Cost-based Query Scrambling for Initial Delays*†

Tolga Urhan*
University of Maryland
urhan@cs.umd.edu

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

Laurent Amsaleg‡
IRISA
Laurent.Amsaleg@irisa.fr

Abstract

Remote data access from disparate sources across a wide-area network such as the Internet is problematic due to the unpredictable nature of the communications medium and the lack of knowledge about the load and potential delays at remote sites. Traditional, static, query processing approaches break down in this environment because they are unable to adapt in response to unexpected delays. Query scrambling has been proposed to address this problem. Scrambling modifies query execution plans on-the-fly when delays are encountered during runtime. In its original formulation, scrambling was based on simple heuristics, which although providing good performance in many cases, were also shown to be susceptible to problems resulting from bad scrambling decisions. In this paper we address these shortcomings by investigating ways to exploit query optimization technology to aid in making intelligent scrambling choices. We propose three different approaches to using query optimization for scrambling. These approaches vary, for example, in whether they optimize for total work or response-time, and whether they construct partial or complete alternative plans. Using a two-phase randomized query optimizer, a distributed query processing simulator, and a workload derived from queries of the TPC-D benchmark, we evaluate these different approaches and compare their ability to cope with initial delays in accessing remote sources. The results show that cost-based scrambling can effectively hide initial delays, but that in the absence of good predictions of expected delay durations, there are fundamental tradeoffs between risk aversion and effectiveness.

*Appears in ACM SIGMOD International Conference on the Management of Data, Seattle, WA, June 1998.

†This work was partially supported by the NSF under grant IRI-94-09575, by the Office of Naval Research under contract number N66001-97-C8539 (DARPA order number F475), by Bellcore, and by an IBM Shared University Research award.

‡Laurent Amsaleg's work was partially supported by an INRIA fellowship. He performed this work while he was with the University of Maryland.

1 Introduction

The ubiquity of wide-area connectivity has led to tremendous increases in the number, variety, and distribution of data sources that can be accessed from one's desktop. At present, most such access is performed in a browsing mode, either by navigating through hyperlinks or by performing word-based searches via search engines. Distributed query processing, as developed for relational and object relational database systems, currently plays little role in wide-area data access. As a result, the benefits of declarative query processing, such as the expressive power of query languages and automated optimization of query plans, are largely unavailable when accessing wide-spread data sources across the internet.

The absence of declarative query processing places unnecessary restrictions on the types of applications that can exploit the increased interconnectivity of data sources. The severity of such a limitation has been demonstrated before, most recently in the battle between object-oriented database (OODB) and object-relational database (ORDB) systems. To date, navigation-oriented OODB approaches have remained largely niche solutions, while the query-oriented ORDB approach has been embraced by most of the major database vendors [CD96]. Given the importance of declarative query processing for many applications, it is natural to investigate ways to provide such functionality over the wealth of data that is available across current wide-area networks.

Query processing in wide-area distributed environments poses a number of difficult technical challenges. Issues such as semantic heterogeneity, manipulation of semi-structured data, and resource discovery (i.e., locating relevant sources) have been the subject of much research in recent years [Kim95, SAD$^+$95, TRV96]. While these problems are daunting in their most general forms, pragmatic approaches that provide useful functionality for many typical situations are starting to appear. In particular, solutions based on the wrapper-mediator model and other non-traditional techniques (eg. [MMM97]), provide the abstractions necessary to implement applications that utilize multiple sources on the Internet.

While significant effort has been placed on addressing the semantic issues of wide-area data access, relatively little effort has been put into solving the performance problems that are inherent in such access. A key performance issue that arises in wide-area distributed information systems is *response-time unpredictability*. Data access over wide-area networks involves a large number of remote data source, intermediate sites, and communications links, all of which are vulnerable to congestion and failures. Such problems can cause significant and unpredictable *delays* in the access of information from remote sources.

Traditional distributed query processing strategies break down in the wide-area environment because they are unable to adapt in response to unexpected delays. 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. Due to the apparent randomness of delays when accessing remote data, it is not possible to optimize for such delays *a priori*. Thus, the execution of any statically optimized query plan is likely to be sub-optimal in the presence of the response time problems that will inevitably arise during the query run-time.

1.1 Query Scrambling

To address the issue of unpredictable delays in the wide-area environment, we proposed a dynamic approach to query execution called *query scrambling* [AFTU96]. Query scrambling reacts to unexpected delays by rescheduling, *on-the-fly*, the operations of a query during its execution. In a remote access setting, query scrambling hides delays encountered when obtaining data from the 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 as defined in [AFTU96] consists of two different phases: a *rescheduling* phase, in which the scheduling of the operators of an active query plan is changed when a delay is detected, and an *operator synthesis* phase in which the query plan is *restructured*, typically by creating new operators that are not in the current query plan. In the original algorithm, both of these phases were *heuristic-driven*. That is, the algorithm was specified as a set of heuristic rules that were activated as delays in obtaining remote data were detected. The heuristics described in that paper were shown to be very effective at hiding delays in some situations, but they were also shown to be prone to making poor scrambling decisions in other cases. In some cases, the proposed heuristics could result in performance that is worse than simply waiting for the delayed data to arrive.

In this paper, we address the shortcomings of the heuristic-based approach by investigating ways to introduce query optimization into the scrambling decision making process. For simplicity, we focus on the problem of *initial delay*, in which delays are manifested as problems in receiving the first tuple from a particular remote source. Initial delays typically arise when there is difficulty in establishing a connection to a remote source or the source is heavily loaded. Also, in the absence of global query optimization (i.e., optimization that considers costs at both the query site and the remote sources such as [RAH+96, TRV96]), initial delays can arise if a remote source must perform a significant amount of work before it can return the first tuple.

1.2 Making Cost-based Decisions

There are a number of fundamental issues that arise when trying to exploit database query optimization technology for scrambling. A basic question is whether the objective function of optimization should be based on *total work or response time*.

Relational optimizers traditionally aim to reduce total work (or "cost"). For example, the cost model of the classic System R-type optimizer includes terms for cpu and disk usage, but does not model the possible overlap of cpu and disk processing [SAC+79]. Likewise, the distributed extensions to this optimizer for the R* system added additional terms for message costs, but also did not model the overlap of such costs [ML86]. In contrast, a response time-based optimizer predicts the overlap of work in addition to the total amount of work [GHK92]. Thus, a response time optimizer might choose a plan with higher total work but more parallelism over a plan with less work but higher sequentiality.

The notion of *delay*, as arises in wide-area remote access is inherently a response time issue. Delays incur no work but still postpone the completion of a given query. We therefore investigate the use of response time-based optimization for query scrambling. In fact, a major result of our work is that if a response time-based optimizer is given an estimate of an expected delay, it can place the access to the delayed data at the proper point in the query execution plan. The quality of such placement of course, depends upon the accuracy of the delay prediction. The current state-of-the-art in delay prediction on the Internet is quite primitive. We therefore investigate two approaches for integrating a response time-based optimizer into the scrambling process. One approach is very aggressive in its scrambling (i.e., it assumes that delays will be long), the other approach is more conservative. In addition, we also develop an algorithm for performing scrambling using an optimizer that is based on total work.

The rest of the paper is organized as follows. Section 2 gives an overview of the cost based query scrambling and describes three approaches for integrating query optimization with scrambling decisions. Section 3 describes the environment used in the experiments. Section 4 analyzes the cost based scrambling algorithms using a two-phase randomized query optimizer, a distributed query processing simulator, and a workload derived from queries of the TPC-D benchmark. Related work is discussed in Section 5. Finally Section 6 presents our conclusions.

2 Cost-based Query Scrambling

2.1 Query Scrambling Overview

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

An example query execution plan for such an environment is shown in Figure 1. The query involves five different base relations stored at four different sites. In the example, relations A and B reside at separate remote sites, relation C resides locally at the query site, and relations D and E are co-located at a fourth site.

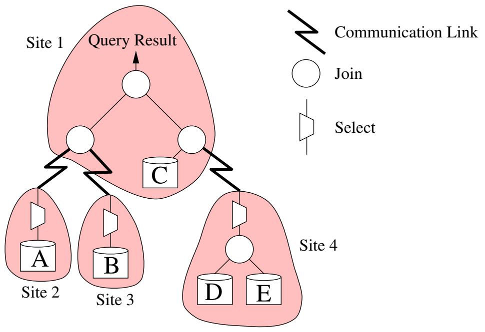

Figure 1: Example of a Complex Query Tree

Using a static scheduling policy, a remote access query plan such as this is susceptible to delays that arise when accessing the remotely-stored data. For example, using an iterator approach [Gra93], the first data access would be to request a tuple of Relation A (from site 2). If there is a delay in accessing that site then the scan of A, and hence the entire query execution, is blocked until the site recovers.

Query Scrambling reacts to such delays in two ways (referred to as Phase 1 and Phase 2 respectively):

- *Rescheduling* - the execution plan of a query can be dynamically rescheduled when a delay is detected. In this case, the basic shape of the query plan remains unchanged (although some additional "materialization" operators may be added as discussed in Section 2.2).
- *Operator Synthesis* - new operators (e.g., a join between two relations that were not directly joined in the original plan) can be created when there are no other operators that can execute. In this case, the shape of the query plan can be significantly modified through the addition, removal and/or re-arrangement of query operators.

Query scrambling is an iterative process; it works by repeatedly (if necessary) applying these two techniques to a query plan. For example, assume that the query shown in Figure 1 stalls while retrieving tuples of A. Instead of waiting for the remote site to recover, Query Scrambling could perform rescheduling, and retrieve the tuples of B while A is unavailable. These tuples would need to be stored temporarily at the query site. If, after obtaining B, A is still unavailable, then rescheduling could be invoked again, for example, to trigger the execution of (D$\bowtie$E) at site 4, and to join this result with C. If at this point, A is still unavailable, then Operator Synthesis can be invoked to create a new join between B and (D$\bowtie$E)$\bowtie$C. Operators initiated by Query Scrambling may as well experience delays, which may cause Scrambling to be invoked further.

In this paper, we assume that once a scrambling step (i.e., the rescheduling of a query sub-tree, or the execution of a synthesized plan) has been started, the system does not check for the availability of delayed data unless the delayed data is accessed during the step. Once the step has been completed, the arrival of the delayed data is checked. If the delayed data has still not arrived, another iteration of the scrambling algorithm is begun. Likewise, if during scrambling, a delay arises when accessing a remote source, the current scrambling step is abandoned, and a new one is started.

As discussed in Section 1.1, the original formulation of scrambling was heuristic-based [AFTU96]. In this paper, we address the shortcomings of that earlier approach by incorporating query optimization into the scrambling process. The focus of the paper is on Phase 2 of scrambling, but we also apply cost-based decision making in Phase 1. We therefore first briefly describe how cost information is used during Phase 1, and then describe our three approaches for integrating cost information into Phase 2. The comparison of these latter three approaches is then addressed in detail in the remainder of the paper.

2.2 Cost-Based Rescheduling (Phase 1)

Phase 1 starts by identifying *runnable subtrees*, i.e., sub-trees of the plan that are made up entirely of operators that are not currently blocked. A runnable subtree can be scheduled out of order by inserting a "materialization" operator between its

[1] As currently specified, query scrambling treats remote sources as black boxes, regardless of how the remote data is computed. Thus, it operates solely at the query sites.

root and the root's parent. These materialization operators issue Open, Next, and Close calls to the root of the subtree and save the result in a temporary relation to be used when the result is needed later.

The original scrambling algorithm rescheduled subtrees simply by traversing the query plan from left to right, and choosing to run each "maximal" runnable subtree (i.e., a runnable subtree whose parent operator is blocked) it encountered. This approach was taken because the original algorithm did not use query optimization for scrambling.

In this paper, we use a query optimizer to compute the expected cost (in terms of total work) of each runnable subtree. Three costs are associated with each runnable subtree: 1) MW, the cost of writing the materialized temporary result produced by the subtree to disk; 2) MR, the cost of reading the temporary from disk when it is to be used; and 3) P, the cost of executing the subtree itself. Note that MW and MR represent the *additional* cost incurred by running the subtree out of order.[2] Also note that MW and MR can differ depending on the relative costs of disk writes and reads in a system.

The *efficiency* of each runnable subtree is then computed as $\frac{P-MR}{P+MW}$. Intuitively $P - MR$ is how much work will be saved in the future by scheduling this subtree, and $P + MW$ is the duration of the scrambled operation. Thus, the ratio gives the improvement in response time per unit of scrambled execution. Phase 1 then chooses to re-schedule runnable subtrees in decreasing order of efficiency. Subtrees with efficiency below a certain threshold (currently set at 0.75) are not considered for execution during Phase 1. This approach to using cost favors runnable subtrees that are materialized by the original plan (they have efficiency = 1) and ones that produce small results.

Each iteration of Phase 1 chooses a runnable subtree, runs it, and then checks to see if the delayed data has started to arrive. If so, then scrambling is terminated at this point. If the delayed data is still missing, then another iteration of scrambling is initiated.

2.3 Cost-Based Operator Synthesis (Phase 2)

Phase 2 of query scrambling is invoked after no more progress can be made in Phase 1. Unlike Phase 1, which simply changes the scheduling of existing operators, Phase 2 actually creates a new plan, which typically contains new operators. In this paper, we study three approaches to using a query optimizer during Phase 2.

2.3.1 Optimization Strategies

As stated in Section 1.2, an optimizer can be integrated with scrambling using an objective function based on total work or response time. In scrambling, we deal with these two types of optimization differently.

Response time-based optimizers are naturally suited for query scrambling because they are able to estimate not only the total work to be done for a query, but also how that work can be overlapped. The ability to consider such overlap can be exploited for query scrambling. Simply by telling the optimizer how long a particular data source will be delayed, the optimizer can be coerced into finding plans that perform other useful work that is overlapped with the delay. Of course, the work that the optimizer schedules to overlap the delay can not be in any way dependent on the delayed data. Fortunately, query optimizers must normally deal with such dependencies, in order to generate valid query plans.

There is a problem with the above approach, however. It requires that the duration of the delay of a source to be known *a priori*. Of course, if such knowledge exists, then there is no need for scrambling. When dealing with delays of remote sources on the Internet the current state of prediction is quite primitive (e.g., note the "time remaining" line on the bottom of your browser window). The approach that we take in this paper is to "lie" to the optimizer by providing it a fixed estimate of the expected delay duration when scrambling is invoked. We propose two such approaches. One approach, called *Include Delay*(IN), simply chooses a very long (relative to the query response time) delay duration so that the optimizer will push any accesses to the delayed data as far back in the plan as possible. The other approach, called *Estimated Delay* (ED) initially assumes that delays will be brief, and subsequently increases its estimate if a delay turns out to be longer than the earlier guess.

An alternative to using a response time-based optimizer is to use a more traditional objective function based on total work (i.e., one that ignores potential overlapping). Such an optimizer, however, can not adequately cope with the notion of delay since delay information is not taken into account in the objective function. Our solution to this problem is to simply remove the accesses to the delayed data from the query. In this paper we explore an approach called Pair, which has the optimizer generate plans for individual joins one at-a-time. This policy is a cost-based analogue to the heuristic-based Phase 2 of the original scrambling algorithm. In the following sections we briefly describe these three approaches: Pair, IN, and ED.

2.3.2 Pair

The Pair approach uses a total work-based optimizer to construct a query plan containing only a single join using two relations that are not currently known to be blocked. The optimizer analyzes each pair of non-blocked relations that share a join predicate (i.e., it avoids Cartesian products) and calculates the cost of the best way of joining them. It then chooses the join with the least total cost and executes it. The cost

[2] Thus, MR and MW are zero for subtrees rooted at operators that write their results to disk under the regular query schedule.

computation includes costs associated with materializing the result. The result of this join is materialized to disk, and is available for use in later scrambling iterations. In addition to avoiding Cartesian products, the pair policy also avoids joins that produce a result that takes longer to read from disk than it would take to compute from scratch later. At the end of each join, the policy checks for the arrival of delayed data. If the data has not yet arrived, another iteration is begun. If Pair runs out of qualified joins before the delay terminates, scrambling simply halts and waits for the delayed data to arrive. When all the blocked relations become available the Pair policy constructs a single query tree that computes the result before the normal execution resumes. This is necessary because during scrambling the Pair policy does not try to maintain a complete query plan that represents the final result, but rather works on pairs of relations. This phase, which is called *reconstruction phase*, utilizes the optimizer to find the optimal query plan.

2.3.3 Include Delayed (IN)

In contrast to the Pair approach, IN uses a response time-based optimizer, and thus, each iteration of scrambling generates a *complete* alternative plan. For all delayed data sources, the optimizer is told that the delay duration will be very large (i.e., many times longer than the expected response time of the non-delayed plan). This approach results in delayed accesses being pushed far back in the plan schedule. It is interesting to note however, that the optimizer does not always push such accesses to the very end of the schedule. In some cases, doing so would incur excessive work after the delayed data has arrived, which would result in even worse performance. The optimizer is naturally able to recognize such situations and places delayed accesses in the "right place" in the plan.

One issue that arises when using a response time-based optimizer in this manner is that the optimizer is geared towards choosing the plan that ultimately results in the best response time for the delay value that we give it. Since we really do not know what the delay will be however, this single-mindedness can sometimes be harmful. In general, when the scrambler decides to initiate work in order to hide delay, it commits to performing an entire step. We refer to the duration of the step as *risk* and to the potential improvement in response time as *benefit*. The optimizer chooses the plan with the greatest benefit whose risk can be overlapped with the expected delay duration. This can cause problems when the delay turns out to be relatively short.

In order to address this problem, we introduce a parameter called the "risk/benefit knob" (*RBknob*), that prevents the optimizer from choosing very high-risk plans for relatively small potential gains over lower-risk plans. The *RBknob* is expressed as the ratio of the amount of benefit the optimizer is willing to give up for a given savings in risk. Increasing the *RBknob* has the effect of making the policy more conservative. The performance of this knob is studied in Section 4.3.

2.3.4 Estimated Delay (ED)

The Estimated Delay (ED) approach works similarly to IN except that rather than starting by assuming a huge delay, it first tries relatively short delays, and successively increases its delay estimates if necessary. The motivation behind this approach is that assuming a large delay initially may cause the optimizer to pick a risky plan that has high payoff for long delays, but hurts performance for short delays. Likewise, if the delay estimate is too small, scrambling may be rendered ineffective for larger delays because the optimizer will refuse to run high risk/high pay off plans.

ED works as follows: It starts by picking an estimated delay value equal to the 25 % of the original query response time. Until the delayed data arrives, iterations are repeated with this estimated delay value as long as some progress is being made by each iteration. When this value becomes too small to allow any progress, it is increased to 50 % of the original query response time. Finally when this becomes insufficient, we use a value of 100 % of the original response time. This scheme allows scrambling to first perform iterations with low risk, but still make progress. Thus, in the event that a delay turns out to be short, scrambling has helped rather than hurt. In the event of longer delays, ED becomes more aggressive, which allows it to attempt higher-risk plans. Note that the *RBknob* described for the IN policy is also used for ED, but in general it has less impact here, because ED is already a more conservative policy than IN.

3 Experimental Environment

Our experiments are performed using a detailed simulator of a distributed query processing environment, a two-phase randomized query optimizer, and a workload based on queries from the TPC-D benchmark. We describe each of these in the following sections.

3.1 Simulation Environment

To study the performance of the cost-based scrambling approaches we implemented them on top of a simulator that models a distributed, peer-to-peer database environment and that is capable of realizing iterator-based or process-based scheduling of query operators. In this study, we used only a single join method, namely, hybrid hash [Sha86].

Table 1 shows the main parameters for configuring the simulator and the main settings used for this study. There are two types of sites. Data sources, which store base data that will be used in queries, and Query sites, which execute queries. Every site has a CPU whose speed is specified by the *Mips* parameter, *NumDisks* disks, and a main-memory buffer pool of size *SourceMem* or *QSiteMem*. For the current study, the simulator was configured to have a single query site and six remote data source sites. In all the experiments described in this paper, we placed no additional load on the source sites beyond what was generated by the query requests.

Parameter	Value	Description
NumSources	6	number of data source sites
Mips	200	CPU speed (10^6 instr/sec)
NumDisks	1	number of disks per site
DskPageSize	4096	size of a disk page (bytes)
WriteBufSize	4	size of disk write buffer (pages)
RequestSize	40	size of a data request (bytes)
NetPageSize	4096	size of a data transfer (bytes)
Compare	4	instr. to apply a predicate
HashInst	25	instr. to hash a tuple
Move	2	instr. to copy 4 bytes
QSiteMem	300, 1,000 or 10,000	Query site memory size (pages)
SourceMem	10,000	Data source memory size (pages)
NetBw	12	network bandwidth (Mbits/sec)
MsgInst	20000	instructions to send or receive a message
PerSizeMI	3	instructions per byte sent
DiskInst	5000	instructions to read a page from disk

Table 1: Simulation Parameters and Main Settings

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

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

In this study, the disk at the query site is used mostly to temporarily store intermediate query results and data obtained from remote sources during a query execution. In addition, some small base relations can also be permanently stored at the query site. The other base relations are stored on disk at the sources as described in Section 3.3. Although sources are configured with memory, the workload used in the experiments here is performed such that the server memory is not useful (i.e., there is no caching across queries and relations at the remote sites are accessed once per query). Thus, in the experiments that follow, base relations are always read (sequentially) from the sources' disks for each query execution. In addition, any selections or projections on base data that are required by a query are performed at the *sources* before the data is shipped to the query site.

At the query site, when scrambling is being used, the buffer manager uses a special policy that pins any memory-resident data that will be used in the next scrambling iteration. Any other cached data is unpinned, and is managed using an LRU policy. In the case that the amount of pinned data prevents the query from allocating the additional buffer space it needs, some of the pinned data is released. In particular, memory-resident relations, which do not have an image on the local disk and are accessed early in the plan are given preference to remain in memory over other relations. Disk writes for intermediate results are buffered in groups of *WriteBufSize* in order to increase the amount of sequential I/O. For join partitions, where memory is at a premium, writes are done one page at-a-time.

The network is modeled as point to point connections between each source and the query site. As such, link failures are independent of each other. The speed of each link (*NetBw* Mbits/sec) is set to be slightly higher than speed of sequential disk access at the data sources, in order to make sure that network speed is not the bottleneck in these experiments. The details of a particular networking technology (e.g., Ethernet, ATM) are not modeled. The cost of sending messages, however, is modeled as follows: the simulator charges for the time-on-the-wire (depending on the message size and the network bandwidth) as well as CPU instructions for networking protocol operations, which consist of a fixed cost per message (*MsgInst*) and a per-byte cost based on the size of the message (*PerSizeMI*). The CPU costs for messages are paid both at the sender and the receiver.

The query execution model uses a page-at-a-time (i.e., non-streaming) approach to remote data access. That is, when an operator running at the query site needs data from a remote source, it sends a request (of *RequestSize* bytes) to that source and waits for the reply. A source responds with a block of *TransferSize* bytes of data. The query site employs prefetching (of one page) to reduce network latency.

In the experiments, delays are modeled by simply blocking the link between a remote source and the query site. Such delays could also be modeled by suspending processing at the source. Since we use point-to-point connections between the sources and the query site, these two methods are equivalent.

3.2 The Query Optimizer

In the study, we use a two-phase randomized optimizer similar to the one described in [IK90, IW87]. The optimizer first runs *Iterative Improvement* (II) algorithm for some time, followed by *Simulated Annealing* (SA). It is possible to trade off the quality of the chosen query plan vs. the optimization time by changing two parameters, *OptTries* and *OptMoves*, which control the number of starting points in the search space and the number of iterations performed on each starting point during the II phase. These variables (shown in Table 2) are scaled with the number of relations used in the query. Thus the number of plans generated by the optimizer increases quadratically with the number of relations. The search space for the optimizer includes left deep, right deep and bushy plans.

Parameter	Value	Description
OptTries	$10 \times NR$	# of starting points in search space
OptMoves	$1 \times NR$	# of iterations performed on a starting point
NR	variable	Number of relations in the query being optimized
RBKnob	0.01	Risk/Benefit knob

Table 2: Optimizer Parameters and Main Settings

The optimizer can use any provided objective function to rate alternative query plans. We use two such functions in this study: one based on total work, and one based on response time. Our response time model is derived from the one defined in [GHK92]. It calculates expected response times by considering potential parallelism in addition to the work done by query operators.

The optimizer takes the following input: 1) information about the relations that participate in the query including cardinalities, fields, and the data source, etc.; 2) the join predicates between the relations; and 3) the selection predicates on the relations together with a selectivity factor for the predicate. Depending on the scrambling approach used, information on all or only a subset of the relations may be given to the optimizer. The Estimated Delay and Include Delayed approaches also provide information about which relations are delayed and an estimate of how long the delay is expected to last. Recall that, as described in Section 2.3, the response time-based approaches also use a special "knob" to control the risk/benefit tradeoffs made by the optimizer. The default value of this knob is 0.01, which is a very aggressive setting. The effect of more conservative settings is investigated in Section 4.3.

3.3 Workload

As stated previously, we examine the performance of the cost-based approaches using queries and a database derived from the TPC-D benchmark. The database is based on a TPC-D Scaling Factor (SF) of 1, and is described in Table 3. The table shows several different data sizes for each relation. In the experiments, we model the effect of projections on the tuples processed at the query site by reducing the size of all projected tuples sent to that site to a fixed amount (40 bytes). Selections and projections are pushed to the data sources where possible. As a result, remote sources read *TuplePages* pages from disk when scanning a relation but transmit only *ProjectedPages* pages to the query source when a projection is applied at the source (and possibly fewer if a selection predicate is also applied). We also use 40 bytes as the size of the tuples produced by a join.

Table	Tuples	Tuple Size	Pages	Projected Pages	Primary Key
Region	5	120	1	1	`regionkey`
Nation	25	120	1	1	`nationkey`
Supplier	10 K	160	400	100	`suppkey`
Customer	150 K	180	6818	1486	`custkey`
Order	1,500 K	100	37500	14852	`orderkey`
Part	200 K	160	8000	1981	`partkey`
Lineitem	6,000 K	120	181818	59406	`orderkey+ linenumber`
PartSupp	800 K	140	28571	7921	`suppkey+ partkey`

Table 3: Database schema and data sizes

Two of the relations (`REGION` and `NATION`) are very small "detail tables" that change very infrequently, so copies of these are maintained and accessed at the query site. The remaining base tables are each placed at a separate remote server.

In terms of queries, we have chosen three of the TPC-D queries (Q5, Q8, and Q9) for our experiments. These queries were chosen because they are fairly complex (6 to 8-way joins) so they provide significant opportunities for interesting scrambling behavior. Because our simulator does not model aggregate functions, GROUP BY and ORDER BY clauses, or sub-queries, we have modified the original queries slightly. The modified versions are described in Section 4. It should be noted that our goal in using TPC-D queries as a starting point is to allow us to examine the approaches using realistic join graphs, cardinalities, and selectivities; we do not claim to draw any conclusions about performance on an actual TPC-D benchmark.

3.4 Experimental Methodology

All the graphs shown in the following section plot the duration of an initial delay of a remote source vs. the response time achieved with each of the scrambling approaches. The results for all of the approaches tend to exhibit a step behavior due to the iterative nature of the scrambling process. The graphs were generated as follows.

First, for each combination of query, memory allocation, and delayed relation, we ran each scrambling approach with a very long delay to find the delay duration (i.e., the point on the x-axis) where each iteration would occur for that approach. For all the delay durations in the interval between two such points, the query response time will be the same. This run, however does not show what the value of that response time will be. We therefore pick one delay value within each interval and run the scrambling approach to obtain the response time with a delay of that duration. This response time is the response time for the entire interval.

Because we are using a randomized optimizer, we needed to be careful that both the initial plans and the scrambled plans that were generated were good plans. Otherwise, particularly bad plans could result in spurious effects that were not due to the scrambling approaches. To ensure that we had good plans we did the following: First, we generated the intervals using higher values for the II parameters to increase the thoroughness of the search. These runs were repeated three times to ensure the repeatability of the scrambling iteration intervals. Then, we also ran each data point (i.e., combination of delay interval, scrambling approach, query, memory allocation, and delayed relation) at least three times (using the normal optimizer parameter settings), and checked that the plans generated at each point conformed to the intervals found initially, and that the generated response times were accurate to within plus or minus 2%.

Num.Relations	Optim.Time
4	1.290 secs
5	2.528 secs
6	5.109 secs
7	8.308 secs
8	11.657 secs

Table 4: Optimization times for various numbers of base relations.

The results that we report in Section 4 do not include the time required for running the query optimizer. The goal was to avoid mixing numbers from the *real* query optimizer with those from a *simulated* system. As can be seen in Table 4, the optimization times obtained with our optimizer (on an IBM RS/6000 42T PowerPC) are quite small compared to the 750+ second-response times of the queries. These times were obtained using an optimizer that was not tuned to reduce optimization time. Thus, we would expect to be able to lower them even further if necessary, for example, in order to handle queries larger than 8-way joins.

4 Experiments and Results

4.1 Experiment 1 - National Market Share

We begin by studying the performance of the three cost-based approaches when delays are encountered during the execution of a modified version of TPC-D Query Q8, the National Market Share Query (referred to as MQ8). The SQL statement for MQ8 is shown in Figure 2.

MQ8 is an 8-way join query, with selections on the `REGION`, `ORDER`, and `PART` relations. Figure 3 shows the *query graph* corresponding to this query. In the query graphs, we abbreviate relation names using their first letters.[3] An edge between two relations indicates a join predicate between those relations in the query; the edge is labeled with the join attribute(s). Selection predicates are indicated by boxes containing the selection attribute(s) and the selectivity of the predicate is listed as a fraction below the selection box.

[3] In this query the detail relation `NATION`, which is kept at the query site is used twice. We refer to these uses as `N1` and `N2`.

```
SELECT  O.ORDERDATE, L.EXTENDEDPRICE, N2.NAME
FROM    PART, CUSTOMER, ORDER, LINEITEM,
        SUPPLIER, NATION N1, NATION N2, REGION
WHERE   P.PARTKEY          = L.PARTKEY
        AND L.SUPPKEY      = S.SUPPKEY
        AND O.ORDERKEY     = L.ORDERKEY
        AND C.CUSTKEY      = O.CUSTKEY
        AND C.NATIONKEY    = N1.NATIONKEY
        AND N1.REGIONKEY = R.REGIONKEY
        AND R.NAME         = 'EUROPE'
        AND S.NATIONKEY    = N2.NATIONKEY
        AND O.ORDERDATE BETWEEN '94-01-01'
                            AND '95-12-31'
        AND P.TYPE         = 'SMALL PLATED STEEL'
```

Figure 2: Modified National Market Share Query (MQ8)

We run the query using each of the three different memory allocations identified in table 1. Because we assume that selections and projections (where appropriate) are applied at the remote data sources, the amount of data that must be processed at the query site is significantly less than the sum of the raw relation sizes. In this experiment, a memory allocation of 1000 pages is more than sufficient to run MQ8 with no hash partitioning. When the smaller memory allocation (300 pages) is used, the two largest (intermediate) relations (the result of `PART`$\bowtie$`LINEITEM` and `ORDER`) must be partitioned in order to be joined.

The initial query plans generated by the optimizer for the 1000 page and 300 page allocations are shown in Figures 4(a) and 4(b), respectively (the initial plan for a memory size of 10,000 is identical to that of Figure 4(a)). All binary operations shown in the figures are *hash joins*, and the bold edges indicate joins which require the relations to be partitioned.

We now turn to the results of the experiment. Figures 5 and 6 show the results for MQ8 with 1000 page and 300 page memory allocation, respectively. In all of the graphs shown in this paper the x-axis indicates the initial delay (in seconds) of a remote relation (in this case, the `PART` relation) and the y-axis indicates the query response time (in seconds). In addition to the curves for each of the scrambling approaches, the graphs also contain two parallel diagonal lines. The lower line simply indicates the magnitude of the delay. Since a query cannot complete until all of the relevant data has been accessed, this "delay" line represents a lower-bound on response time. The higher diagonal line, labeled "No Scr" represents the response time that would be obtained if scrambling is not used.

In both cases, we delay `PART`, which is the left-most relation of the optimized query plan. `PART` is a very valuable relation in this plan for two reasons. First, since the iterator execution model activates operators in a pre-order manner, delaying the left-most relation leaves the most possible remaining work to be done in the absence of scrambling. Second, in

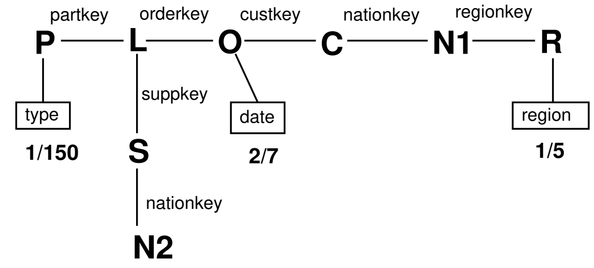

Figure 3: Query graph for query MQ8

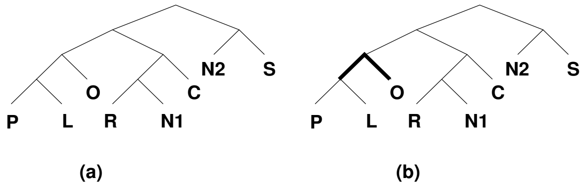

Figure 4: Query plans for (a) memory≥1000 and (b) memory=300

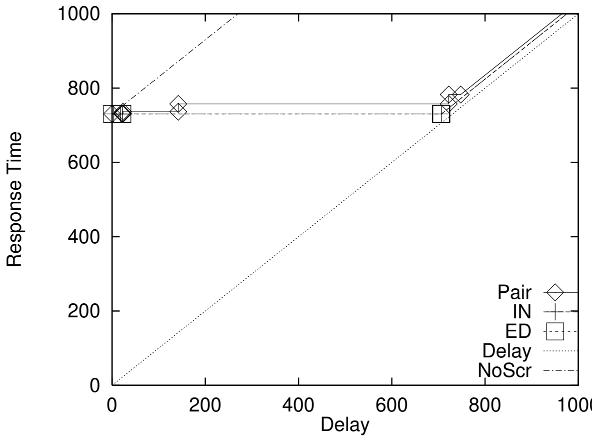

Figure 5: MQ8, PART delayed, Memory=1000

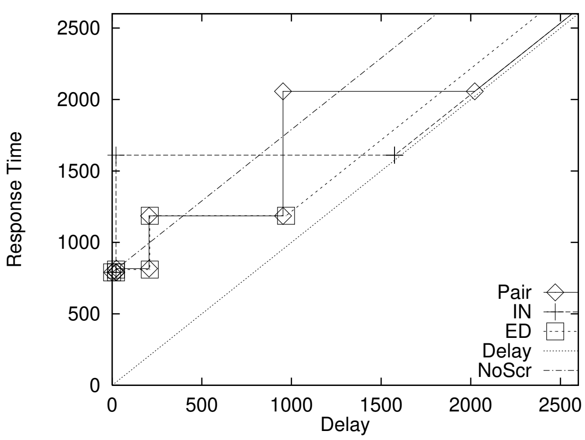

Figure 6: MQ8, PART delayed, Memory=300

query MQ8, the PART relation plays the role of a *reducer* for LINEITEM, the largest relation in the schema. That is, because of its selection predicate and the fact that it participates in a functional join with LINEITEM (and assuming uniform distribution and independence of the join attribute values), the selection on PART reduces the size of the intermediate result, PART⋈LINEITEM by the selectivity of its selection predicate. Thus, the presence of PART is important here, because it significantly reduces the number of tuples that must be processed later in the query.

4.1.1 Query MQ8 - Large Memory Allocations

Turning to the 1000 page case (Figure 5), it can be seen that all three of the cost-based scrambling approaches are very effective at hiding the delay of PART. In fact, IN and ED are able to effectively hide nearly 100% of the delay here; when the delay is 706 seconds or less (i.e., up to the knee in the curve) the response time is virtually unchanged from the non-delayed value of 730.5 seconds. In other words, the two approaches are able to effectively hide a delay that is nearly equal to the original response time of the entire query in this case. Beyond a delay of 706 seconds, scrambling has run out of additional work to perform, so the response time increases parallel to the delay. The difference between the response time lines and the delay line represents the amount of work that must be done after the delayed tuples begin to arrive. Note that the Pair approach also does well here; it performs slightly worse than IN and ED because it materializes some intermediate results to disk.[4]

In this case the first phase is run when scrambling starts, materializing the subtree that contains relation REGION, NATION1 and CUSTOMER. After this iteration the second phase begins. In this case, all of the approaches are able to find alternative plans that perform well. As shown in Figures 3 and 4, the initial plan basically traverses the graph from left to right. When PART is delayed this traversal becomes impossible. It is, however, possible to start at the other end of the query graph and traverse from right to left. This traversal picks up the other reducers in the query (the result computed in the first phase which contains the reducers REGION, and ORDER) before accessing the large LINEITEM relation.

In contrast to the cost-based approaches, the original heuristic-based scrambling algorithm [AFTU96] follows the policy of executing the left-most runnable sub-tree of the query plan, which in this case, results in joining LINEITEM and ORDER in the absence of the other reducer, REGION (the heuristic-based algorithm is not shown in the figure). With 1000 pages of memory, this join requires partitioning, which results in a large performance hit. For this experiment, the response time obtained with the heuristic-based approach jumps to 1621 seconds, for delays between 23 and 1542 sec-

[4] The extra materializations are due to our particular implementation of Pair on our simulator, and could be avoided by using a more sophisticated memory management approach in the simulator.

onds long. Thus, for many delay values, that algorithm performs significantly worse than simply waiting for the delayed relation to arrive.

We also ran this experiment for a memory allocation of 10,000 pages (not shown). In this case all of the cost-based approaches performed identically to the 1000 page case (because 1000 pages is sufficient to run the scrambled plans without partitioning). The main difference was that with this large memory (i.e., approximately 33 times more than what was allocated to the original query plan), the heuristic-based algorithm performed roughly as well as the cost-based approaches.[5] Its better performance here is due to the fact that the extra memory allows even the inefficient joins that it picks to run without partitioning. In this environment, the CPU costs are a negligible portion of the query execution time so avoiding partitioning leads to reasonable performance. The tradeoffs for the heuristic-based algorithm are similar for the rest of the experiments in this study, so we do not show any further results for that algorithm. Rather, we focus on the tradeoffs among the three cost-based approaches.

4.1.2 Query MQ8 - Small Memory Allocation

With the smaller memory allocation (300 pages, shown in Figure 6), the story changes significantly. 300 pages is sufficient to run the query when there are no delays with reasonable efficiency (the response time here is 790.5 seconds, only slightly higher than in the larger memory case). The smaller memory, however, causes problems when scrambling is required and results in different performance for the various approaches. Query scrambling starts by running the first phase on the subtree containing `REGION`, `NATION1`, and `CUSTOMER`. This result will be used by the subsequent steps of the different scrambling algorithms.

The IN approach is the most aggressive — it assumes that the delay of `PART` will be long, so it is willing to initiate a lot of scrambling work in order to be able to hide more delay. In this case, the IN approach simply pushes `PART` to the far right of the query plan, and joins the remaining relations in the same order as in the previous case. These joins are more expensive here, however, because the lack of memory results in more partitioning and thus, more local I/O. For delays less than 970 seconds, IN has the worst performance of the scrambling approaches, even performing worse than not scrambling for much of that range. Ultimately, however, at a delay of 1576 seconds, IN manages to perform nearly all of the work of the query during the delay, so its performance becomes nearly the same as the delay.

The ED approach is more conservative here. It begins by joining the result computed during the first phase (i.e. `REGION` $\bowtie$ `NATION1` $\bowtie$ `CUSTOMER`) with `ORDER`, and `NATION2` with `SUPPLIER` (this is the second step in the curve). It then brings `LINEITEM` over, writes it to the local disk, and waits for `PART` to arrive (at this point, its curve goes diagonal). This more conservative behavior results in better performance than IN for shorter delays, but ultimately worse performance for longer delays, since more work remains to be done when the tuples of `PART` eventually begin to arrive. Finally, Pair initially performs the same steps as ED, but at 953 seconds (roughly when ED stops scrambling) it chooses to perform a join that includes `LINEITEM`, which requires partitioning, and hence, is quite expensive. Given a long enough delay, this additional join will eventually pay off, with Pair having similar performance to IN after a delay of 2022 seconds. It is interesting to note that while Pair performs the same joins as IN, it has worse performance than IN in the delay range of 953 to 2022 seconds. This worse performance arises because Pair does not generate a complete plan, but rather, makes local decisions one pair at a time. It is not able to make intelligent decisions on how to partition the results of its joins, because it does not know if or how those results will be used in a subsequent operation. In other words Pair policy does not recognize interesting orders and therefore cannot asses the future savings due to executing a slightly more expensive plan which generates an interesting order. As a result, Pair simply materializes its intermediate results, and re-reads them to partition them later if necessary. This re-partitioning is expensive, because it generates significant amounts of *random* I/O.

4.2 Experiment 2 - Local Supplier Volume

We now turn to our second set of experiments, which uses a modified version of TPC-D Query 5 that we call MQ5. The SQL for this query is shown in Figure 7. This query is a 6-way join with two selection predicates. As shown in figure 8, the query graph of MQ5 contains a cycle, unlike the “chain” graph of MQ8 in the previous experiments.

```
SELECT N.NAME,L.EXTENDEDPRICE*(1-L.DISCOUNT)
FROM   CUSTOMER, ORDER, LINEITEM,
       SUPPLIER, NATION, REGION
WHERE  C.CUSTKEY       = O.CUSTKEY
       AND O.ORDERKEY  = L.ORDERKEY
       AND L.SUPPKEY   = S.SUPPKEY
       AND C.NATIONKEY = S.NATIONKEY
       AND S.NATIONKEY = N.NATIONKEY
       AND N.REGIONKEY = R.REGIONKEY
       AND R.NAME      = "AMERICA"
       AND O.ORDERDATE BETWEEN '95-01-01'
                           AND '95-12-31'
```

Figure 7: Modified Local Supplier Volume Query (MQ5)

Figures 9(a) and 9(b) show the initial optimized query plans for memory allocation of 1000 pages or greater and a memory allocation of 300 pages, respectively. Notice that

[5] Actually, in this case the original heuristic based algorithm works well with as few as 5000 pages, or 16.67 times more memory than allocated to the original query.

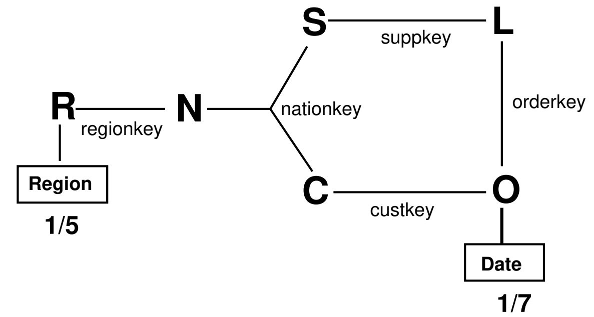

Figure 8: Query graph for query MQ5

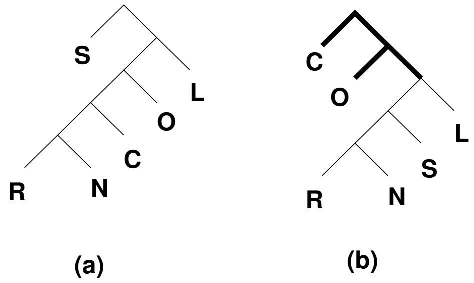

Figure 9: Query plans generated for (a) memory$\geq$1000 and (b) memory=300

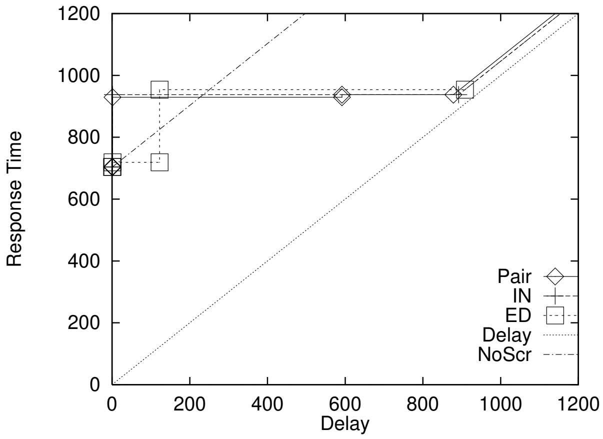

Figure 10: MQ5, CUSTOMER delayed, Mem=1000

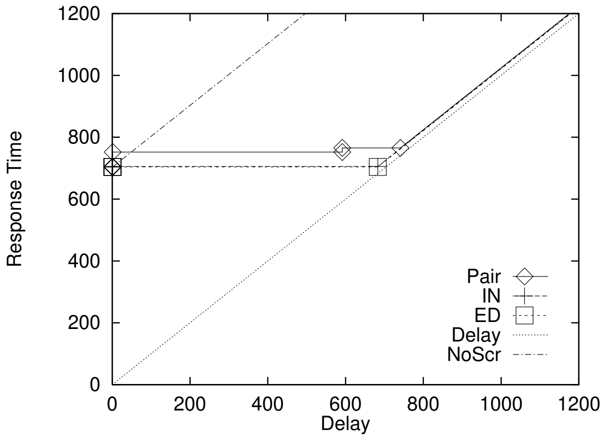

Figure 11: MQ5, CUSTOMER delayed, Mem=10000

with the larger memory allocation, the bulk of the query execution proceeds in a counter-clockwise direction around the join cycle. For the smaller allocation the execution proceeds in the opposite direction, and two of the joins require partitioning. In both memory cases we delay `CUSTOMER`. In the large memory case, the hash table for `SUPPLIER` will be built in the memory and `REGION` $\bowtie$ `NATION` will be computed before `CUSTOMER` is determined to be blocked. In the small memory case the delayed relation is encountered immediately.

4.2.1 Query MQ5 - Large Memory Allocations

Figure 10 shows the results for an initial delay on `CUSTOMER` with a memory allocation of 1000 pages. `CUSTOMER` is an important relation in the query plan because it helps transmit the selection predicates on `REGION` and `ORDER` to the large `LINEITEM` relation. In this case, all of the algorithms provide benefits over not scrambling beyond a delay of approximately 250 seconds, and hide nearly all of the delay when the duration is about 950 seconds. The Pair and IN approaches perform similarly here because they basically execute the same operations, even though Pair produces its plan one join at-a-time. Both of the approaches perform a join that requires partitioning (because it involves `ORDER` without first reducing it by joining it with `REGION` as is done in the initial plan). As a result, the response time of the scrambled plan is approximately 234 seconds longer than that of the initial plan (with no delay). In contrast, ED performs better for short delays (up to 122 seconds) due to its conservative approach. It first brings `ORDER` from the remote site and stores it on the local disk before committing to any other scrambling moves. Fetching `ORDER` has little risk for short delays; if `CUSTOMER` arrives during this time, `ORDER` can be used later in the query. ED's slight performance penalty between delays of 122 seconds and 908 seconds results from the fact that `ORDER` needs to be repartitioned if `CUSTOMER` does not arrive in time and ED re-scrambles. Thus, ED pays a small penalty for its conservative approach in this delay range in order to win its advantage for small delays.

The results for the 10,000 memory allocation are shown in Figure 11. In this case the same initial plan is used, but in the presence of delay, different scrambling plans are produced. Because of the larger memory, these scrambled plans can be executed with no partitioning. Thus, they all provide excellent protection from delays of `CUSTOMER`, up to approximately 683 seconds (or 97% of the non-delayed query response time). As was seen in the experiments with MQ8, Pair pays a slight penalty due to unnecessary materializations of temporary results to the local disk.

4.2.2 *Query MQ5 - Small Memory Allocation*

We also experimented with query MQ5 using the small (300 pages) memory allocation when CUSTOMER is delayed (not shown). In this case all of the scrambling approaches perform well (i.e., the response time is basically flat for delay durations up to the response time of the non-delayed query). This is because the first phase of scrambling (i.e., rescheduling) is able to perform all of the joins except the one involving CUSTOMER (see Figure 9(b)) without needing to create any new operators. That is, the entire sub-tree which computes ORDER ⋈ REGION⋈ NATION ⋈ SUPPLIER ⋈ LINEITEM is simply executed by rescheduling. Thus, the different cost-based approaches do not come into play here.

4.3 Experiment 3 - Product Type Profit Measure

The third (and final) set of experiments we describe were performed using a modified version of TPC-D Q9, shown in Figure 12. The query graph for this query is shown in Figure 13. In this case, we obtained similar results for all three memory sizes, so we show results only for the 10,000 page allocation. The initial query plan in this case is shown in Figure 14. For this experiment, we delay PART, the only reducer in the query.

```
SELECT N.NAME, O.ORDERDATE.YEAR,
       L.EXTENDEDPRICE*(1-L.DISCOUNT) -
          (PS.SUPPLYCOST * L.QUANTITY)
FROM   PART, SUPPLIER, LINEITEM, PARTSUPP,
       ORDER, NATION
WHERE  S.SUPPKEY       = L.SUPPKEY
       AND PS.SUPPKEY  = L.SUPPKEY
       AND PS.PARTKEY  = L.PARTKEY
       AND P.PARTKEY   = L.PARTKEY
       AND O.ORDERKEY  = L.ORDERKEY
       AND S.NATIONKEY = N.NATIONKEY
       AND P.NAME LIKE '%magenta%'
```

Figure 12: Modified Product Type Profit Measure Query (MQ9)

The performance of the scrambling approaches for varying delays of PART is shown in Figure 15. The IN approach joins all relations other than PART, which in query MQ9, results in intermediate results that are 20 times larger than if PART had been used. These intermediate results propagate through the entire plan, resulting in a very high-risk move. The Pair approach performs the same joins as IN, but performs them one-at-a-time. This has two effects: it reduces risk for short delays, but also incurs additional overhead for long delays, due to the need to partition intermediate results that it has saved to the local disk (a similar effect was seen in query MQ8). The more conservative ED approach performs much better than IN and as good as the Pair policy for delays up to about 2000 seconds (i.e., more than twice the non-delayed response time). ED avoids joining large relations, choosing rather to simply wait for the delayed relation beyond a certain point. As usual this conservatism results in a penalty for longer delays, but in this case, the penalty is quite small, and is more than outweighed by the advantages for shorter delays.

The preceding experiment demonstrated clearly the potential benefits of making conservative scrambling decisions. Recall that the ED and IN approaches both incorporate a "risk/benefit" knob (which is used by the response time based optimizer), that prevents the policies from choosing very high-risk plans for relatively small potential gains over lower-risk plans. In all of the experiments described so far, this knob was set at 0.01 (as described in Table 2), which means that the optimizer is willing to give up 0.01 units (e.g., seconds) of potential benefit for long delays to get a plan whose total work is 1 unit less, which results in less risky behavior for short delays.

To study the effect of the setting of this knob, we repeated this experiment using several different values (0, 0.10, 0.20, and 0.30). The results using 0 and 0.10 were similar to the results using the default setting (0.01). The more conservative settings did have an impact however. The results with a setting of 0.30 are shown in Figure 16 (the ones using 0.20 are similar). First, the knob has no effect on the Pair approach, because that approach is based only on total work; it has no notion of risk vs. benefit. In contrast, both the ED and IN approaches are effected, but the more conservative setting has a greater impact on IN. Overall, the conservative setting results in substantially better performance for IN with short delays. For example, for delays between 1 second and 837 seconds, the response time using IN is 1184 seconds, and for delays between 837 and 1073 seconds the response time using IN is 1300 seconds, compared to 2263 seconds for these ranges with the aggressive setting. For these additional benefits, IN pays only a 46 second cost in terms of the amount of delay that it can hide for long delays, and this cost only arises for delays over 2000 seconds. Thus, the advantages for short delays clearly outweigh the costs at higher delays. It is interesting to note that with the conservative knob setting, IN requires a second scrambling iteration (for delays greater than 837 seconds), because its first iteration produces a plan that leaves work remaining to be done, even without the delayed relation.

The more conservative setting has a lesser effect on the ED approach. This is because ED already favors conservative decisions for small delays. In this case, the higher knob value prevents ED from performing its last iteration, which has an ultimate benefit of 5 seconds, at a risk of 472 seconds. This results in ED having better performance here for delays between 1072 and 1539 seconds than it did with the more aggressive knob setting.

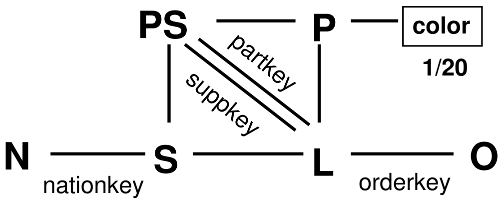

Figure 13: Query graph for query MQ9

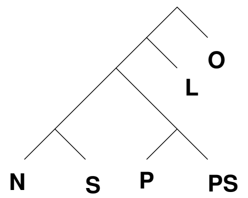

Figure 14: Query plan generated for memory=10000

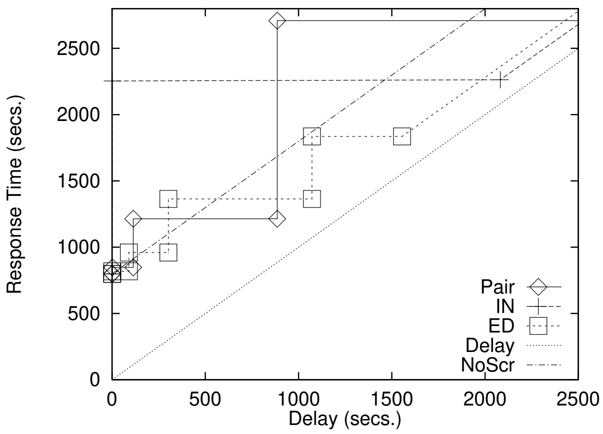

Figure 15: MQ9, PART delayed, Mem=10000, RBknob = 0.01

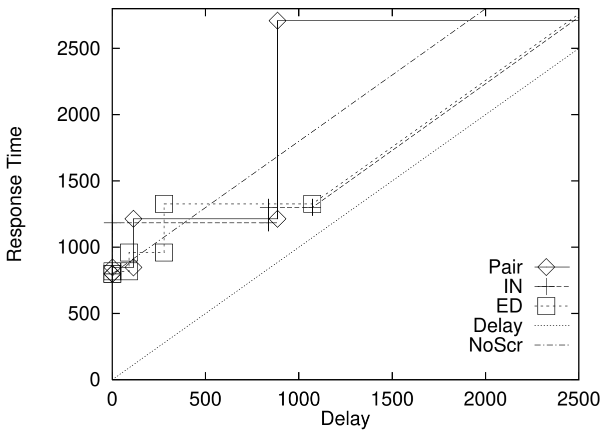

Figure 16: MQ9, PART delayed, Mem=10000, RBknob = 0.30

4.3.1 Summary of Results

The experiments we have described in this section demonstrate several important results for cost-based query scrambling. We briefly summarize those results here. First, the experiments showed that with sufficient memory, all of the cost-based approaches are able to effectively hide initial delays for realistic data processing queries. When the delayed relation is encountered early in the query execution, a delay as long as the normal (non-delayed) response time of the query can be almost completely absorbed. In contrast, the original heuristic-based algorithm can actually perform significantly worse than simply waiting for the delay to end unless substantial extra memory is dedicated to scrambling.

Second, for the cost-based approaches, in the absence of a reasonable prediction of delay duration there is a tradeoff between conservative approaches, which are safer for short delays, and more aggressive approaches which lead to bigger savings in the event of long delays. In general, the amount of delay that can be hidden by scrambling (in the absence of creating additional parallelism, as is discussed in [AFT98]) is limited by the normal response time of the query. This is because scrambling hides delays by performing other useful work, so its ability to hide delay is limited by the amount of useful work that can be done. Thus, as the delay increases beyond the normal response time of the original query, the benefits of scrambling *as a percentage of total execution time* begin to decrease. This argument would lead towards favoring more conservative policies rather than taking larger risks.

Third, as the memory available for scrambling is reduced, scrambled plans in general become more expensive and hence, a longer delay duration may be required in order for scrambling to pay off. Thus, in a low-memory situation scrambling becomes less conservative, and therefore, in the absence of predictions of delay durations, more dangerous.

Fourth, we showed how the aggressiveness of the IN and ED policies can be adjusted through the use of a parameter that tells the optimizer to give up potential gains for long delays in order to reduce risk for short delays. As stated above, this tradeoff makes sense in the absence of reasonably accurate predictions of delay durations.

A final important result from these cases, is that approaches that lack a global view of the scrambled plan (e.g., Pair) may perform unnecessary work. By considering only pairs of relations the Pair policy, is unable to pick slightly suboptimal plans that generate interesting orders. In order to have a complete (and reasonable) scrambled plan, however, one must use an optimizer that uses *response time* as its objective function. A response time-based optimizer allows the delayed relation to be placed at its proper point in the plan (for a given predicted delay), which allows a complete alternative query plan to be generated.

We have also conducted experiments using more than one delayed relation on synthetically generated queries. We have found that when more relations are delayed, the risk associated with each scrambling decision is increased, favoring more conservative algorithms.

5 Related Work

We now briefly discuss related work. The Volcano optimizer [CG94, Gra93] provides dynamic query scheduling by introducing choose-plan operators into a query plan above a set of alternative subplans in order to compensate for the lack of information about system parameters at compile time. At query startup time the appropriate subplan is chosen depending on the current value of the parameters. [INSS92] proposes a related approach that generates multiple alternative plans, and chooses among them when the query is initialized. Neither of these approaches, however, can adapt to changes in the system parameters that occur *during* the query execution.

Rdb/VMS uses a different approach as described in [Ant93]. In this approach, multiple different executions of the same logical operator are started at the same time. When one execution of an operator is determined to be better, the other execution is terminated, and the winner is executed to completion.

The work most closely related to ours is the MIND heterogeneous database project [ONK+97], which performs optimization during the query execution. A query is divided into subqueries and each subquery is sent to a participating site for execution. The results are then composed incrementally by dynamically introducing operators that process them as the results arrive. As such, their algorithm resembles our Pair algorithm with a different set of heuristics that rely on statistical techniques in order to avoid bad decisions.

In [DSD95] the response time of queries is improved by reordering left-deep join trees into bushy join trees and creating subtrees without increasing the cost. Several reordering algorithms are presented. Although this work is limited to left-deep queries and assumes that reordering is done entirely at compile time, one can still use it to *bushify* the plans during run time, possibly at the expense of a slight increase in total work. Bushy plans are generally less vulnerable to delays since different branches of tree can be found that are not directly affected by the delayed relations; such subtrees can be executed independently.

The research prototype Mermaid [CBTY89] and its commercial successor InterViso [THMB95] are heterogeneous distributed databases that 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 the intermediate relation cardinalities and system 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.

6 Conclusions and Future Work

In this paper, we proposed and investigated three different approaches to using a query optimizer to help make intelligent choices during query scrambling. Two of the approaches used an optimizer with an objective function based on response time, while the other approach used a more traditional optimizer based on total work. In general, the use of a response time optimizer has the advantage of being able to construct complete query execution plans that include access to delayed data. Based on an estimate of the expected delay duration, the optimizer places the accesses to delayed data to the proper place in the plan.

Given the poor state of current estimation techniques for wide-area data access, we proposed two different ways of using a response-time optimizer. We demonstrated that these approaches exhibit fundamental tradeoffs between risk aversion (for short delays) and the ability to hide large delays. However, we also showed that in many cases the algorithms were very effective at hiding delays over a wide range. In the best cases, the approaches were able to hide delays of a duration equal to the response time of the query in a non-delayed situation.

Due to the growing importance of wide-area data access, particularly in chaotic environments such as the Internet, there is much future work that can be done on scrambling and related dynamic techniques. First, although not discussed in this paper, the scrambling techniques we have described here can be adapted for use with other types of delay, such as bursty arrival, in which sites repeatedly stall and recover. As described in [AFT98] such delays introduce a number of scheduling and memory management issues that must be addressed by scrambling. In addition, we would like to investigate the use of delay prediction techniques in the scrambling approaches. Finally, as described in [ABF+97], additional techniques are required for dealing with very long periods of outage. Unlike scrambling, these techniques necessarily change the answer that is returned to the user, and thus, raise a number of interesting semantic questions in addition to the performance-oriented questions that we have addressed here.

Our current focus is on incorporating the cost based query scrambling into the query engine of PREDATOR [SLR97] and extending it by adding remote access capability. We plan to use this system as a test bed for query scrambling over the Internet.

References

[ABF+97] L. Amsaleg, P. Bonnet, M. Franklin, A. Tomasic, and T. Urhan Improving Responsiveness for Wide-Area Data Access. *IEEE Data Engineering Bulletin*, Vol. 20, No. 3.

[AFT98] L. Amsaleg, M. J. .Franklin, and A. Tomasic. Dynamic Query Operator Scheduling for Wide-Area Remote Access. *Journal of Distributed and Parallel Databases*, Vol. 6, No. 3, July 1998.

[AFTU96] L. Amsaleg, M. J. .Franklin, A. Tomasic, and T. Urhan. Scrambling Query Plans to Cope With Unexpected Delays. *PDIS Conf.*, Miami, USA, 1996.

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

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

[CD96] M. J. Carey, D. J. DeWitt. Of Objects and Databases: A Decade of Turmoil. *22nd VLDB Conf.*, Bombay, India, 1996.

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

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

[GHK92] S. Ganguly, W. Hasan, R. Krishnamurthy. Query Optimization for Parallel Execution. *ACM SIGMOD Conf.*, San Diego, CA, 1992.

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

[IK90] Y. E. Ioannidis, and Y. Kang. Randomized algorithms for optimizing large join queries. *ACM SIGCOM Conf.*, Atlantic City, NJ, 1990.

[INSS92] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. *18th VLDB Conf.*, Vancouver, BC, Canada, 1992.

[IW87] Y. E. Wong, and E. Wong. Query optimization by simulated annealing. *ACM SIGMOD Conf.*, San Fransisco, CA, 1987.

[Kim95] M. Kim. *Modern Database Systems: the Object Model, Interoperability, and beyond*. ACM Press, New York, NY, 1995.

[MMM97] A. O. Mendelzon, G. A. Mihaila, T. Milo. Querying the World Wide Web. *PDIS Conf.*, Miami, USA, 1996.

[ML86] L. F. Mackert, G. M. Lohman R* Optimizer validation and Performance Evaluation for Distributed Queries *12th VLDB Conf.*, Kyoto, Japan, 1986.

[ONK+97] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, A. Dogac. Dynamic Query optimization in Multidatabases. *Data Engineering Bulletin*, Vol. 20, No. 3, September, 1997.

[RAH+96] M. T. Roth, M. Arya, L. M. Haas, M. J. Carey, W. F. Cody, R. Fagin, P. M. Schwarz, J. Thomas, E. L. Wimmers. The Garlic Project. *ACM SIGMOD Conf.*, Montréal, Canada, 1996.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. *ACM SIGMOD Conf.*, Boston, USA, 1979.

[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.

[Sha86] L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Trans. on Database Systems*, Vol. 11, No. 3, 1986.

[SLR97] P. Seshadri, M. Livny, R. Ramakrishnan. The Case for Enhanced Abstract Data Types. *23rd VLDB Conf.*, Athens, Greece, 1997.

[THMB95] M. Templeton, H. Henley, E. Maros, and D. Van Buer. InterViso: Dealing with the Complexity of Federated Database Access. *VLDB Journal*, 4(2), 1995.

[Tra97] Transaction Processing Council. TPC Benchmark D Standard Specification, Rev. 1.2.3.

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