Run-Time Optimizations of Join Queries for Distributed Databases over the Internet *

Cyrus Shahabi, Latifur Khan, and Dennis McLeod
Integrated Media Systems Center &
Department of Computer Science
University of Southern California
Los Angeles, California 90089
[shahabi, latifurk, mcleod]@usc.edu

Vishal Shah†
C&C Research Labs,
NEC USA Inc.
110 Rio Robles,
San Jose, CA 95134
vishal@ccrl.sj.nec.com

Abstract

A new probe-based run-time 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 DBMS 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 run-time probing mechanism. An implementation of our run-time optimization technique for join queries was constructed in the Java language and incorporated into the experimental setup. The results demonstrate the superiority of run-time probe-based optimization over a static optimization.

1 Introduction

A distributed database is a collection of partially independent databases that share a common schema, and coordinates processing of non-local transactions. Processors communicate with one another through a communication network [SKS97, YM98]. We focus on distributed database systems with sites running homogeneous software (i.e., database management system, DBMS) on

*This research was supported in part by gifts from Informix, JPL/NASA and NSF grants EEC-9529152 (IMSC ERC) and MRI-9724567. †This work has been completed when the author was a graduate student at USC.

heterogeneous hardware (e.g., PC and Unix workstations) connected via the Internet. Distributed 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 highly depends on the amount of participation by each site as well as the amount of data shipment between the sites. Assuming a private/dedicated network and servers, this cost can be computed a priori due to the predictability of servers and network conditions. However, in the Internet environment which is based on a best effort service, there are a number of unpredictable factors that make the cost computation complicated. 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 due to these unpredictable factors. In the following paragraph, we explain some of these factors via simple examples.

Participating sites (or servers) of a distributed database system 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. For example, one site might only run a database server while the other is a database server, a web server, and an e-mail server. Moreover, the workload on each server might vary over time. A server running overnight backup processes is more loaded at night as compared to a server running 8a.m.-5p.m. office transactions. Due to time differences, a server in New York might receive more queries at 5a.m. in pacific standard time as compared to those received by a server in Los Angeles. The network traffic is another major factor. It is not easy to predict network delay in the Internet. 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.

Although we believe our probe-based run-time optimization technique is applicable to *multi-databases* with sites running heterogeneous DBMS, we do not consider such a complex environment

in order to focus on the query processing and optimization issues(see Sec. 5). There has been an extensive research in query processing and optimization in both distributed databases and multi-databases [ABF+97, AHY83, BGW+81, BRJ89, BRP92, CY92, EDNO97, KYY83, RK91]. Among those, only a few considered run-time parameters in their optimizers [ABF+97, BRP92, EDNO97]. We distinguish these studies from ours in Sec. 2. 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. In this paper, we demonstrate the importance and effectiveness of a probe-based run-time optimization technique for join queries in a distributed database system over the Internet. We focussed on *join* queries because join operation is not only frequently used but also expensive [YM98].

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 [Inf97]) connected via the Internet. We then split the BUCKY database (from the BUCKY benchmark [CDN+97]) 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 by 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 the average by 32.5% over a static optimizer while the probing overhead only results in an average of 6.4% increase in response time. We also discuss an enhanced version of our optimizer which reduces the overhead by an average of 45% (i.e., observing 3.5% increase in response time due to overhead) by utilizing the results of the probe query. Obviously, these numbers depend on the number of tuples sampled by the probe queries.

The remainder of this paper is organized as follows. Section 2 covers some related work on

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

query processing and optimization in both distributed databases and multidatabases. Section 3 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 4 consists of a performance study to compare the performance of our run-time optimization technique with that of a static optimizer. Finally, Section 5 concludes the paper and provides an overview on our future plans.

2 Related Work

There have been various studies on query processing and optimization in distributed, federated, and multidatabase systems [AHY83, BGW+81, BPR90, CY92, KYY83, RK91]. What distinguish us from these studies is our consideration of run-time environment in order to optimize the queries. There are, however, other studies considering run-time environment [ABF+97, AFTU96, BRJ89, BRP92, EDNO97, ONK+97, ML86]. Here we discuss those studies in more details and distinguish them from this study.

In [EDNO97, ONK+97], assuming a multidatabase system they realized the importance of run-time optimization (they used the term *dynamic* optimization) and proposed a weight function to capture the workload and transmission cost for each participating site. The objective is to choose the sites whose cost functions are less than a certain threshold in order to participate them in the query execution. They use a similar technique to our probing mechanism to capture the workload; however, the communication cost is calculated as a linear function of the size of the tuples transmitted. We show that the communication cost is not a linear function of the size (see Fig. 8). That is, the cost of transferring 4MBytes of data is not 1000 times higher than that of transferring 4 KBytes. Furthermore, we show that other factors such as server load and choice of implementation also impact the communication cost.

In [ABF+97, AFTU96], they also realized the inadequacy of static query optimization and proposed a *detective* technique to identify sites with delays higher than expected during query execution. Subsequently, they stop waiting for problematic sites and reschedule the plan for other sites. In this case, their technique might generate incomplete results if the problematic sites never recover. While their approach detect the problem and try to resolve it, our approach is *predictive* and try to avoid the problem all together. Although a predictive approach results in an initial overhead, we show that in some cases we can minimize the overhead by utilizing the results of our probing query. Furthermore, for expensive queries the overhead is marginal.

In [BRJ89, BRP92], various types of adaptive query execution techniques are discussed. The idea is to monitor the execution of a plan and if the performance is lower than what estimated, then the plan is corrected utilizing the newly captured information. Again, this is a detective technique trying to compensate for a wrong decision by re-planning. Finally, [BRJ89, BRP92] also assume communication costs are directly proportional to the volume of data transferred.

In [ML86], query optimizer estimates the total cost of a plan by summing up the CPU cost, I/O cost, message passing cost and communication cost. The last two costs are computed based on heuristics. Communication cost is estimated as the product of number of bytes transfered and effective bandwidth available between the two sites. Over the Internet, it is not trivial to obtain the effective bandwidth between two sites. Furthermore, effective bandwidth is changed frequently, due to the network dynamics and it is hard to maintain updated effective bandwidth information.

3 Run-Time Optimization (RTO) 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. Note that query optimization within a database site is beyond the scope of this paper and our techniques rely on each site for local query optimizations.

3.1 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. Utilization of semi-joins to reduce the size of the intermediate relations has received a great deal of attention [BGW+81, YM98]. 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

[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 [CP84, YM98].

to decide which plan to choose in order to minimize the response time of a certain join query. It is the responsibility of a query optimizer to assign a cost to each plan and then choose the cheaper plan.

3.2 Static Query Optimizer (SQO)

In this section, we explain a conventional method [BGW+81, CP84, AHY83] 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 distinct values for common attribute A 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 startup 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:

a. 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. The cost to transfer $\Pi_A(R_\ell)$ from S_1 to S_2 is:

$$C_0 + C_1 \times S_{R_A} \times N_\ell \tag{2}$$

b. Now $\Pi_A(R_\ell)$ is joined with R_r at S_2 with a zero cost $(R' = \Pi_A(R_\ell) \bowtie R_r)$

c. Suppose $\mid R' \mid$ is the cardinality of relation R', the cost of sending R' to S_1 is:

$$C_0 + C_1 \times S_{R_r} \times \mid R' \mid \tag{3}$$

d. Finally, R' is joined with R_ℓ at S_1 with a zero cost $(Res = R_\ell \bowtie R')$

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

$$Cost(P_{sj}) = 2 \times C_0 + C_1 \times (S_{R_A} \times N_\ell + S_{R_r} \times \mid R' \mid) \tag{4}$$

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

$$C_0 + C_1 \times (S_{R_A} \times N_\ell + S_{R_r} \times \mid R' \mid) \leq C_1 \times S_{R_r} \times N_r \tag{5}$$

SQO can examine the above inequality accurately only if it has all the required information (e.g., N_ℓ, S_{R_r}) a priori. In addition, it needs to estimate the size of intermediate results (i.e., $\mid R' \mid$). One estimation is as follows:

$$\mid R' \mid = domain(A) \times sel(R_\ell, A) \times sel(R_r, A) \tag{6}$$

where $sel(R_\ell, A)$ and $sel(R_r, A)$ are the selectivity of attribute A in relations R_ℓ and R_r, respectively. Eq. 6 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. Later in Sec. 3.4, we show that as a by product of our probing technique, we do not need to make any of these assumptions.

3.3 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 tuple transfer 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 Eq. 5 by the estimated costs. We assume that there would be no sudden changes in the behavior of run-time environment between the time that a probe query is submitted and the time that the original query will be submitted. In addition, RTO analyzes the results of the probe queries to estimate the size of R' more accurately. This last step of RTO is of course identical to the concept of *sampling*.

For the remainder of this section, we first describe the probe queries and how their measured performance values are incorporated into Eq. 5. Next, we propose an enhanced version of RTO to reduce the overhead of probing by utilizing its results to support the original query. Later, in Sec. 3.4 we argue how our modification to Eq. 5 can capture run-time behavior and estimate the size of intermediate relations more accurately.

3.3.1 Probe Queries

Our main objective is to modify the SQO main equation (Eq. 5) 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 Eq. 5 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 (see Appendix A.2). This probe sends the A attribute of X number of tuples of R_ℓ, denoted as 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 then is 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:

$$C_{\ell 2r} = \frac{S(X) + RIC_Q + RIC(X) + JC_r}{X} \tag{7}$$

In Eq. 7, $S(X)$ 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. Note that due to stateless nature of HTTP (which is the protocol used within our setup to access remote sites, see Sec. 4.1), RIC is a linear function of the number of inserted tuples to S_2(i.e., X). 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 size of R' can be estimated as:

$$Sample\_estimate(R') = \frac{X_j \times N_\ell}{X} \tag{8}$$

Probe Query B: The second probe query strives to replace the term $C_1 \times S_{R_r}$ of Eq. 5 with a more accurate estimation. It receives X number of tuples of R_r, denoted as 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:

$$C_{r2\ell} = \frac{RIC(X) + S(X) + JC_\ell}{X} \tag{9}$$

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

In Eq. 9, $S(X)$ 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. Again, RIC is a linear function of the number of requested tuples (i.e., X).

In Eqs. 7 and 9, $S(X)$ is capturing the following run-time parameters:

$$S(X) = Delay_{send}(X) + Delay_{network}(X) + Delay_{receive}(X) \tag{10}$$

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.

Now we can modify Eq. 5 of SQO as follows:

$$N_\ell \times C_{\ell 2r} + Sample\_estimate(R') \times C_{r2\ell} \leq N_r \times C_{r2\ell} \tag{11}$$

In Eq. 11, the terms $C_1 \times S_{R_A}$, and $C_1 \times S_{R_r}$ of Eq. 5 are replaced by $C_{\ell 2r}$ and $C_{r2\ell}$; and R' is computed using Eq. 8 instead of Eq. 6.

Selection of X tuples: Both probe queries transfer X tuples for their estimations. Therefore, the value of X (i.e., the number of tuples transferred) 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 only impacts the estimated size of R'. This *sampling* should be done in a way that X be a good representative of R_ℓ. This can be achieved by random selection of tuples from the relation R_ℓ. There are alternative techniques described in the literature for random selections of tuples from a relation such as heap scan [HHW97], index scan [HHW97] and an index sampling technique [Olk93]. There are many issues in obtaining a good random representative specially when there are index structures on the relation. The details of sampling are beyond the scope of this paper.

Scalability: Although we describe our probe queries for joins between two relations (i.e., 2-way join), the technique is indeed generalizable to k-way join. When joining k relations on a common attribute, the k-way join can be considered as $(k-1)$ 2-way joins. Hence, besides typical optimization challenges for 2-way joins, an additional challenge 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 [AHY83]. 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. As we showed in Appendix A.2, this assumption is not true for the Internet-based environment. Hence, 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 due 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. In our experiments, we utilized Java multi-threading primitives [Ree97] to perform probe queries concurrently. Therefore, the overhead observed for k-way join optimization is equal to the maximum delay incurred among all the probe queries. After determining the optimal execution order, we can now utilize probe queries A and B to choose either semi-join or simple join plans per each 2-way join. Alternatively, at this step we can only submit probe query A because the results for probe query B have already been collected. These results have been collected when the $k \times (k-1)$ probe queries were sent in order to determine the optimal execution order. Currently, we are investigating the extension of our probe-based technique to support k-way join within our experimental setup.

3.3.2 Enhanced RTO

One major problem with our RTO is the overhead associated with probing queries. This overhead can be alleviated by a simple enhancement. Recall that during the first step of probe query A, X tuples of R_ℓ each consisting of single common attribute A are transfered to S_2. The idea is to keep that relation R_{XA} at S_2 and do not discard it. Therefore, if P_{sj} is selected by RTO as the superior plan, it will not be required to send that X tuples to S_2 again. This results in saving both $S(X)$ and $RIC(X)$. We evaluated the impact of this enhancement in our performance evaluation and an average of 45% reduction in overhead has been observed for a given value of X.

3.4 Analysis and Comparison

In this section, we analyze why Eq. 11 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 (see Sec. 2) assumed communication cost is linearly proportional to the size of data transferred.

The same assumption has also been made by the static query optimization technique discussed in this paper (see Sec. 3.2). However, researchers [Jac97] in network community demonstrate that over the Internet, the communication cost does not increase as a linear function of the size of data transfered. This can also be observed in our experiments as reported in Appendix A.2 (see Fig. 8). In our experiments, however, we observed that the communication cost is indeed a linear function of the **number** of tuples transfered (see Fig. 7 of Appendix A.2). This is because the granularity of data transfer in our experiments was in tuples. These observations explain why SQO which assumes communication cost is a linear function of S_{R_A} and S_{R_r} (see Eq. 5) fails. Instead, 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. 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}. Note that the same argument holds if the granularity of data transfer is in blocks instead of tuples. However, the probe queries must be modified to extrapolate on the number of block movement as opposed to tuple movement.

Remote Invocation Cost: As discussed in Sec. 4.1, in our experimental setup Remote Method Invocation (RMI) was employed in order to access a remote server. An interesting distinction between simple join and semi-join plan is that in general semi-join plan uses remote invocation more often as compared to that of simple join plan. To illustrate, P_{sj} utilizes remote invocation N_ℓ times to insert tuples into S_2, one time to execute join remotely at S_2, and R' times to fetch the semi-join result 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 (see Sec. 4) by $C_{r2\ell}$ and $C_{\ell 2r}$. Therefore, a general conclusion is that our run-time probing mechanism can capture any surprises associated with specific implementations (e.g., RMI in our case) which can never be accounted for by the static optimizer. Note that other alternative implementations will also observe some overheads similar to the RMI overhead. For example, if Java Database Connectivity (JDBC) is employed to connect to the database servers, remote sites can be accessed in three alternative ways depending on the JDBC driver implementation [Ree97]: 1) distributed objects implemented in RMI, 2) message-passing technique, or 3) Common Object Request Broker Adapter (CORBA) [Far98]. Trivially, all three methods introduce some overheads when accessing remote sites.

Load Cost: From Eq. 5, 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. However, in Appendix A.1 we show the important impact of the load in choosing the best plan. On the other hand, with RTO, it is trivial from Eqs. 7, 9, and 10 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 remote site and the other at local, while simple join only performs one heavy but local join operation. Beside these operations that are highly dependent on the server workload, there exists other dependencies. 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. SQO makes two major assumptions in order to estimate the size of R' in Eq. 6: 1) domain of A is discrete and can be considered as A's sample space, and 2) there is no correlation between R_ℓ and R_r. Instead, RTO estimates the size of R' by sampling (see Eq. 8) and thus is independent of both of these assumptions. That is, 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 Eq. 8) accordingly. Therefore, a positive correlation results in higher value of X_j and vice-versa.

4 Performance Evaluation

As we argued in Sec. 1, the run-time behavior is too unpredictable and sophisticated to be captured and analyzed by analytical models or simulations. Hence, we decided to implement a real experimental setup. 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.

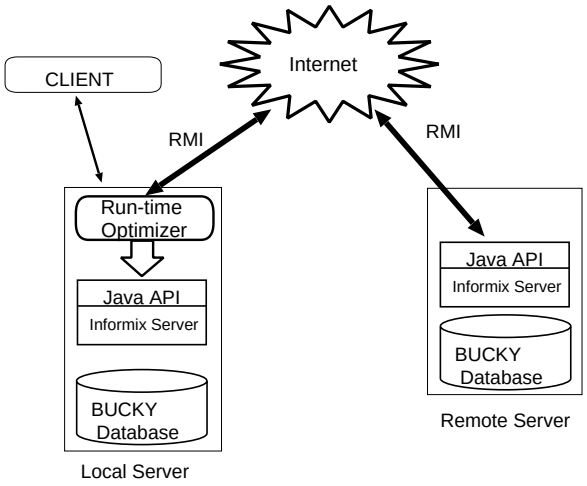


Figure 1: Experimental Setup.

Our experiments verified that RTO can adopt itself to workload changes and always chooses the best plan while SQO's decision is static and always a specific plan is 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. We did not report our experiments for variable network load because both of our join plans utilize network almost identically and hence a congested (or free) network will not result in preferring one plan to the other. We plan to do more experiments with other sorts of queries that give rise to plans utilizing network differently. Finally, we did some experiments to investigate the overhead associated with probe queries and quantify the reduction in overhead by employing our enhanced version of RTO.

4.1 Experimental Setup

Fig. 1 depicts our experimental setup which consists of two sites S_1 and S_2. The sites are Unix-boxes with an identical hardware platform (a SUN Sparc Ultra 2 model with 188 MBytes 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 emaluating 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.

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, Remote Method Invocation (RMI) is used to open a connection to the database server residing on the other site. The *Credential* class of RMI has a public constructor that specifies enough information to open a connection to a database server. Two types of Credentials are used: 1) *Direct Credentials* for local applications, and *Remote Credentials* to access the remote database server using typical HTTP credentials. The BUCKY database, from the BUCKY benchmark [CDN+97], 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 and its HTTP credentials 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. For example, in a real-world university application, the information on faculty is kept at a site in the human resources (S_2) while the TA information is kept at (say) computer science department site (S_1). 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. When the query is submitted through an interface (a Java applet running at S_1), the query optimizer consults the metadata to identify the location of the *TA* and the *PROFESSOR* relations. RTO will then decide using *probe queries A & B* which plan to choose. 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., $100 \times \frac{N_\ell}{N_\ell + N_r}$). For comparison purposes, we also measured the response time of SQO, *semi-join* and *simple join* for each experiment.

4.2 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 expect 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 Eq. 5. RTO, however, realizes that communication cost is not a linear function of amount of data shipped (as shown in Fig. 8) 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 Sec. 3.4) impacts semi-join more than simple join. By capturing these two facts and amortizing their 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. This is exactly why a run-time optimizer is required.

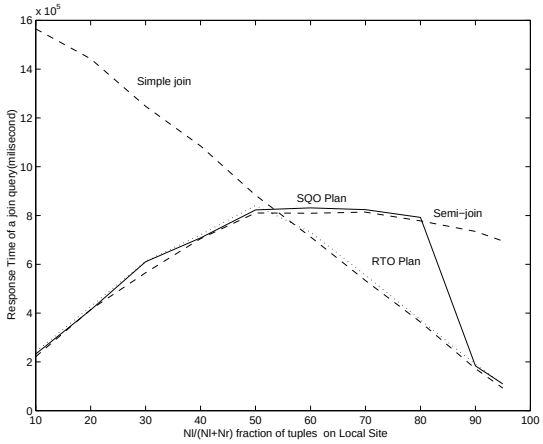
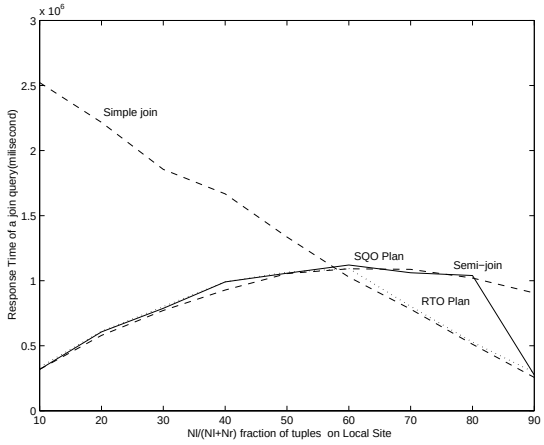
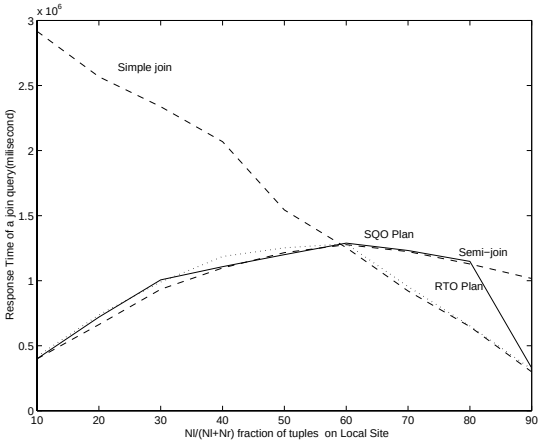


Figure 2: Response time of a join query for different plans.

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 due to the overhead of probe queries. We further reduced this marginal overhead of RTO, by employing our enhanced RTO (see Sec. 3.3.2). As expected, when the optimal plan is simple join, the overhead cannot be avoided and both of RTOs behaved almost identically. However an average reduction of 45% in overhead was observed for the cases where semi-join was the optimal plan.



a.10 Processes running on local site



b. 15 Processes running on local site

Figure 3: Impact of load on the local server, S_1

In the second set of experiments, we spawned some processes (performing I/O's in cycles) on the local server. Fig. 3(a) and 3(b) demonstrate the performance of different optimizers when 10 and 15 processes are activated on the local site, respectively. 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(a) and 3(b) where the switching point (the point that simple join starts to outperform semi-join) is shifting to the right (also see Fig. 4). 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 (see Fig. 4). In Fig. 4, observe how the query response time has been increased as we activate more processes on the local server. Note that the variable load on servers can also be interpreted as if the local server is a low-end system as compared to a high-end remote server. Therefore, RTO can also capture the heterogeneity of servers.

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. 5). The first impression is that since semi-join utilizes the remote server more than simple join, hence the switching point should shift to the left (the reverse behavior as compared to 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. 5, 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 Eq. 10). Therefore, the simple-join plan will suffer as well. The beauty of

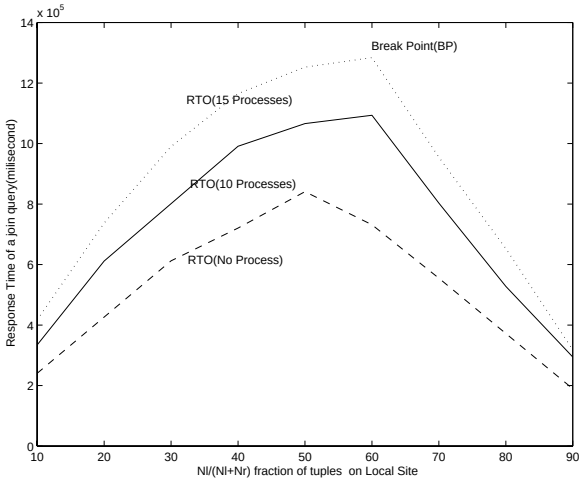


Figure 4: Adoption of RTO to workload changes

our technique is that RTO does not need to take all these arguments into consideration in order to decide which plan to choose. The probe queries by measuring $C_{\ell 2r}$ and $C_{r2\ell}$, automatically capture all these behaviors. Therefore, as depicted in Fig. 5, RTO can still choose the optimal plan.

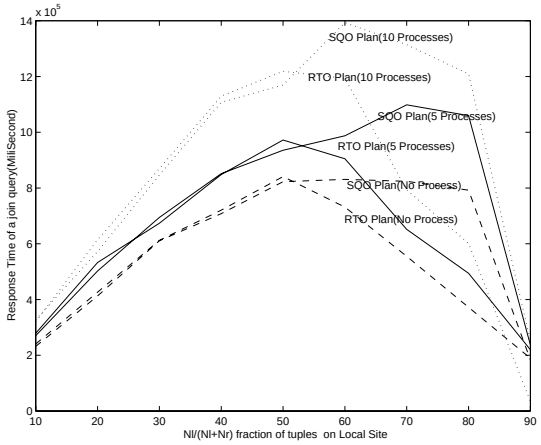


Figure 5: Impact of load on the remote server, S_2.

5 Conclusions and Future Directions

A new probe-based run-time optimization technique is developed and demonstrated in the context of an Internet-based distributed database environment. By implementing a sample distributed database system consisting of heterogeneous servers running homogeneous DBMS and connecting them via 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. Furthermore, RTO relies less on the remote relation profiles than SQO since most of these information are captured during the probing process as a byproduct. Therefore, it becomes a better candidate for query optimization in multidatabase systems where the profiles resident on one site is not readily accessible to other sites.

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 want to run other DBMS softwares (e.g., DB2 and Oracle 8) on some of the sites in order to study our optimizer in a multidatabase environment.

References

[ABF+97] L. Amsaleg, P. Bonnet, M. Franklin, A. Tomasic, and T. Urha. Improving Responsiveness for Wide-area Data Access. *Data Engineeering*, 20(3):3–11, September 1997.

[AFTU96] L. Amsaleg, M. Franklin, A. Tomasic, and T. Urha. Scrambling Query Plans to Cope with Unexpected Delays. In *Proc. of the Int. Conf. on Parallel and Distributed Information Systems(PDIS), Miami Beach, Florida*, December 1996.

[AHY83] P. Apers, A. Hevner, and S. B. Ya. Algorithms for Distributed Querie. *IEEE Transactions on Software Engineering*, 9(3):57–68, January 1983.

[BGW+81] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie. Query Processing in a System for Distributed Databases. *ACM Transactions on Database Systems(TDS)*, 6(4):602–625, December 1981.

[BPR90] P. Bodorik, J. Pyra, and J. Riordo. Correcting Execution of Distributed Queries. In *Proceedings Second Intl. Symp. On Database in Paralle and Distributed System*, July 1990.

[BRJ89] P. Bodorik, J. Riordo, and C. Jacob. Dynamic Distributed Query Processing techniques. In *Proc ACM CSC'89 Conference*, February 1989.

[BRP92] P. Bodorik, J. Riordo, and J. Pyra. Deciding to Correct Distributed Query Processing. *IEEE Transactions on Knowledege and Data Engineering*, 4(3), June 1992.

[CDN+97] Michael Carey, David DeWitt, Jeffrey Naughton, Mohammad Asgarian, Paul Brown, Johannes Gehrke, and Dhaval Shah. The Bucky Object-Relational Benchmark. In *Proc. ACM SIGMOD*, May 1997.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases Principles and Systems*, pages 141–156. McGraw-Hil, 1984.

[CY92] M. Chen and P. Yu. Interleaving a Join Sequence with semi-joins in Distributed Query Processing. *IEEE Transactions of Parallel and Distributed Systems*, 3(5):611–621, September 1992.

[EDNO97] C. Evrendile, A. Dogac, S. Nural, and F. Ozcan. Multidatabase Query Optimization. *Journal of Distributed and Parallel Databases*, 5(1), January 1997.

[Far98] Jim Farley. *Java Distributed Computing*, pages 189–225. O'REILLY, 1998.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *Proc. ACM SIGMOD*, May 1997.

[Inf97] Informix. *Informix Universal Server: Informix guide to SQL: Syntax volume 1 & 2 version 9.1*, 1997.

[Jac97] Van Jacobson. pathchar - a tool to infer characteristics of Internet paths. `ftp://ftp.ee.lbl.gov/pathchar/`, April 1997.

[KYY83] Y. Kambayaashi, M. Yoshikawa, and S. Yajima. Query Processing for Distributed Databases using Generalized Semi-join. In *ACM SIGMOD International Conference on Management of Data, San Jose, CA*, May 1983.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. 12th Intl. Conference on Very Large Data BasesVLDB*, August 1986.

[Olk93] F. Olken. Random Sampling from Databases. PhD thesis, University of California, Berkeley, 1993.

[ONK+97] F. Ozcan, Sena Nural, Pinar Koskal, C. Evrendile, and Asuman Dogac. Dynamic Query Optimization in Multidatabases. *Data Engineeering*, 20(3):38–45, September 1997.

[Ree97] George Reese. *Database Programming with JDBC and JAVA*, pages 41–55. O'REILLY, 1997.

[RK91] N. Roussopoulos and H. Kang. A Pipe n-way join algorithm based on the 2-way semi-join program. *IEEE Transactions on Knowledege and Data Engineering*, 3(4):486–495, 1991.

[SKS97] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*, chapter 18, pages 587–631. McGraw-Hill, 1997.

[THO96] J. Touch, J. Heidemann, and K. Obraczka. Analysis of HTTP Performance. Released as web page `http://www.isi.edu/lsam/ib/http-perf/`, June 1996.

[YM98] Clement Yu and Weiyi Meng. *Principles of Database Query Processing for Advanced Application*, chapter 3, pages 81–116. Morgan Kaufman, 1998.

A Appendix

In this appendix, we report how the server workload impacts the query response time of different query plans and how the communication cost varies with the number of tuples and the size of each tuple. Our experimental setup is identical to that of Sec. 4.1.

A.1 Impact of Server Workload

To observe the impact of server workload, the $Staff(id, name, street, city, state, zipcode, birthDate, picture, latitude, longitude, dept, Dathired, status, annualSalary)$ relation of BUCKY was horizontally partitioned across sites S_1 and S_2. We varied the number of tuples residing on each site. The query is: *Find all Staff who make more than* \$96000 *per year.*
In SQL:

Select e.name, e.street, e.city, e.state, e.zipcode
from Staff e
where e.annualSalary $\geq$ 96000

To serve this query, first $\sigma_{annualSalary \geq 96000}$ and project operation are executed at both sites. Subsequently, the results obtained at one site are shipped to the other in order to perform a union operation. Depending on the site that performs the final union operation two plans can be generated: P_A if union is performed at site S_1, and P_B if union is performed at site S_2. For now, we ignore the final shipment of the result to the site that serves the original query. Intuitively, the dominant cost in this query execution is the communication cost. Hence, a static query optimizer might select the plan which ships the least amount of data (i.e., number of tuples, since all tuples have identical size) from one site to another. For example, if 70% of tuples are at site S_1, then P_A will be chosen. However, considering the workload on each server, the above decision might be wrong. To illustrate, consider Fig. 6.

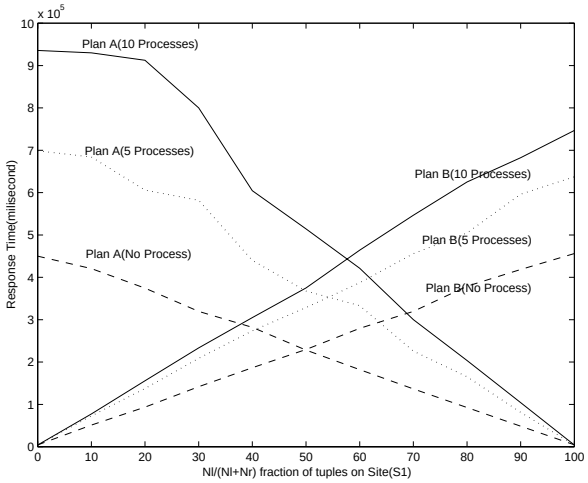


Figure 6: Impact of workload on a select query.

In Fig. 6, the x-axis is the percentage of the number of tuples in Staff relation that resides at S_1. The y-axis is the query response time in milliseconds. The dashed lines represent the performance of the two plans P_A and P_B when there is no extra workload on the servers. As expected when 50% of tuples are in each side (perfect split), the two plans perform identically. Hence, when we spawn 5 and 10 auxiliary processes on S_1 (dotted lines and solid lines, respectively), then P_B becomes the superior plan at the point of perfect split. This is because P_B performs the union operation at site S_2 which is now less loaded as compared to S_1. When running 10 auxiliary processes and at the point of perfect split, choosing P_A versus P_B will result in a 29% increase in the observed response time.

A.2 Impact of Communication Cost

In this section, we describe how the number of tuples and size of tuple impact the communication cost. To demonstrate the impact of number of tuples on communication delay, in Fig. 7, we varied

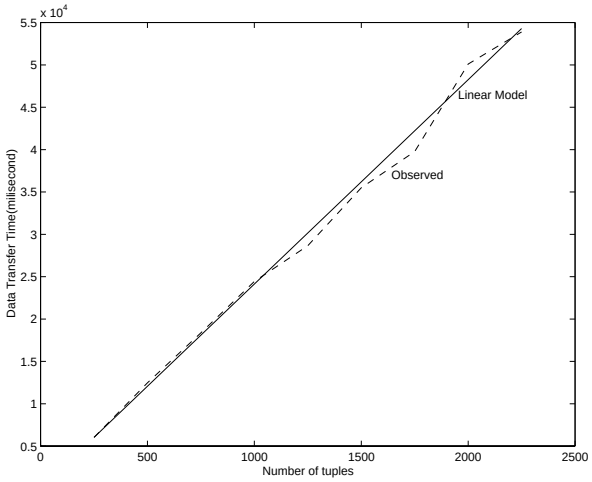


Figure 7: Communication cost for tuple transfer over the Internet

number of tuples from 500 to 2500 (X-axis) and measured the data shipment cost in millisecond (Y axis). Here, the solid line represents data shipment cost as a linear function of number of tuples shipment and the dotted line represents the observed cost during the experiment. As depicted in Fig. 7 the dotted line follows the solid line, hence communication cost is linearly proportional to the number of tuples movement for a fixed size tuples.

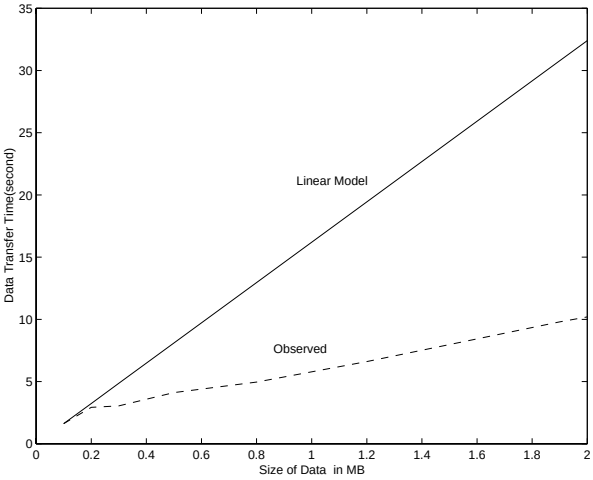


Figure 8: Communication cost for data transfer over the Internet.

Here, we demonstrate over the Internet, communication cost increases non linearly as the size of data transfer. We conducted a simple experiment transferring different amount of data using FTP. Although we employ HTTP in our experimental setup, we argue that for this experiment the choice of protocol is irrelevant. This is because both protocols use a reliable connection-oriented transport protocol, TCP [THO96]. Both HTTP and FTP use a separate TCP connection per transaction. Their only difference is that HTTP does not require to maintain a control channel (i.e. it is stateless). Instead, FTP opens a control channel once when the client is connected to the server. Subsequently, for every file transfer from that server a new TCP connection is required (similar to the case of HTTP) [THO96]. In order to compensate for this difference and eliminate the overhead of control channel maintenance for FTP, we ignored the time to transfer the first file for each connection and only report the required time to transfer remaining files. In Fig. 8, we varied the amount of data from 0.1MB to 2.0 MB (X-axis) and measured the transfer time in second (Y-axis). We measured the time to transfer the same amount of data from one site to another five times, and reported the average time for each experiment (depicted as dotted line in Fig. 8). For

the purpose of comparison, the solid line represents transfer time as a linear function of amount of data shipped. As depicted in Fig. 8, the communication cost is not a linear function of the amount of data transfered.

Finally, anybody browsing the Web has already experienced the impact of network congestion on communication cost. That is, during different time of the day the time required to download the same web-page varies significantly. A static optimizer that assumes the communication cost is a static function of the volume of data shipped, will select the same plan under these two conditions which might result in choosing an expensive plan.