Knowledge and Information Systems (2000) 2: 373–385
© 2000 Springer-Verlag London Ltd.

Knowledge and Information Systems

Short Paper

A Probe-Based Technique to Optimize Join Queries in Distributed Internet Databases

Cyrus Shahabi, Latifur Khan, Dennis McLeod

Integrated Media Systems Center and Department of Computer Science,
University of Southern California, Los Angeles, CA, USA

Abstract. An adaptive probe-based optimization technique is developed and demonstrated in the context of an Internet-based distributed database environment. More and more common are database systems which are distributed across servers communicating via the Internet where a query at a given site might require data from remote sites. Optimizing the response time of such queries is a challenging task due to the unpredictability of server performance and network traffic at the time of data shipment; this may result in the selection of an expensive query plan using a static query optimizer. We constructed an experimental setup consisting of two servers running the same database management system connected via the Internet. Concentrating on join queries, we demonstrate how a static query optimizer might choose an expensive plan by mistake. This is due to the lack of a priori knowledge of the run-time environment, inaccurate statistical assumptions in size estimation, and neglecting the cost of remote method invocation. These shortcomings are addressed collectively by proposing a probing mechanism. An implementation of our run-time optimization technique for join queries was constructed in the Java language and incorporated into an experimental setup. The results demonstrate the superiority of our probe-based optimization over a static optimization.

Keywords: Distributed databases; Internet databases; Join queries; Query optimization

1. Introduction

A distributed database is a collection of partially independent databases that share a common schema, and coordinate processing of non-local transactions. Processors communicate with one another through a communication network (Silberschatz et al, 1997; Yu and Meng, 1998). We focus on distributed database systems with sites running homogeneous software (i.e., database management systems DBMS) on heterogeneous hardware (e.g., PC and Unix workstations)

Received 6 February 1999
Revised 15 February 2000
Accepted 10 May 2000

connected via the Internet. The Internet databases are appropriate for organizations consisting of a number of almost independent sub-organizations such as a university with many departments or a bank with many branches. The idea is to partition data across multiple geographically or administratively distributed sites where each site runs an almost autonomous database system.

In a distributed database system, some queries require the participation of multiple sites, each processing part of the query as well as transferring data back and forth among themselves. Since usually there is more than one plan to execute such a query, it is crucial to obtain the cost of each plan, which greatly depends on the amount of participation by each site as well as the amount of data shipment between the sites. In an Internet environment which is based on a best-effort service, there are a number of unpredictable factors that make the cost computation complicated (Paxson and Floyd, 1997). A *static* query optimizer that does not consider the characteristics of the environment or only considers the a priori knowledge on the run-time parameters might end up choosing expensive plans because of these unpredictable factors. In the following paragraph, we explain some of these factors via simple examples.

Participating sites (or servers) of Internet database systems might have different processing powers. One site might be a high-end multiprocessor system while the other is a low-end PC running (say) Windows NT. In addition, since most queries are I/O intensive, a site having faster disk drives might observe a better performance. In an Internet-based environment these sites might be dedicated to a single application or multiple simultaneous applications. Moreover, the workload on each server might vary over time. The network traffic is another major factor. It is not easy to predict network delay in the Internet owing to variability of effective network bandwidth among the sites. A query plan which results in less tuple shipments might or might not be superior to the one preferring extensive local processing, depending on the network traffic and server load at the time of query processing. Briefly, there is just too much uncertainty and a very dynamic behavior in an Internet-based environment that makes the cost estimation of a plan a very sophisticated task.

There has been extensive research in query processing and optimization in both distributed databases and multi-databases (Bernstein et al, 1981; Apers et al, 1983; Bodorik et al, 1992; Zhu and Larson, 1994; Amsaleg et al, 1997; Evrendile et al, 1997). Among those, only a few considered run-time parameters in their optimizers. Briefly, most of these studies propose a *detective* approach to compensate for lack of run-time information, while our approach is *predictive* and prevents the selection of expensive queries at run time. Furthermore, previous studies on distributed query optimization assume that network bandwidth is dedicated to the system and remains constant and is proportional to the size of data transferred. In this paper, we demonstrate the importance and effectiveness of a probe-based optimization technique for join queries in the Internet databases which estimates network bandwidth on the fly. We focused on *join* queries because join operation is not only frequently used but also expensive (Yu and Meng, 1998).

In order to demonstrate the importance of run-time optimization, we implemented an experimental distributed database system connected through the Internet. Our setup consists of two identical servers[1] both running the same object-relational DBMS (i.e., Informix Universal Server; Informix, 1997) connected via

[1] By spawning a number of auxiliary processes on one of the servers, we emulated an environment with heterogeneous servers.

the Internet. We then split the BUCKY database (from the BUCKY benchmark; Carey et al, 1997) across the two sites. We implemented a probe-based run-time optimization module for join queries in Java language. The optimizer first issues two probe queries each striving to estimate the cost of either semi-join or simple join plans. Consequently, the cheapest plan will be selected. The query optimizer of a distributed database system can be extended with our probe queries to capture run-time behavior of the environment. Furthermore, as a byproduct, the result of the probe queries can be utilized for estimating the size of intermediate relations in a join plan. This estimation is shown to be less sensitive to statistical anomalies as compared to that of static optimizers. Finally, the probe-based technique identified some hidden costs (e.g., the cost of remote invocation of methods with RMI) that should be considered in order to select the cheapest plan. That is, our probing mechanism can capture any surprises associated with specific implementations (e.g., RMI in our case) which can never be accounted for by static optimizers. The experiments show that for expensive queries processing many tuples the response time can be improved, on average, by 32.5% over a static optimizer, while the probing overhead only results in an average of 6.4% increase in response time.

The remainder of this paper is organized as follows. Section 2 states the problem, reviews a conventional solution, and finally explains our proposed extensions to capture run-time parameters and utilize them to improve the optimizer. Section 3 consists of a performance study to compare the performance of our run-time optimization technique with that of a static optimizer. Finally, Section 4 concludes the paper and provides an overview on our future plans.

2. Run-Time Optimization for Join Queries

In this section, we start by defining the problem of query optimization for join queries in distributed databases. Subsequently, we briefly describe a conventional solution to the problem. Finally, we propose our probing mechanism and compare it with the conventional solution.

2.1. Problem Statement

Suppose there are two relations R_ℓ at *local* site S_1 and R_r at *remote* site S_2. Consider the query that joins R_ℓ and R_r on attribute A and requires the final result to be at[2] S_1. The objective is to minimize the query response time. A straightforward plan, termed *simple join plan* (P_j), is to send relation R_r to site S_1 and perform a local join at S_1. This approach observes one data transfer and one join operation. The second plan employs semi-join[3] and is termed *semi-join plan* (P_{sj}). This strategy incurs two data transfers and also performs join twice. The decision between choosing one plan over the other is not straightforward and depends on a number of parameters such as the size and cardinality of relations R_ℓ and R_r. Therefore, the problem is how to decide which plan to choose in order to minimize the response time of a certain join query.

[2] For the remainder of this paper, we focus on the same exact scenario without loss of generality.

[3] For a detailed description of semi-join consult Ceri and Pelagatti (1984) and Yu and Meng (1998).

2.2. Static Query Optimizer

In this section, we explain a conventional method (Bernstein et al, 1981; Apers et al, 1983; Ceri and Pelagatti, 1984) to estimate the costs associated with both simple join and semi-join plans. Since the parameters used for this cost estimation are all known a priori before the execution of the plans, this query optimizer is termed *Static Query Optimizer (SQO).*

Given the number of tuples in R_r as N_r and the size of a tuple in R_r as S_{R_r}, the cost of simple join is trivially computed as follows:

$$Cost(P_j) = C_0 + C_1 \times S_{R_r} \times N_r \tag{1}$$

where C_0 is the cost to start up a new connection and C_1 is the communication cost per byte transfer.

The computation of the cost of semi-join plan is more complicated.

Let us denote the size of the common attribute A as S_{R_A}, and the number of distinct values for attribute A in local relation (R_ℓ) as N_ℓ. $\Pi_A(R_\ell)$ relation is transferred from S_1 to S_2 and joined with R_r at S_2 ($R' = \Pi_A(R_\ell) \bowtie R_r$). Suppose $|R'|$ is the cardinality of relation R'.

Therefore, the overall cost for the semi-join plan is

$$Cost(P_{sj}) = 2 \times C_0 + C_1 \times (S_{R_A} \times N_\ell + S_{R_r} \times |R'|) \tag{2}$$

SQO will choose the semi-join plan if $Cost(P_{sj}) \leqslant Cost(P_j)$, or if

$$C_0 + C_1 \times (S_{R_A} \times N_\ell + S_{R_r} \times |R'|) \leqslant C_1 \times S_{R_r} \times N_r \tag{3}$$

SQO can examine the above inequality accurately only if it has all the required information (e.g., N_ℓ, S_{R_r}) a priori. Note that C_1 is simply a reciprocal of network bandwidth. However, over the Internet, effective network bandwidth between two sites is extremely difficult to estimate because it is changing more frequently. Finally, SQO needs to estimate the size of intermediate results (i.e., $|R'|$). One estimation is as follows:

$$|R'| = domain(A) \times sel(R_\ell, A) \times sel(R_r, A) \tag{4}$$

where $sel(R_\ell, A)$ and $sel(R_r, A)$ are the selectivity of attribute A in relations R_ℓ and R_r, respectively. Equation (4) is based on two assumptions. First, it assumes that the domain of A is discrete and can be considered as A's sample space. Second, tuples are distributed between R_ℓ and R_r independent of the values of A. That is, there is no correlation between R_ℓ and R_r based on the join attribute A.

2.3. Our Proposed Solution

We now describe our run-time optimization (RTO) technique, which is an extension to SQO. To summarize, RTO first submits two *probe* queries to estimate the run-time costs corresponding to plans P_j and P_{sj} by measuring the response time observed by each probe query. Subsequently, it replaces $C_1 \times S_{R_A}$ and $C_1 \times S_{R_r}$ in equation (3) by the estimated costs. In addition, RTO analyzes the results of the probe queries to estimate the size of R' more accurately.

2.3.1. Probe Queries

Our main objective is to modify the SQO main equation (equation 3) in order to take the run-time parameters into the consideration. To achieve this, we submit the following two probe queries to collect some parameters at run-time.

Probe query A. The first probe query strives to replace the term $C_1 \times S_{R_A}$ of equation (3) with a more accurate estimation. This is because $C_1 \times S_{R_A}$ is based on the simplistic assumption that communication cost is a linear function of the amount of data transferred and network bandwidth ($1/C_1$) is also available. This probe sends the A attribute of X number of tuples of R_ℓ, denoted R_{XA}, from local site S_1 to remote site S_2; joins R_{XA} with R_r at remote site S_2; and receives back the size of the result denoted as X_j. The time to execute this probe query is measured and is then normalized by dividing it by X. The result is the cost of this probe and is denoted by $C_{\ell 2r}$. To illustrate the costs that have been captured by $C_{\ell 2r}$, consider the following equation:

$$C_{\ell 2r} = \frac{S(X) + RIC_Q + RIC(X) + JC_r}{X} \tag{5}$$

In equation (5), $S(X)$ is the cost to ship X tuples (each tuple consists of only one attribute A) from S_1 to S_2, RIC_Q is the remote invocation cost for the join operation at S_2, $RIC(X)$ is the remote invocation cost to insert X tuples into S_2, and JC_r is the cost to perform the join operation at[4] S_2. Observe that as a byproduct, R' can now be estimated more accurately because X_j is the number of tuples in R' if R_ℓ had X tuples. Now that R_ℓ has N_ℓ tuples then the size of R' can be estimated as

$$Sample\_estimate(R') = \frac{X_j \times N_\ell}{X} \tag{6}$$

Probe query B. The second probe query strives to replace the term $C_1 \times S_{R_r}$ of equation (3) with a more accurate estimation. It receives X number of tuples of R_r, denoted R_X, from remote site S_2; joins R_X with R_ℓ at local site S_1; and measures the time to complete this process. This time is then normalized by dividing it by X and is the cost of this probe (denoted by $C_{r2\ell}$). To illustrate the costs that have been captured by $C_{r2\ell}$, consider the following equation:

$$C_{r2\ell} = \frac{RIC(X) + S(X) + JC_\ell}{X} \tag{7}$$

In equation (7), $S(X)$ is the cost to ship X tuples from S_2 to S_1, JC_ℓ is the cost to perform the join operation at S_1, and $RIC(X)$ is the remote invocation cost to request X tuples from S_2. In equations (5) and (7), $S(X)$ is capturing the following run-time parameters:

$$S(X) = Delay_{send}(X) + Delay_{network}(X) + Delay_{receive}(X) \tag{8}$$

where $Delay_{send}(X)$ is the time required at the sender site to emit X tuples, $Delay_{receive}(X)$ is the time required at the receiver site to receive X tuples and $Delay_{network}(X)$ is the network delay.

[4] We ignored the cost of returning X_j to S_1 since X_j is only a single integer.

Now we can modify equation (3) of SQO as follows:

$$N_\ell \times C_{\ell 2r} + Sample\_estimate(R') \times C_{r2\ell} \leqslant N_r \times C_{r2\ell} \qquad (9)$$

In equation (9), the terms $C_1 \times S_{R_4}$, and $C_1 \times S_{R_r}$ of equation (3) are replaced by $C_{\ell 2r}$ and $C_{r2\ell}$; and R' is computed using equation (6) instead of equation (4).

Selection of X tuples. Both probe queries transfer X tuples for their estimations. Therefore, the value of X has an impact on the accuracy of the estimations. Trivially, the larger the value of X the more accurate the estimation. However, large value of X results in more overhead observed by the probe queries. In our experiments, we varied X from 1% to 10% of N_ℓ. Besides the value of X, the way that X tuples are selected impacts the estimated size of R'. This *sampling* should be done in a way such that X is a good representative of R_ℓ (see Haas and Swami, 1995, for more details).

Scalability. Although we describe our probe queries for joins between two relations (i.e., 2-way join), the technique is indeed generalizable to a k-way join. When joining k relations on a common attribute, the k-way join can be considered as $(k-1)$ 2-way joins. The purpose of this join is to reduce the size of relations and determine which tuples of relations are participating in the final result. Finally, all processed relations are transmitted to a final site where joins are performed and the answer to the query obtained (Chen and Victor, 1984). Hence, the optimization challenge in the reducing phase is to identify the optimal execution order of the k-way join. Static optimizers for distributed databases address this challenge by sorting the k relations in ascending order of their volumes (Apers et al, 1983). Assuming the communication cost is independent of the network load and is linearly proportional to the volume of transferred data, then this sorted order specifies the optimal execution order. For the Internet, variable network load should be taken into account to identify the optimal plan. Therefore, our probe-based technique can be utilized in a similar way to estimate the communication cost among all the k participating sites (assuming one relation per site). As a result $k \times (k-1)$ probe queries are generated among the k sites. One can argue that our technique is not scalable owing to the extensive increase in the number of probe queries in a k-way join optimization. However, it is important to note that these probe queries are independent of each other and thus can be executed in parallel. Therefore, the overhead observed for k-way join optimization is equal to the maximum delay incurred among all the probe queries. After estimating the communication costs from site to site, the optimal execution order is determined by the ascending order of number of tuples transferred multiplied by the communication cost between the corresponding sites.

2.4. Analysis and Comparison

In this section, we analyze why equation (9) can now capture run-time behavior and estimate the size of intermediate relations more accurately than SQO.

Communication cost. Almost all the previous studies on distributed query optimization assumed communication cost is proportional to the size of data transferred. They also assume network bandwidth information is available to the system and remains constant. The same assumptions have also been made by the static query optimization technique discussed in this paper (see Section 2.2). However, researchers (Paxson, 1997) in the network community demonstrate that over the Internet it is hard to estimate the effective network bandwidth. In addition, network bandwidth between two sites varies significantly with time owing to the Internet dynamics. In our experiments, however, we observed that the communication cost is indeed a linear function of the *number* of tuples transferred. This is because the granularity of data transfer in our experiments was in tuples. With RTO, $C_{\ell 2r}$ and $C_{r2\ell}$ are the linear extrapolation of the time to move X tuples and hence are based on *number* of tuples moved at the time of query execution between the two participating sites. In addition, the size of tuples is also taken into consideration by measuring the actual time to transfer X tuples of size S_{R_A} and S_{R_r}. By doing this, we are inherently capturing the available network bandwidth between two sites at run time. Note that the same argument holds if the granularity of data transfer is in blocks instead of tuples.

Remote invocation cost. As discussed in Section 3.1, in our experimental setup Remote Method Invocation (RMI) was employed in order to access a remote server. An interesting distinction between the simple join and semi-join plan is that in general the semi-join plan uses remote invocation more often as compared to that of the simple join plan. To illustrate, P_{sj} utilizes remote invocation N_ℓ times to insert tuples into S_2, one time to execute the join remotely at S_2, and R' times to fetch the semi-join results back to S_1. This is while P_j utilizes RMI only N_r times to fetch the remote tuples into S_1. Obviously, this hidden RMI cost has not been captured by SQO because this cost is very specific to our implementation and experimental setup. The interesting observation, however, is that this cost has automatically been captured by $C_{r2\ell}$ and $C_{\ell 2r}$.

Load cost. From equation (3), it is obvious that SQO does not consider the time to process different operations such as project, join, and semi-join which are impacted by server workload. This is because it assumes that communication cost is the dominant factor in estimating the cost of a plan. With RTO, it is trivial from equations (5), (7), and (8) that the workload of the server can be captured by $C_{r2\ell}$ and $C_{\ell 2r}$ due to the following terms: $JC_r, JC_\ell, Delay_{send}$, and $Delay_{receive}$. Hence, another distinction between P_{sj} and P_j can be captured by our RTO. That is, semi-join performs two light joins, one at a remote site and the other local, while simple join only performs one heavy but local join operation. Further, a heavily loaded server also impacts the communication cost since it sends and receives tuples slower than a lightly loaded server (i.e., $Delay_{send}$ and $Delay_{receive}$). Consequently, it is not straightforward to model the impact of load on the cost of a plan. This is exactly why our probing mechanism can automatically capture these chaotic behaviors and aggregate them out within two simple terms of $C_{r2\ell}$ and $C_{\ell 2r}$.

Statistical assumptions. Regarding the statistical assumptions, RTO has two major advantages over SQO. First, RTO does not rely on remote profiles. Accessing metadata from the remote sites is not easy because statistic profiles are changed frequently and hence the process of collecting and updating the statistical information about the remote site is expensive. Recall that while SQO needs the value of S_{R_r} and $sel(R_r, A)$ for its computations, RTO relies on neither of these values. Second, RTO is less sensitive to the statistical anomalies as compared to SQO. RTO estimates the size of R' by sampling (see equation 6) and with RTO A's sample space is R_ℓ; moreover, it utilizes the entire R_r which is R_r's best possible sample. In addition, if there is a correlation between the two relations, it will impact X_j (in equation 6) accordingly. Therefore, a positive correlation results in higher value of X_j and vice versa.

2.5. An Adaptive Optimization Technique

In some cases, a single probing may not be enough to predict the run-time environment during the original query execution time. This is because some queries might take minutes to execute and hence there is a possibility of changes in the run-time parameters. Due to unpredictability of server performance and network traffic, the run-time environment can be changed. Therefore, we partition a join query into K series of smaller joins. Subsequently, for each smaller join, we re-evaluate the run-time parameters and make a decision either to continue with the current plan or switch to another plan. Our technique, however, does not treat each smaller join in isolation. It ensures that no smaller join performs redundant work that has already been done by the previous joins in the series. Briefly, the optimizer collects statistics to update the cost model at each re-evaluation point, termed *cost-update points*. Using the updated cost model, costs of different plans to complete the query are estimated and the optimizer chooses the least expensive one. To achieve this, we need to recompute $C_{\ell 2r}$ and $C_{r2\ell}$ at each cost-update point. For most of the cases, our adaptive technique can estimate $C_{\ell 2r}$ and $C_{r2\ell}$ by just timing the execution of the plan as it progresses. Hence, new probe queries are not required to be sent explicitly. For other cases, it needs to submit new probe queries.

3. Performance Evaluation

We conducted a number of experiments to demonstrate the superiority of RTO over SQO for join queries. In these experiments, first we varied the workload on the two servers in order to simulate a heterogeneous environment and/or variable run-time behavior of the environment. Our experiments verified that RTO can adapt itself to workload changes and always chooses the best plan, while SQO's decision is static and a specific plan is always chosen independent of the load on the servers. Second, our experiments showed that even in case of a balance load RTO outperforms SQO because it captures both the communication cost and the overhead attributed to a specific implementation setup (e.g., RMI cost) correctly.

3.1. Experimental Setup

Figure 1 depicts our experimental setup, which consists of two sites S_1 and S_2 that are not within a LAN but within the campus area network (CAN). The sites are Unix boxes with an identical hardware platform (a SUN Sparc Ultra 2 model with 188 MB of main memory and 100 clock ticks/second speed). Note that in our experiments we degrade the performance of one server by loading it with additional processes and emulating an environment with heterogeneous servers. Each process increases server disk I/O by repeatedly running Unix 'find' system call. The additional load is quantified by the number of these processes spawned on a server.

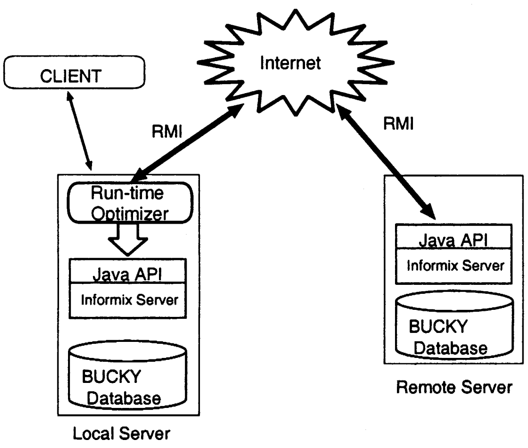

Fig. 1. Experimental setup.

Each site runs an Informix Universal Server (IUS), which is an object-relational DBMS. The run-time optimizer and its different plans are implemented in Java. The run-time optimizer communicates with the database servers through Java API, which is a library of Java classes provided by Informix. It provides access to the database and methods for issuing queries and retrieving results. From applications running on one site, RMI is used to open a connection to the database server residing on the other site. The BUCKY database, from the BUCKY benchmark (Carey et al, 1997), was distributed across the two sites. The queries are submitted to site S_1 as a local server and might require data to be shipped from site S_2, which is the remote server. RTO resides at S_1 and employs RMI to access the remote site. We concentrate on the two *TA* and *PROFESSOR* relations of BUCKY. The *TA* relation (or R_ℓ) at S_1 and the *PROFESSOR* relation (or R_r) resides at S_2. The number of tuples per relation residing on each site has been varied for our experiments. We fixed the total number of tuples (i.e., $N_\ell + N_r$) at 25,000. Without loss of generality and to simplify the experiments we assumed no duplications in the relations.

The join query is: *Find the Name, Street, City, State, Zipcode for every TA and his/her advisor*, in SQL:

```
Select T.Name, T.Street, T.City, T.State, T.Zipcode,
   P.Name, P.Street, P.City, P.State, P.Zipcode
from  TA T, PROFESSOR P
where  T.advisor = P.id
```

The size of the join attribute id/advisor (i.e., S_{R_A}) is 4 bytes and the size of attributes Name, Street, City, State, and Zipcode of *PROFESSOR* relation are 20, 20, 10, 20 and 6 bytes, respectively. We varied the number of tuples per relation (the x-axis of the reported graphs) and measured the response time of the join query (in milliseconds) for each tuple distribution (the y-axis of the reported graphs). The x-axis is the percentage of the number of tuples of *TA* relation that resides at S_1 (i.e., $100 \times (N_\ell/N_\ell + N_r)$).

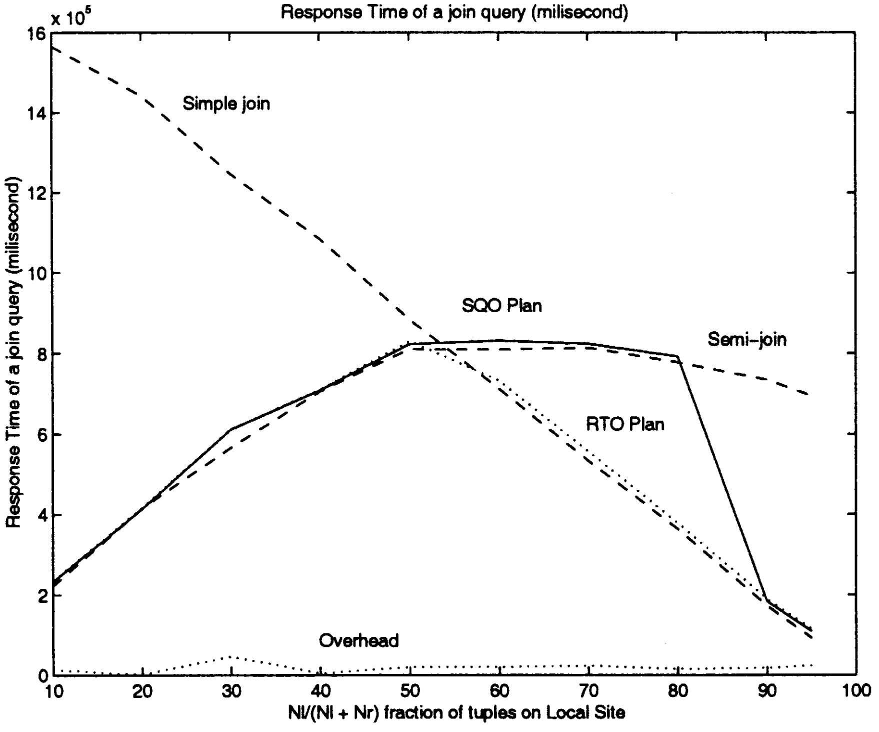

Fig. 2. Response time of a join query for different plans.

3.2. Results

For the first set of experiments, we compared the performance of SQO and RTO when the two servers are equally loaded. In this case, one expects to see a similar performance for SQO and RTO. However, as seen in Fig. 2, RTO (the dotted line) always chooses the correct plan by switching from semi-join to simple join plan at 50% tuple distribution. Instead, SQO (the solid line) wrongly continues preferring semi-join to simple join until 80% of tuple distribution. That is, when the difference between N_ℓ and N_r is significant, both optimizers can correctly determine the best plan. The decision becomes more challenging when N_ℓ and N_r have values with marginal differences. SQO prefers semi-join because it overestimates the communication cost of simple join due to equation (3). RTO, however, realizes that communication cost is not only affected by the amount of data shipped but also other factors and hence simple join, which ships more volume of data, might not be as bad as expected. In this situation, the cost of remote invocation (see Section 2.4) impacts semi-join more than simple join. By capturing the facts and amortizing the associated cost by incorporating $C_{\ell 2r}$ and $C_{r2\ell}$ into its equations, RTO detects the superiority of simple join to semi-join after 50% tuple distribution. Note that switching at the point of 50% tuple distribution cannot be generalized by a static optimizer because it is very much dependent on our experimental setup and the participating BUCKY relations. For this experiment, RTO outperformed SQO by an average of 32.5%. Meanwhile, RTO incurred an average of 6.4% extra delay as compared to the optimal plan

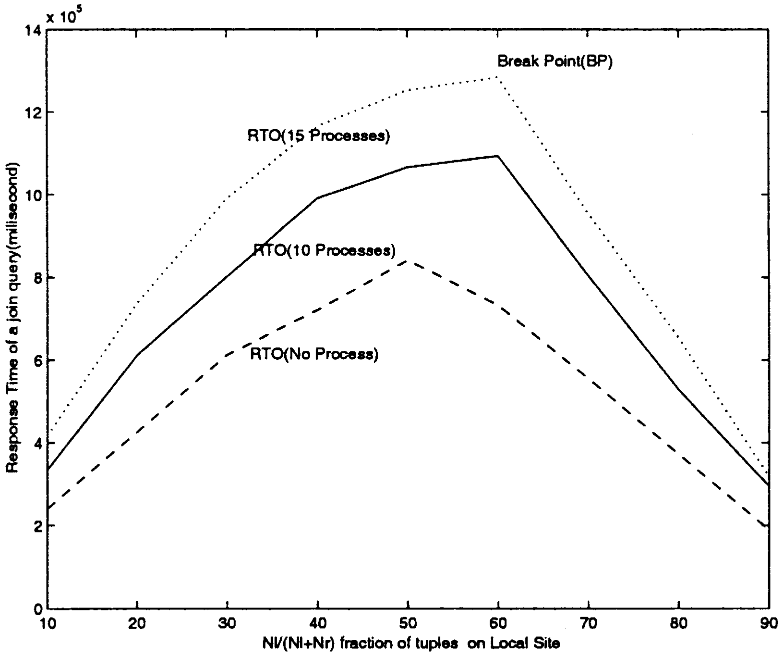

Fig. 3. Adaptation of RTO to workload changes.

due to the overhead of probe queries. This overhead is shown as a dotted line at the bottom of Fig. 2.

In the second set of experiments, we spawned some processes (performing I/Os in cycles) on the local server. We demonstrate the performance of different optimizers when 10 and 15 processes are activated on the local site. Recall that simple join performs one heavy join operation at the local site. Therefore, as the local site becomes more loaded, simple join becomes a less attractive plan. This behavior is illustrated in Fig. 3, where the switching point (the point at which simple join starts to outperform semi-join) is shifting to the right. Trivially, since SQO does not take the server workload into consideration, it always performs identically independent of the load. RTO, on the other hand, captures the server load and hence switches to the superior plan exactly at the switching points. In Fig. 3, observe how the query response time has been increased as we activate more processes on the local server.

Finally, to show that the impact of load on the remote server and local server is not symmetrical, we activated some processes on the remote server (see Fig. 4). The first impression is that since semi-join utilizes the remote server more than simple join, the switching point should shift to the left (the reverse behavior as compared to the previous set of experiments). That is, as one increases the load on the remote server, the simple join plan should outperform semi-join sooner. However, as illustrated in Fig. 4, this is not the case. The reason is that by overloading the remote server it will send data to the local server at a lower rate (this is due to the impact of $Delay_{send}(X)$ and $Delay_{receive}(X)$ factors in equation 8). Therefore, the simple join plan will suffer as well. The probe queries, by measuring

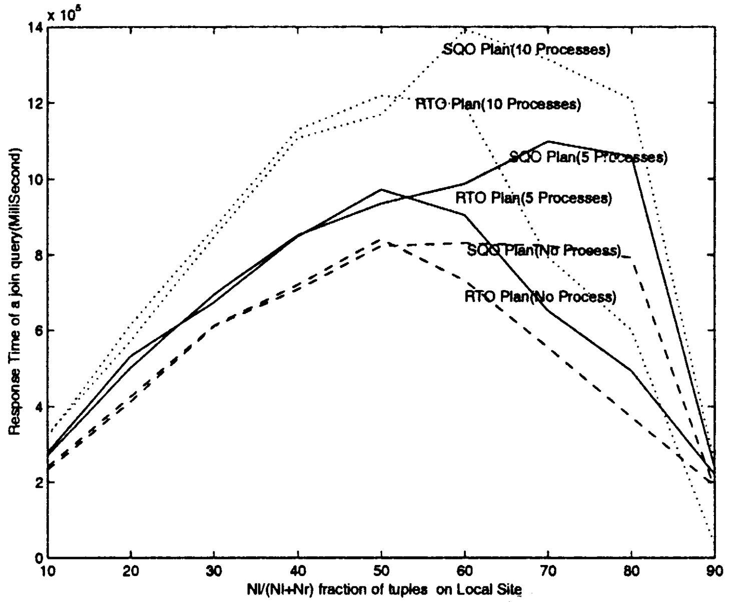

Fig. 4. Impact of load on the remote server, S_2.

$C_{\ell 2r}$ and $C_{r2\ell}$, automatically capture all these behaviors. Therefore, as depicted in Fig. 4, RTO can still choose the optimal plan.

4. Conclusions and Future Directions

By implementing a sample distributed database system consisting of heterogeneous servers running homogeneous DBMS and connecting them via the Internet, the importance and effectiveness of run-time optimizations have been demonstrated. Our run-time *join* optimizer (RTO) issues two probe queries striving to estimate the cost of semi-join and simple join plans. By measuring the performance of the probe queries and analyzing the results, RTO selects an optimal plan taking into account run-time behavior of the environment at the time of query execution. We demonstrated through analysis and experiments that our RTO can capture the communication delay, server workload, and other hidden costs specific to certain implementation (i.e., RMI cost in our case). This is achieved without making any assumptions or attempts to model the chaotic behavior of the Internet-based environment. As a byproduct, our RTO is less sensitive to statistical anomalies than SQO.

We intend to extend this work in three directions. First, we want to extend our experimental setup to multiple sites to extend RTO to support k-way joins. Second, we want to populate our object-relational database with multimedia data types in order to compare CPU-intensive plans with communication-intensive ones. We expect that here network congestion would have a high impact on preferring one plan to the other. Finally, we would like to implement an *adaptive* optimization technique (see Section 2.5) in order to capture sudden changes in run-time behavior.

Acknowledgements. This research was supported in part by gifts from Informix, Intel, NASA/JPL contract no. 961518 and NSF grants EEC-9529152 (IMSC ERC) and MRI-9724567.

References

Amsaleg L, Bonnet P, Franklin M, Tomasic A, Urhan T (1997) Improving responsiveness for wide-area data access. Data Engineeering 20(3): 3–11

Apers P, Hevner A, Ya SB (1983) Algorithms for distributed queries. IEEE Transactions on Software Engineering 9(3): 57–68

Bernstein P, Goodman N, Wong E, Reeve C, Rothnie J (1981) Query processing in a system for distributed databases. ACM Transactions on Database Systems 6(4): 602–625

Bodorik P, Riordo J, Pyra J (1992) Deciding to correct distributed query processing. IEEE Transactions on Knowledge and Data Engineering 4(3): ● ● ●–● ● ●

Carey M, DeWitt D, Naughton J, Asgarian M, Brown P, Gehrke J, Shah D (1997) The Bucky object-relational benchmark. In Proceedings of ACM SIGMOD, May

Ceri S, Pelagatti G (1984) Distributed databases: principles and systems. McGraw-Hill, New York, pp 141–156

Chen ALP, Li VOK (1984) Improvement algorithms for semijoin query processing programs in distributed database systems. ACM Transactions on Computers 33(11): 959–967

Chen M, Yu P (1992) Interleaving a join sequence with semi-joins in distributed query processing. IEEE Transactions of Parallel and Distributed Systems 3(5): 611–621

Evrendile C, Dogac A, Nural S, Ozcan F (1997) Multidatabase query optimization. Journal of Distributed and Parallel Databases 5(1): ● ● ●–● ● ●

Haas PJ, Swami AN (1995) Sampling-based selectivity estimation for joins using augmented frequent value statistics. In Proceedings of 11th IEEE international conference on data engineering, March

Informix (1997) Informix Universal Server: Informix guide to SQL. Syntax vols 1 and 2, version 9.1

Paxson V, Floyd S (1997) Why we don't know how to simulate the Internet. In Proceedings of the 1997 Winter Simulation Conference, September

Paxson V (1997) Measurements and analysis of end to end Internet dynamics. PhD thesis, University of California, Berkeley (http://www-nrg.ee.lbl.gov/nrg-papers.html), April

Silberschatz A, Korth H, Sudarshan S (1997) Database system concepts. McGraw-Hill, New York, pp 587–631

Yu C, Meng W (1998) Principles of database query processing for advanced application. Morgan Kaufman (1994), San Francisco, CA, pp 81–116

Zhu Q, Larson P-A (1994) A query sampling method for estimating local cost parameters in a multidatabase system. In Proceedings of 10th IEEE international conference on data engineering, February

Correspondence and offprint requests to: C. Shahabi, Department of Computer Science, University of Southern California, Los Angeles, CA 90089, USA. Email: shahabi@rcf.usc.edu