delays by making progress on other parts of the query until the problem is resolved. The scrambling algorithm is invoked once a delayed relation is detected (e.g., via a timeout mechanism). The algorithm is iterative; during each iteration it selects part of the plan to execute and materializes the corresponding temporary results to be used later in the execution.

The scrambling algorithm executes in one of two phases. During *Phase 1*, each iteration modifies the schedule in order to execute operators that are not dependent on any data that is known to be delayed. For example, in the query of Figure 1, Phase 1 might result in materializing the join of relations C, D and E while waiting for the arrival of A. During *Phase 2*, each iteration synthesizes new operators (joins for example) in order to make further progress. In our example, a Phase 2 iteration might choose to join relation B with the result of C$\bowtie$D$\bowtie$E as computed previously.

At the end of each iteration the algorithm checks to see if any delayed sources have begun to respond, and if so, it stops iterating and returns to normal scheduling of operators, possibly re-invoking scrambling if additional delayed relations are later detected. If, however, no delayed data has arrived during an iteration, then the algorithm iterates again. The algorithm moves from Phase 1 to Phase 2 when it fails to find an existing operator that is not dependent on a delayed relation. If, while in Phase 2, the algorithm is unable to create any new operators, then scrambling terminates and the query simply waits for the delayed data to arrive. In the following sections we describe, in detail, the two phases of scrambling and their interactions.

2.2 Phase 1: Materializing Maximal Runnable Subtrees

2.2.1 Definitions of Blocked and Runnable Operators

A query is represented by a tree of query operators that have producer-consumer relationships. Given an operator, its ancestor operator is the operator that consumes its tuples. Conversely, its descendant operators are the ones that produce the tuples it consumes. The producer-consumer relationships create *execution dependencies* between operators, as one operator can not consume tuples before these tuples have been produced. For example, a select operator can not consume tuples if the corresponding base relation is not available. In that case, the select operator is blocked. If the select can not consume any tuple, it can not produce any tuple. Consequently, the consumer of the select is also blocked. By transitivity, all the ancestors of the unavailable relation are blocked.

When the system discovers that a relation is unavailable, query plan scrambling is invoked. Scrambling starts by splitting the operators of the query tree into two disjoint queues: a queue of *blocked operators* and a queue of *runnable operators*. These queues are defined as follows:

Definition 2.1 Queue of Blocked Operators: Given a query tree, the queue of blocked operators contains all the ancestors of each unavailable relation. □

Definition 2.2 Queue of Runnable Operators: Given a query tree and a queue of blocked operators, the queue of runnable operators contains all the operators that are not in the queue of blocked operators. □

Operators are inserted in the runnable and blocked queues according to the order in which they would be executed by an iterator-based scheduler. This order corresponds to a depth-first traversal of the query tree.

2.2.2 Definition of a Maximal Runnable Subtree

Each iteration during Phase 1 of query plan scrambling analyzes the runnable queue in order to find a *maximal runnable subtree* to materialize. A maximal runnable subtree is defined as follows:

> **Definition 2.3 Maximal Runnable Subtree:** Given the query tree and the queues of blocked and runnable operators, a runnable subtree is a subtree in which all the operators are runnable. A runnable subtree is maximal if its root is the first runnable descendant of a blocked operator. □

None of the operators belonging to a maximal runnable subtree depend on data that is known to be delayed.[2] Each iteration initiates the materialization of the first maximal runnable subtree found. This materialization completes only if no relations used by this subtree are discovered to be unavailable during the execution. When the execution of a runnable subtree is finished and its result materialized, the algorithm removes all the operators belonging to that subtree from the runnable queue. It then checks if missing data have begun to arrive. If the missing data are still unavailable, the first phase of query plan scrambling initiates another iteration. The new iteration analyzes (again) the runnable queue to find the next maximal runnable subtree to materialize.

2.2.3 Runnable Subtrees and Data Unavailability

It is possible that during the execution of a runnable subtree, one of the participating base relations is discovered to be unavailable. This is because a maximal runnable subtree is defined with respect to the *current* contents of the blocked and runnable queues. The runnable queue is only a guess about the real availability of relations. When the algorithm inserts operators in the runnable queue, it does not know if their associated relations are actually available or unavailable. This will be discovered only when the corresponding relations are requested.

In the case where a relation is discovered to be unavailable during the execution of a runnable subtree, the current iteration finishes and the algorithm updates the runnable and blocked queues. All the ancestors of the unavailable relation are extracted from the runnable queue and inserted in the blocked queue. Once the queues are updated, the scrambling of the query plan initiates a new Phase 1 iteration in order to materialize another maximal runnable subtree.

[2]Note that in the remainder of this paper, we use "maximal runnable subtree" and "runnable subtree" interchangeably, except where explicitly noted.

2.2.4 Termination of the First Phase

At the end of each iteration, the algorithm checks for data arrival. If it is discovered that an unavailable relation has begun to arrive, the algorithm updates the blocked and runnable queues. The ancestors of the relation that is unblocked are extracted from the blocked queue and inserted in the runnable queue. Note that any ancestors of the unblocked relation that also depend on other blocked relations are not extracted from the queue. The first phase then terminates and the execution of the query returns to normal scheduling of operators. Note that if no further relations are blocked, the execution of the query will proceed until the final result is returned to the user. The scrambling algorithm will be re-invoked, however, if the query execution blocks again.

Phase 1 also terminates if the runnable queue is empty. In this case, the first phase can not perform any other iteration because all remaining operators are blocked. When this happens, query plan scrambling switches to its second phase. The purpose of the second phase is to process the available relations when all the operators of the query tree are blocked. We present the second phase of query plan scrambling in Section 2.3. First, however, we present an example that illustrates all the facets of Phase 1 described above.

2.2.5 A Running Example

This example reuses the complex query tree presented at the beginning of Section 2. To illustrate the behavior of the first phase, we follow the scenario given below:

1. When the execution of the query starts, relation A is discovered to be unavailable.
2. During the third iteration, relation G is discovered to be unavailable.
3. The tuples of A show up at the query execution site at the end of the fourth iteration.
4. At the time the first phase terminates, no tuples of G have been received.

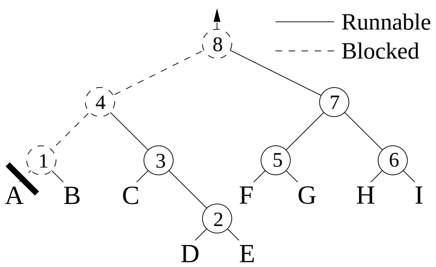

Figure 2: Blocked and Runnable Operators with Relation A Unavailable

The execution of the query begins by requesting tuples of relation A from a remote site. Following the above scenario, we assume relation A is unavailable. In Figure 2, the thick solid line indicates that relation A is unavailable. The operators that are blocked by the delay of A are depicted using a dashed line.

The unavailability of A invokes the first phase of query scrambling which updates the blocked and runnable queues and initiates the first iteration. This iteration analyses the runnable queue and finds that the first[3] maximal runnable subtree consists of a unary operator to partition relation B. Once the operator is materialized (i.e., B is partitioned), the algorithm checks for the arrival of the tuples of A. Following the above scenario, we assume that the tuples of A are still unavailable, so another iteration is initiated. This second iteration finds the next maximal runnable subtree to be the one rooted at operator 3. Note the subtree rooted at operator 2 is *not* maximal since its consumer (operator 3) is not blocked.

Figure 3 shows the materialization of the runnable subtrees found by the first two iterations of query scrambling. Part (a) of this figure shows the effect of materializing of the first runnable subtree: the base relation B is now partitioned into **B'**. It also shows the second runnable subtree (indicated by the shaded grey area). Figure 3(b) shows the query tree after the materialization of this second runnable subtree. The materialized result is called **X1**.

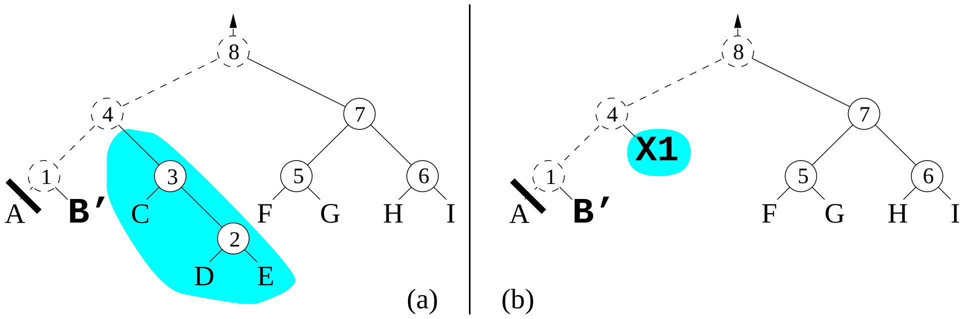

Figure 3: Query Tree Before and After the Execution of the Runnable Subtree Rooted at Op. 3

Once **X1** is materialized, another iteration starts since, in this example, relation A is still unavailable. The third iteration finds the next runnable subtree which is the join F, G, H and I, rooted at operator 7. The execution of this runnable subtree starts by building the left input of operator 5 (partitioning F into **F'**). It then requests relation G in order to probe the tuples of F. In this scenario, however, G is discovered to be unavailable, triggering the update of the blocked and runnable queues. Figure 4(a) shows that operators 5 and 7 are newly blocked operators (operator 8 was already blocked due to the unavailability of A). Once the queues of operators are updated, another iteration of scrambling is initiated to run the next runnable subtree, i.e., the one rooted at operator 6 (indicated by the shaded grey area in the figure). The result of this subtree execution is called **X2**.

Figure 5 illustrates the next step in the scenario, i.e., it illustrates the case where just after **X2** is materialized it is discovered that the tuples of relation A have begun to arrive. In this case, the

[3] As stated earlier, operators are inserted into the queues with respect to their execution order.

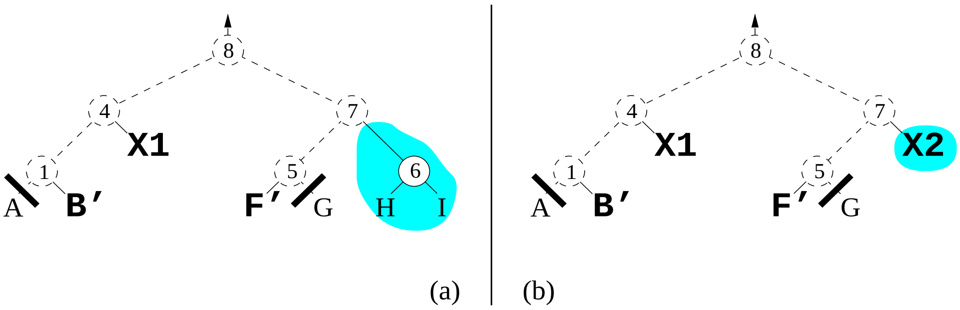

Figure 4: Unavailability of Relation G and Materialization of X2

algorithm updates the runnable and blocked queues. As shown in Figure 5(a), operators 1 and 4 that were previously blocked are now unblocked (operator 8 remains blocked however). The first phase then terminates and returns to the normal scheduling of operators which materializes the left subtree of the root node (see Figure 5(b)). The resulting relation is called **X3**.

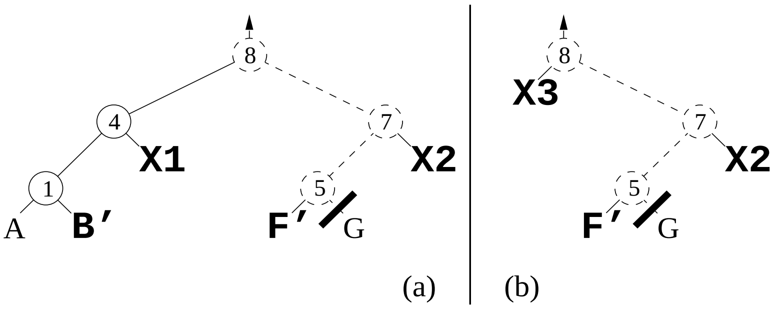

Figure 5: Relation A is Available

After **X3** is materialized, the query is blocked so the first phase of query scrambling is re-invoked. The first phase computes the new contents of the runnable and blocked queue. Once this done, it discovers that the runnable queue is empty since all remaining operators are ancestors of G, which is not available. As such, no more iterations can be initiated. The first phase terminates and the scrambling of the query plan enters Phase 2. We describe Phase 2 of the algorithm in the next section.

2.3 Phase 2: Creating New Joins

Scrambling moves into its second phase when the runnable queue is empty but the blocked queue is not. The goal of the second phase is to *create* new operators to be executed. Specifically, the second phase creates joins between relations that were not directly joined in the original query tree, but whose consumers are blocked (i.e., in the blocked queue) due to the unavailability of some other data.

In contrast to Phase 1 iterations, which simply adjust scheduling to allow *runnable* operators to execute, iterations during the second phase of scrambling actually create new joins. Because the operations that are created during the second phase were not chosen by the query optimizer when the original query plan was generated, it is possible that these operations may entail a significant amount of additional work. If the joins created and executed by the second phase are too expensive, query scrambling could result in a net degradation in performance. Thus, unlike Phase 1, where the potential overhead due to scrambling is low, Phase 2 has the potential to negate or even reverse the benefits of scrambling if care is not taken.

One way to ensure that Phase 2 does not generate overly expensive joins is to involve the query optimizer in the choice of new joins to execute. While such involvement may be possible in certain architectures, the use of the optimizer complicates the scrambling algorithm and adds potential overhead. In this paper, therefore, we instead use the simple heuristic of avoiding Cartesian products. In Section 3, we analyze the performance impact of varying the cost of created joins relative to the cost of the joins in the original query plan.

2.3.1 Creating New Joins

At the start of the second phase, the scrambling algorithm constructs a graph $\mathcal{G}$ of possible joins. Each node in $\mathcal{G}$ corresponds to a relation, and each edge in $\mathcal{G}$ indicates that the two connected nodes have common join attributes, and thus can be joined without causing a Cartesian product. There are two types of relations that may appear in $\mathcal{G}$: 1) base relations from the original query that have not yet been accessed (i.e., direct inputs of blocked binary operators); and 2) temporary results that were materialized earlier in the scrambling process. Unavailable relations are not placed into $\mathcal{G}$.

Once $\mathcal{G}$ is constructed, the second phase starts to iteratively create and execute new join operators. Each iteration of the second phase performs the following steps:

1. In $\mathcal{G}$, find the two *leftmost* joinable (i.e., connected) relations i and j. The notion of *leftmost* is with respect to the order in the query plan, or equivalently, to the order of the consumer operators of the relations in the blocked queue. If there are no joinable relations in $\mathcal{G}$, then terminate scrambling.
2. Create a new join operator $i \bowtie j$.
3. Attempt to materialize $i \bowtie j$. If either of i or j is found to be unavailable then remove the unavailable relations from $\mathcal{G}$, stop the current iteration and begin a new iteration.[4]
4. If the materialization completes, update $\mathcal{G}$ by replacing i and j with the materialized result of $i \bowtie j$. Update the runnable and blocked queues.

[4]Note that blocking at this point is unlikely as most relations will be materialized results, and hence, stored at the query source.

5. Test to see if any unavailable data has arrived. If so, then terminate scrambling, else begin a new iteration.

Figure 6 demonstrates the behavior of Phase 2 by continuing the example of the previous section. The figure is divided into three parts. Part (a) shows the query tree at the end of Phase 1. In this case, $\mathcal{G}$ would contain **F'**, **X2**, and **X3**. Assume that, in $\mathcal{G}$, relations **F'** and **X2** are directly connected but relation **X3** is not connected to either (i.e., assume it shares join attributes only with the unavailable relation G). In this example, therefore, **F'** and **X2** are the two leftmost joinable relations; **X3** is the leftmost relation, but it is not joinable.

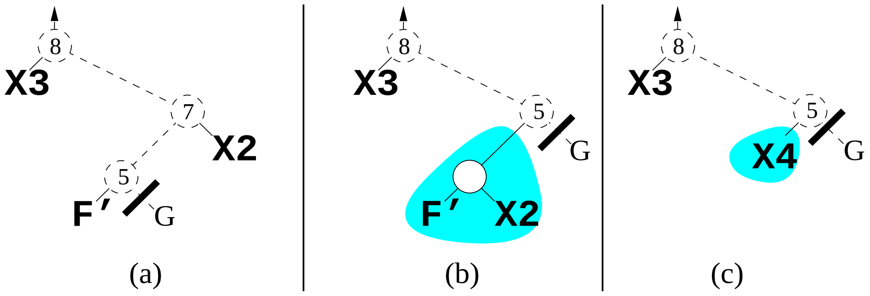

Figure 6: Performing a New Join that Produces X4

Figure 6(b) shows the creation of the new join of **F'** and **X2**. The creation of this join requires the removal of join number 7 from the blocked queue and its replacement in the ordering of execution by join number 5. Finally, Figure 6(c) shows the materialization of the created operator. The materialized join is called **X4**. At this point, $\mathcal{G}$ is modified by removing **F'** and **X2** and inserting **X4**, which is not joinable with **X3**, the only other relation in $\mathcal{G}$.

2.3.2 Termination of the Second Phase

After each iteration of the second phase, the number of relations in $\mathcal{G}$ is reduced. Phase 2 terminates if $\mathcal{G}$ is reduced to a single relation, or if there are multiple relations but none that are joinable. As shown in the preceding example, this latter situation can arise if the attribute(s) required to join the remaining relations are contained in an unavailable relation (in this case, relation G).

The second phase can also terminate due to the arrival of unavailable data. If such data arrive during a Phase 2 iteration, then, at the end of that iteration, the runnable and blocked queues are updated accordingly and the control is returned to the normal scheduling of operators. As mentioned for the first phase, query scrambling may be re-invoked later to cope with other delayed relations.

2.3.3 Physical Properties of Joins

The preceding discussion focused on restructuring *logical* nodes of a query plan. The restructuring of *physical* plans, however, raises additional considerations. First, adding new join operators may require the introduction of additional unary operators to process the inputs of the new join so that the join can be correctly processed. For example, a merge join operator requires that the tuples it consumes are sorted, and thus may require that sort operators be applied to its inputs. Second, deleting operators, as was done in the preceeding example, may also require the addition of unary operators. For example, relations may need to be repartitioned in order to be placed as children of an existing hybrid hash node. Finally, changing the inputs of an existing join operator may also require modifications — if the new inputs are sufficiently different than the original inputs, the physical join operators may have to be modified. For example, an indexed nested loop join might have to be changed to a hash join if the inner relation is replaced by one that is not indexed on the join attribute.

2.4 Summary and Discussion

Our query plan scrambling algorithm can be summarized as follows:

- When a query becomes blocked (because relations are unavailable), the query plan scrambling is initiated. It first computes a queue of blocked operators and another queue of runnable operators. The blocked operators are all the ancestors of the unavailable data.
- The first phase then analyses the queue of runnable operators, picks a maximal runnable subtree and materializes its result. This process is repeated until the queue of runnable operators is empty. At this point, the system switches to the second phase.
- The second phase then tries to create a new operator that joins two relations that are available and joinable. This process is repeated until no more joinable relations can be found.
- After each iteration of the algorithm, it checks to see if any unavailable data have arrived, and if so, control is returned to normal scheduling of operators, otherwise another iteration is performed.

There are two additional issues regarding the algorithm that deserve mention, here. The first issue concerns the knowledge of the actual availability of relations. Instead of discovering, as the algorithm does now, during the execution of the operations performed by each iteration that some sources are unavailable, it is possible to send some or all of the initial data requests to the data sources as soon as the first relation is discovered to be unavailable. Doing so would give the algorithm immediate knowledge of the availability status of all the sources. Fortunately, using the iterator model, opening multiple data sources at once does not force the query execution site to consume all the tuples simultaneously — the iterator model will suspend the flow of tuples until they are consumed by their consumer operators.

The second issue concerns the potential additional work of each phase. As described previously, the first phase materializes existing subtrees that have been optimized prior to runtime by the query optimizer. During the second phase, however, we create new joins from scratch. In this paper, the query optimizer is not involved in the choice of the relations to join. Instead, we use the simple heuristic of avoiding Cartesian products. The advantage of this approach is its simplicity. The disadvantage, however, is the potential overhead caused by the possibly sub-optimal joins. We study the performance impact of varying costs of the created joins in the following section. This issue raises the possibility of integrating scrambling with an existing query optimizer. Such an integration, is one aspect of our future work.

3 Performance

In this section, we examine the main characteristics of the performance of the query scrambling algorithm. The first set of experiments shows the typical performance of any query that is scrambled. The second set of experiments studies the sensitivity of the second phase of query scrambling to the selectivity of the new joins it creates. We first describe the simulation environment used to study our algorithm.

3.1 Simulation Environment

To study the performance of the query scrambling algorithm, we extended an existing simulator [FJK96, DFJ+96] 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 these experiments. More detailed descriptions of the simulator can be found in [FJK96, DFJ+96].

Table 1 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. For the current study, the simulator was configured to model a client-server system consisting of a single client connected to seven 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 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 *DiskPageSize*. The unit of transfer between sites are pages of size *NetPageSize*.

Parameter	Value	Description
NumSites	8	number of sites
Mips	30	CPU speed of a site (10^6 inst/sec)
NumDisks	1	number of disks on a site
DiskPageSize	4096	size of one disk data page (bytes)
NetBw	1	network bandwidth (Mbit/second)
NetPageSize	8192	size of one network data page (bytes)
Compare	4	instructions to apply a predicate
HashInst	25	instructions to hash a tuple
Move	2	instructions to copy 4 bytes
ClientMemSize	Large/Small	size of the memory at the client

Table 1: Simulator Model Parameters and Main Settings

The network is modeled simply as a FIFO queue with a specified bandwidth (*NetBw*); the details of a particular technology (i.e., Ethernet, ATM, etc.) are not modeled. The simulator also charges 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.

In this paper, the simulation is used primarily to demonstrate the properties of our scrambling algorithm, rather than for a detailed analysis of the algorithm. As such, the specific settings used in the simulator are less important than the way in which delay is either hidden or not hidden by the algorithm. In the experiments, the various delays were generated by simply requesting tuples from a "unavailable" source at the end of the various iterations of query plan scrambling. That is, rather than stochastically generating delays, we explicitly imposed a series of delays in order to study the behavior of the algorithm in a controlled manner. For example, to simulate the arrival of blocked tuples during, say, the third iteration of the first phase, we scrambled the query 3 times, and then initiated the transfer of tuples from the "blocked" relation so that the final result could be computed and to be returned to the "user".

3.2 A Query Tree for the Experiments

For all the experiments described in this section, we use the query tree represented in Figure 7. We use this query tree in order to show a case where the work done during the first and the second phases of query scrambling is balanced.

Each base relation has 10,000 tuples of 100 bytes each. We assume that the join graph is fully connected, that is, any relation can be joined with any relation. In the first set of experiment, we study the performance of query plan scrambling in the case where all the joins in this query tree produce the same number of tuple, i.e., 1,000 tuples. As such, 1,000 tuples are returned to the user. In the second set of experiments, however, we study the case where the joins in the query tree have different selectivities and thus produce results of various sizes.

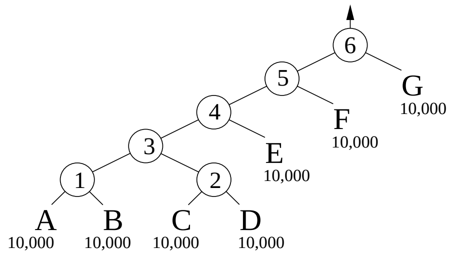

Figure 7: Query Tree Used for the Experiments

For all the experiments, we study the performance of our approach in the case where a single relation is unavailable. This relation is the *left-most* relation, i.e., relation A. Studying the cases where more than one relation is unavailable would not change the basic lessons, of the study. It would, however, reduce the effectiveness of query plan scrambling since fewer iterations of the algorithm could be performed.

3.3 Experiment 1: The Step Phenomenon

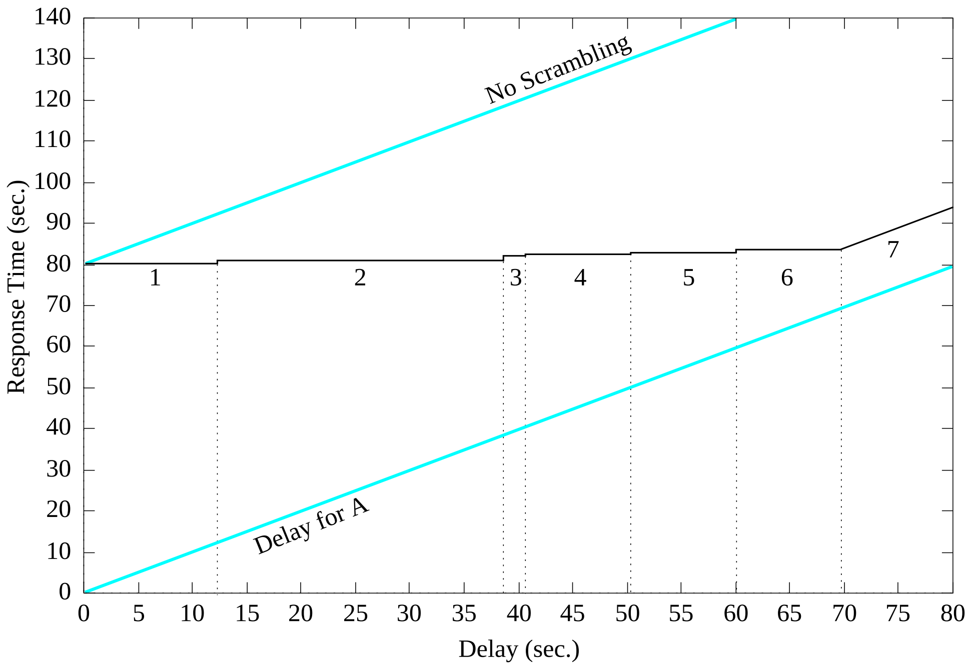

Figure 8: Response Times of Scrambled and Non-Scrambled Queries
(Small Memory, Varying the Delay of A.)

Figure 8 shows the response time for the scrambled query plans that are generated as the delay for relation A (the leftmost relation in the plan) is varied. The delay for A is shown along the X-axis, and is also represented as the lower grey line in the figure. The higher grey line shows the performance of the unscrambled query, that is, if the execution of the query is simply delayed until the tuples of relation A begin to arrive. The distance between these two lines, therefore is constant, and is equal to the response time for the original (unscrambled) query plan, which is 80.03 seconds in this case. In this experiment, the memory size of the query execution site is small. With this setting, the hash-tables for inner relations for joins can not entirely be built in memory and partitioning is required.

The middle line in Figure 8 shows the response time for the scrambled query plans that are executed for various delays of A. In this case, there are six possible scrambled plans that could be generated. As stated in Sections 2.2 and 2.3, the scrambling algorithm is iterative. At the *end of each iteration* it checks to see if delayed data has begun to arrive, and if so, it stops scrambling and normal query execution is resumed. If, however, at the end of the iteration, the delayed data has still not arrived, another iteration of the scrambling algorithm is initiated. The result of this execution model is the step shape that can be observed in Figure 8.

The *width* of each step is equal to the duration of the operations that are performed by the current iteration of the scrambling algorithm, and the *height* of the step is equal to the response time of the query if normal processing is resumed at the end of that iteration. For example, in this experiment, the first scrambling iteration results in the retrieval and partitioning of relation B. This operation requires 12.23 seconds. If at the end of the iteration, tuples of relation A have begun to arrive then no further scrambling is done and normal query execution resumes. The resulting execution in this case, has a response time of 80.10 seconds. As a result, the first step shown in Figure 8 has a width of 12.23 seconds and a height of 80.10 seconds. Note that in this case, scrambling is effective at hiding the delay of A; the response time of the scrambled query is nearly identical to that of original query with no delay of A.

If no tuples of A have arrived at the end of the first iteration, then another iteration is performed. In this case, the second iteration retrieves, partitions, and joins relations C and D. As shown in Figure 8, this iteration requires an additional 26.38 seconds, and if A begins to arrive during this iteration, then the resulting query plan has a total response time of 80.90 seconds. Thus, in this experiment, scrambling is able to hide delays of up to 38.61 seconds with a penalty of no more than 0.80 seconds (i.e., 1%) of the response time of the original query with no delay. This corresponds to an absolute response time improvement of up to 32% compared to not scrambling.

If, at the end of the second iteration, tuples of A have still failed to arrive, then the third iteration is initiated. In this case however, there are no more runnable subtrees, so scrambling switches to its second phase, which results in the creation of new joins (see Section 2.3). In this third iteration, the result of C$\bowtie$D is partitioned and joined with relation B. This iteration has a width of only 2.01 seconds, because both inputs are already present, B is already partitioned, and the result of C$\bowtie$D is fairly small. The response time of the resulting plan is 82.22 seconds, which again represents a response time improvement of up to 32% compared to not scrambling.

Scrambled Plan #	Performed by Iteration	Total Delay	Response Time	Savings
1	Partition B	0–12.23	80.10	up to 13.18%
2	X1←C⋈D	12.23–38.61	80.90	12.31–31.81%
3	X2←B⋈X1	38.61–40.62	82.22	30.69–31.85%
4	X3←X2⋈E	40.62–50.32	82.51	31.61–36.70%
5	X4←X3⋈F	50.32–60.07	83.05	36.28–40.72%
6	X5←X4⋈G	60.07–69.79	83.52	40.38–44.21%

Table 2: Delay Ranges and Response Times of Scrambled Query Plans

The remaining query plans exhibit similar behavior. Table 2 shows the the additional operations and the overall performance for each of the possible scrambled plans. In this experiment, the largest relative benefit (approximately 44%) over not scrambling is obtained when the delay of A is 69.79 seconds, which is the time required to complete all six iterations. After this point, there is no further work for query scrambling to do, so the scrambled plan must also wait for A to arrive. As can be seen in Figure 8, at the end of iteration six the response time of the scrambled plan increases linearly with the delay of A. The distance between the delay of A and the response time of the scrambled plan is the time that is required to complete the query once A arrives.

Although it is not apparent in Figure 8, the first scrambled query is slightly slower than the unscrambled query plan when A is delayed for a very short amount of time. For a delay below 0.07 seconds, the response time of the scrambled query is 80.10 seconds while it is 80.03 seconds for the non-scrambled query. When joining A and B, as the unscrambled query does, B is partitioned *during* the join, allowing one of the partitions of B to stay in memory. Partitioning B *before* joining it with A, as the first scrambled query plan does, forces this partition to be written back to disk and to be read later during the join with A. When A is delayed by less than the time needed to perform these additional I/O, it is cheaper to stay idle waiting A during such a short time.

3.4 Experiment 2: Sensitivity Analysis of the Second Phase

In the previous experiment all the joins produced the same number of tuples, and as a result, all of the operations performed in the second phase were beneficial. In this section, we examine the sensitivity of the second phase to changes in the selectivities of the alternative joins it creates. Varying selectivities changes the number of tuples produced by these joins which affects the width and the height of each step.

For our test query, the first join created in Phase 2 is the join of relation B with the result of C⋈D (which was materialized during the first phase). In this set of experiments, we vary the selectivity of this new join to create a result of a variable size. The selectivity of this join is adjusted such that it produces from 1,000 tuples up to several thousand tuples. The other joins that phase two may create behave like functional joins and they simply carry all the tuples created by (B⋈(C⋈D)) through the query tree. At the time these tuples are joined with A, the number of tuples carried along the query tree returns to normality and drops down to 1,000. Varying the selectivity of the first join produced by the second phase is sufficient to generate a variable number of tuples that are carried all along the tree by the other joins that Phase 2 may create. We believe this is representative of the potential overhead of created joins.

The two next sections present the results of this sensitivity analysis for a small and then a large memory case. When the memory is small, relations have to be partitioned before joined as mentioned in the previous experiment. This partitioning adds to the potential cost of scrambled plans because it results in additional I/O that would not have been present in the unscrambled plan. When the memory is large, however, hash-tables can be built entirely in memory so relations do not need to be partitioned. Thus with large memory, the potential overhead of scrambled plans is lessened.

3.4.1 Small Memory Case

Figure 9 shows the response time for several scrambled query plans that differ by the number of tuples created during the second phase as the delay for relation A increases. In this experiment, we study 3 different selectivities for the first join created by the second phase of query scrambling.

This delay is, as in the previous experiment, shown along the X-axis, and is also represented as the lower grey line in the figure. The higher grey line shows the response time of the unscrambled query, which as before, increases linearly with the delay of A. The lines for the delay of A and for the response time of the unscrambled query are exactly the same than the ones presented in the previous experiments. As such, the distance between these two lines is constant and equal to 80.03 seconds and this time, as before, includes the partitioning of the base relations.

All the lines in the middle show the response time for the scrambled query plans that are executed for various delays of A and for various selectivities. Note all these scrambled query plans share the exact same response times for the first two iterations performed during the first phase of query scrambling. These two first iterations corresponds exactly to the scrambled plans 1 and 2 described in the previous experiment. At the end of the second iteration (38.61 seconds), however, if the tuples of A have still failed to arrive, a third iteration is initiated and the query scrambling enters its second phase which creates new joins.

The first selectivity for this join is such that it produces a result of 1,000 tuples. The corresponding line is the lowest dotted line in the figure. This line is identical to the one showed in the previous experiment since all the joins were producing 1,000 tuples.

With the second selectivity, the first join created by the second phase produces 10,000 tuples. If at the end of this iteration, the tuples of A have still not arrived, another iteration is initiated

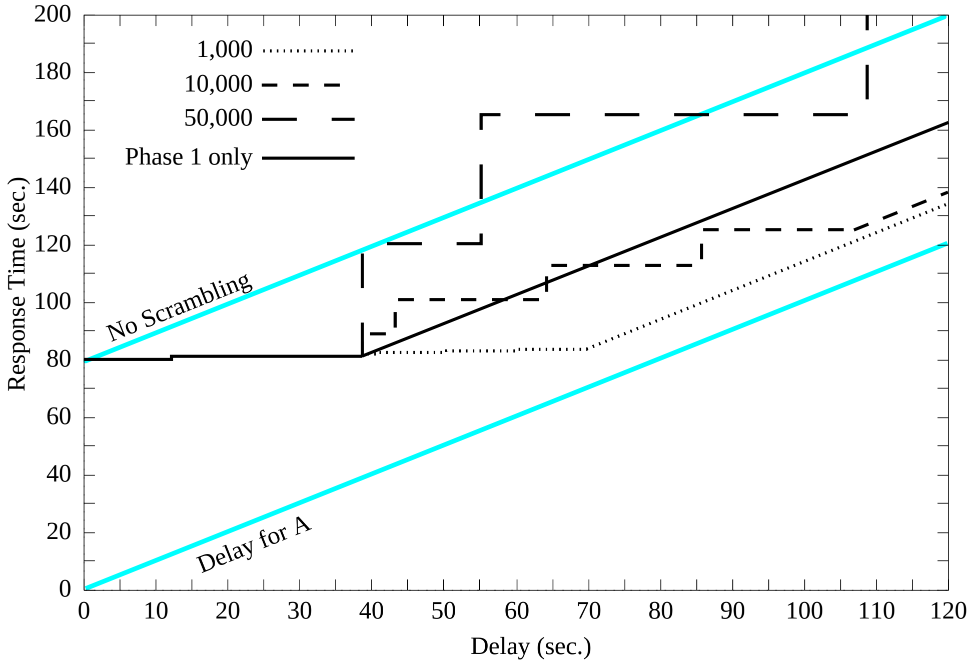

Figure 9: Response Times of Scrambled and Non-Scrambled Queries

(Small Memory, Varying Selectivity and Delay.)

and this iteration has to process and to produce 10,000 tuples. The corresponding line in the figure is the lowest dashed line. In this case, where 10 times more tuples have to be carried along the scrambled query plans, each step is higher (roughly 12 seconds) and wider since more tuples have to be manipulated than in the case where only 1,000 tuples are created. Even with the additional overhead of these 10,000 tuples, however, the response times of the scrambled query plans are far below the response times of the unscrambled query with equivalent delay.

With the third selectivity, this join creates a result of 50,000 tuples. The response time of the corresponding scrambled query plans are represented by the higher dashed line made of huge steps (about 40 additional seconds per step). Note that only the response time of the two first iterations performed during the second phase are represented. The overhead of manipulating this large number of tuples (50 times the number of tuples planed in the original query tree) makes the response time of the scrambled plan almost equal or even worse that the one of the original unscrambled query *including the delay for A*. It is more costly to carry this large number of tuples and to join them with A to execute the original but delayed query.

This experiment shows that, depending on the number of tuples and the delay, the response time of second phase scrambled query plans may stay below the response time of the unscrambled query (e.g., for 10,000 tuples), may alternate between below and above (e.g., for 50,000 tuples) or may be completely above (for huge Cartesian products for example).

The previous observation suggests that it may not be worthy to perform some joins during this second phase if they appear to be costly. It may be better, in certain cases, to stop the scrambling right after the end of the first phase, to never enter the second phase and just to wait until the tuples of A, in this case, arrive.

The performance of a scrambled query plan that follows this behavior is depicted using a solid line on the Figure. This line goes diagonal right after the end of the first phase. Intuitively, it is not interesting to perform a second phase for scrambled queries whose total response time is located above this line and below the line for the original query. Such costly joins consume a lot of resources for a little improvement. On the other hand, scrambled queries whose response time is far below this line effectively improve the response time since their overhead is small and the gain is large.

3.4.2 Large Memory Case

Figure 10 shows the response time for several scrambled query varying the selectivity of the joins of the second phase in the case where the memory is large enough to allow inner relations for joins to be built entirely in main memory. With large memory, no partitioning of relations needs to be done.

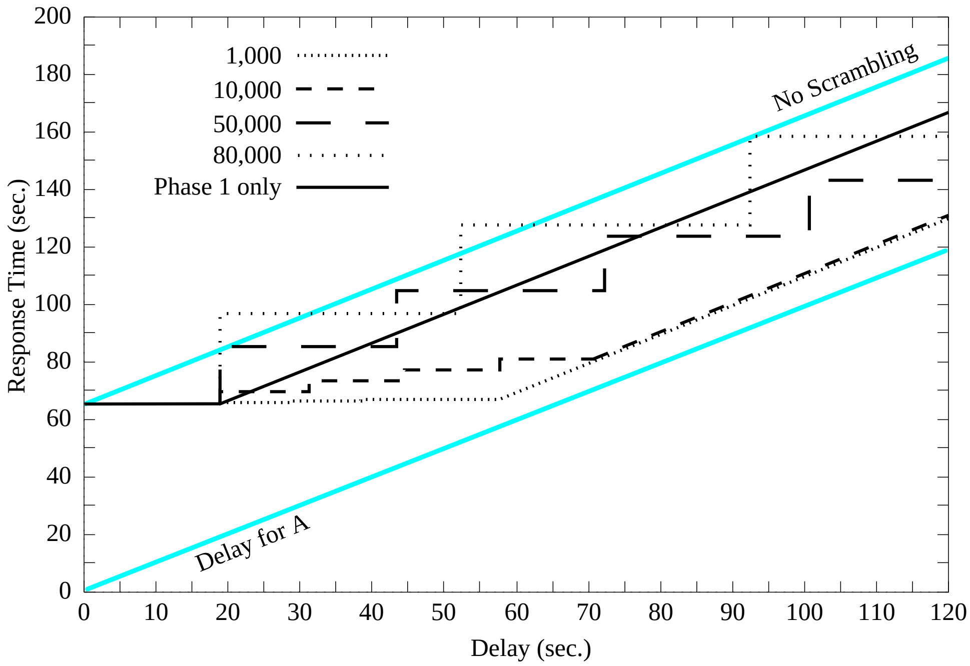

Figure 10: Response Times of Scrambled and Non-Scrambled Queries

(Large Memory, Varying Selectivity and Delay.)

For the large memory case, the lines showing the increasing delay of A and the response time of the unscrambled query when this delay increases are separated by 65.03 seconds and the second phase starts if A is delayed by more than 18.95 seconds. 4 different selectivities are represented on this Figure.

In contrast to the previous experiment where 50 times more tuples were negating the benefits of scrambling, up to 80 times more tuples can be carried by the scrambled query plans before the benefits become close to zero. With a large memory, results computed by each iteration need only be materialized and can be consumed as is. In contrast, when the memory is small, materialized results have to be partitioned before being consumed. With respect to a small memory case, not partitioning the relation when the memory is large reduces the number of I/O and allows the scrambled plans to manipulate more tuples for the same overhead.

3.5 Discussion

The experiments presented in this section have shown that query scrambling can be an effective technique that is able to improve the response time of queries when data are delayed. These improvements come from the fact that each iteration of a scrambled query plan can hide the delay of data. However, depending on the overhead due to materializations and created joins, the improvement may be small or big.

Scrambled plans constructed during the first phase are always profitable. These scrambled plans follow the directives of the query optimizer since each iteration of the first phase only materializes one existing subtree that has been optimized prior to run-time. The size of the result of each materialized subtree is thus minimal, limiting the I/O overhead due to its materialization.

Phase 2 is more risky. The algorithm must be careful in choosing the two relations to join in order to avoid huge temporary results. During each join created by Phase 2, the system may try to estimate the size of the final result during its computation. Sampling each created join and aborting it in case it appears to be too costly can be helpful not to negate or reverse the benefits of scrambling.

With respect to the Figures 9 and 10 presented above, when many iterations can be done during the first phase, the point where the second phase starts shifts to the right. This increases the distance between the Phase 1 only diagonal line and the response time of the unscrambled query. In turn, our algorithm can handle a wider range of bad selectivities for the joins it creates during Phase 2.

The improvement that scrambling can bring also depends on the amount of work done in the original query. The bigger (i.e., the longer and the costly it is) the original query is, the more improvement our technique can bring since it will be able to hide larger delays by computing costly operations. The bigger the query is, the greater the distance between the delay and the response time of the query is, and the more iterations can fit in between without crossing the upper response time line.

4 Related Work

In this section we consider related work with respect to (a) the point in time that optimization decisions are made, i.e., compile time, query start-up time, or query run-time; (b) the variables used for dynamic decisions, i.e. if the response time of a remote source is considered; (c) the nature of the dynamic optimization, i.e. if the entire query can be rewritten; and (d) the basis of the optimization, i.e. cost-based or heuristic based.

The Volcano optimizer [CG94, Gra93] does dynamic optimization 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. At query start up time, all incomparable the cost comparisons are re-evaluated. According to the result of the reevaluation, the choose-plan operator selects a particular query execution plan. All decisions regarding query execution are made at query start-up time. In contrast, we delay decisions until the middle of query execution. We believe that our work is complimentary to the Volcano optimizer. That optimizer handles optimization until query start-up but can not adapt to changes once the evaluation of the query effectively started.

Other work in dynamic query optimization either does not consider the distributed case [DMP93, OHMS92] or only optimizes access path selection and cannot reorder joins [HS93]. Thus, direct considerations of problems with response times from remote sources are not accounted for. However, these articles are a rich source of optimizations which can be carried over into our work.

The reference [Ant93] is interesting because multiple different executions of the same logical operator occur at the same time. They compete for producing the best execution – when one execution of an operator is (probably) better, the other execution is terminated. This technique has a similar flavor to our work. However, we do not run competing executions of operations, but simply reschedule operations which are delayed.

In reference [DSD95], the response time of queries is improved by reordering left-deep join trees into bushy join trees. Several reordering algorithms are presented. This work assumes that reordering is done entirely at compile time. This work cannot easily be extended to handle run-time reordering, since the reorderings are restricted to occur at certain locations in the join tree.

The reference [ACPS96] tracks the costs of previous calls to remote sources (in addition to caching the results) and can use this tracking 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.

The research prototype Mermaid [CBTY89] and its commercial successor InterViso [THMB95] are heterogeneous distributed databases which perform dynamic query optimization. Mermaid constructs its query plan entirely at run-time, thus each step in query optimization is based on dynamic information such as intermediate join result sizes and network performance. Mermaid neither takes advantage of a statically generated plan nor does it dynamically account for a source which does not respond at run-time.

The Sage system [Kno95] is an AI planning system for query optimization for heterogeneous distributed sources. This system interleaves execution and optimization and responds to unavailable data sources. We believe that our algorithm can be readily included in this system.

5 Conclusion and Plan for Future Work

Query plan scrambling is a novel technique that refers to the design of flexible distributed query processing strategies that can dynamically adjust to changes in the run-time environment. We presented an algorithm which specifically deals with variability in performance of remote data sources and accounts for *initial* delays in their response times. The algorithm consists of two phases. The first phase changes the scheduling of existing operators produced as a result of query optimization. This first phase is iteratively applied until no more changes in the scheduling are possible. At this point, the algorithm enters its second phase which creates new operators to further process available data. New operators are iteratively created until there is no further work for query plan scrambling to do.

The performance experiments demonstrated how the technique hides delays in receiving the initial requested tuples from remote data sources. They showed that the overhead of each first phase iteration is reasonable since the algorithm strictly follows the directives of the query optimizer. They also showed that the overhead of each second phase iteration is potentially higher and examined the sensitivity the performance of scrambled plans to the selectivity of the joins created in the second phase.

This work represents an initial exploration into the development of flexible systems that dynamically adapt to the changing properties of the environment. Among our future research plans, we are developing algorithms that can scramble under different failure models to handle environments where data arrives at a bursty rate or at a steady rate that is significantly slower than expected. We are also studying the use of partial results which approximate the final results. We also plan to study the potential improvements of basing scrambling decisions on cost-based knowledge. Studying interaction between the run-time engine and the optimizer may suggest some architectural changes that we definitely will explore.

Finally, Query Plan Scrambling is a promising approach to addressing many of the concerns addressed by dynamic query optimization. Adapting the query plan at run-time to account for the actual costs of operations could compensate the often inaccurate and unreliable estimates used by the query optimizer. Moreover, it could account for remote sources that do not export any cost information, which is especially important when these remote sources run complex subqueries. Thus, we plan to investigate the use of scrambling as a complimentary approach to dynamic query optimization.

Acknowledgements We would like to thank Björn T. Jónsson, Jean-Robert Gruser and Alon Levy for their helpful comments on this work.

References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. *ACM SIGMOD Conf.*, Montréal, Canada, 1996.

[Ant93] G. Antoshenkov. Dynamic Query Optimization in Rdb/VMS. *ICDE Conf.*, Vienna, Austria, 1993.

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

[Bro92] K. Brown. PRPL: A Database Workload Specification Language. Master's thesis, Univ. of Winsconsin, Madison, WI, 1992.

[CBTY89] A. Chen, D. Brill, M. Templeton, and C. Yu. Distributed Query Processing in a Multiple Database System. *IEEE Journal on Selected Areas in Communications*, 7(3), 1989.

[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. *ACM SIGMOD Conf.*, Minneapolis, MN, May 1994.

[CG94] R. Cole and G. Graefe. Optimization of Dynamic Query Execution Plans. *ACM SIGMOD Conf.*, pages 150–160, Minneapolis, MN, May 1994.

[DFJ+96] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. *22nd VLDB Conf.*, Bombay, India, September 1996.

[DMP93] M. Derr, S. Morishita, and G. Phipps. Design and Implementation of the Glue-Nail Database System. *ACM SIGMOD Conf.*, Washington, DC, May 1993.

[DSD95] W. Du, M. Shan, and U. Dayal. Reducing Multidatabase Query Response Time by Tree Balancing. *ACM SIGMOD Conf.*, pages 293–303, San Jose, CA, May 1995.

[FJK96] M. Franklin, B. Jónsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. *ACM SIGMOD Conf.*, Montréal, Canada, June 1996.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

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

[Kno95] Craig A. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence*, Montréal, Canada, 1995.

[OHMS92] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query Processing in the ObjectStore Database System. *ACM SIGMOD Conf.*, San Diego, CA, June 1992.

[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.

[THMB95] M. Templeton, H. Henley, E. Maros, and D. Van Buer. InterViso: Dealing with the Complexity of Federated Database Access. *VLDB Journal*, 4:287–317, 1995.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. *ICDCS Conf.*, Hong Kong, 1996.