Load and Network Aware Query Routing for Information Integration

Wen-Syan Li  Vishal S. Batra  Vijayshankar Raman  Wei Han  K. Selçuk Candan  Inderpal Narang

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120 USA
Email: wen@almaden.ibm.com

Abstract

Current federated systems deploy cost-based query optimization mechanisms; i.e., the optimizer selects a global query plan with the lowest cost to execute. Thus, cost functions influence what remote sources (i.e. equivalent data sources) to access and how federated queries are processed. In most federated systems, the underlying cost model is based on database statistics and query statements; however, the system load of remote sources and the dynamic nature of the network latency in wide area networks are not considered. As a result, federated query processing solutions can not adapt to runtime environment changes, such as network congestion or heavy workloads at remote sources. We present a novel system architecture that deploys a Query Cost Calibrator to calibrate the cost function based on system load and network latency at the remote sources and consequently indirectly “influences” query routing and load distribution in federated information systems.

1 Introduction

A wide variety of applications require access to multiple heterogeneous, distributed data sources. By transparently integrating such diverse data sources, underlying differences in DBMSs, languages, and data models can be hidden and users can use a single data model and a single high-level query language to access the unified data through a global schema.

To address the needs of such federated information systems, IBM has developed the DB2 Information Integrator (II) [10] which provides relational access to both relational DBMSs and non-relational sources, such as file systems and web services. These data sources are registered at II as nicknames and thereafter can be accessed via wrappers. Statistics about the remote data sources are collected and maintained at II for later use by the II query optimizer for costing query plans. Figure 1 shows an architectural overview of the federated information system design using DB2 II. The operational flows on the system have two phases, a compile time phase and a runtime phase:

Compile time phase:

1. A user query submitted to the II is intercepted by the Query Patroller. The user query statement and query submission time are recorded. The query is then forwarded to II for further processing.

2. II looks up the nickname (i.e. local names of the remote database tables) definitions in the user query and breaks (i.e. rewrites) the query into multiple sub-queries and forwards these sub-queries to the appropriate wrappers according to their types.

3. For those sub-queries to be forwarded to a relational wrapper, the wrapper will return the query fragments that can be executed at each remote server and their estimated costs. For those sub-queries that are forwarded to a file wrapper, file paths are returned to II without estimated cost. For the cost estimation purpose, the wrapper may contact remote servers to obtain possible supported execution plans and their estimated costs.

Runtime phase:

1. After II receives all query fragments that can be executed at the remote sources and their estimated costs, the II query optimizer performs global query optimization. The query fragments selected by the query optimizer and their estimated costs as well as the estimated execution cost of the global query plan are stored in the explain table. Other information stored includes the execution descriptors of the selected query fragments that are needed to executed the query fragments at the remote servers.

2. II then forwards the query fragments selected by the global query plan to the wrapper for the remote sources for execution.

3. Query fragments are executed at the remote servers and results are returned to II through the wrappers. The results are merged by II and then sent back to the user.
4. After the query execution is completed, Query Patroller records the query completion time in the log for future use.

1.1 Problem Statement

DB2 Information Integrator (II) deploys cost-based query optimization to select a low cost global query plan to execute. Thus, cost functions used by II heavily influence what remote servers to be accessed and how federated queries are processed. Cost estimation is usually based on database statistics, query statements, and the local and remote system configuration (including the CPU power and I/O device characteristics such as seek time and transmission rates). In addition, DB2 allows the system administrator to specify expected network latencies between II and the remote servers. Cost functions have significant impacts on the choice of remote servers (i.e. equivalent data sources); however, existing cost functions do not consider

- the loads on the remote servers;
- dynamic nature of network latency between remote servers and II; and
- the availability characteristics of the remote sources.

As a result, federated systems can not dynamically adapt to runtime environment changes, such as network congestions or load spikes at the remote sources. Furthermore, since the query plans are generated via cost-based decision making process, currently, there are no mechanisms to avoid fast but unreliable sources, when alternatives are available. In addition, the II query optimizer optimizes user queries individually rather treating a workload as a whole. In some scenarios, selecting a low cost global query plan and applying this plan to all similar queries is not necessarily ideal when the workload needs to be distributed among alternative servers for better overall system performance via load balancing.

In this paper, we introduce a novel system architecture that deploys a query cost calibrator (QCC) to calibrate the cost function based on system availability, process and network latencies at the remote sources, and the system load at the information integration (II). QCC transparently adapts the cost functions to the runtime properties of the environment, and indirectly influences the federated query optimizer. This enables the selection of the right remote sources or replicas (i.e equivalent data sources) which yield the fastest overall query response time for a workload. QCC also identifies alternative query plans and recommends load balancing strategies to improve overall system performance. We have implemented the II middleware with QCC and evaluations show that it consistently outperforms a prototype version of DB2 Information Integrator.

1.2 Paper Organization

The rest of paper is organized as follows. In Section 2, we describe the architectural design of the current DB2 Information Integrator featuring adaptive query routing and load balance functionalities. In Section 3, we introduce the novel Query Cost Calibrator (QCC) component and its key features. In Section 4, we describe the load distribution scheme used in our system for load balance. In Section 5, we present the experimental evaluation of the proposed solution to demonstrate the benefits of the QCC in improving response time. In Section 6, we discuss the related work and compare with the work we present here. In Section 7, we give our concluding remarks.

2 Proposed System Architecture

In this section, we describe how our proposed system architecture (shown in Figure 2) builds on top of this architecture and enhances it transparently with two complementary components: (1) a meta-wrapper (MW) and (2) a query cost calibrator (QCC).

The meta-wrapper serves as a middleware between the information integrator and wrappers. At the compile time, MW receives queries from II and records (a) the incoming federated query statements, (b) the estimated cost of the federated queries, (c) the outgoing query fragments, and (d) their mappings to the remote servers. Furthermore, since the query plans are generated via cost-based decision making process, currently, there are no mechanisms to avoid fast but unreliable sources, when alternatives are available. In addition, the II query optimizer optimizes user queries individually rather treating a workload as a whole. In some scenarios, selecting a low cost global query plan and applying this plan to all similar queries is not necessarily ideal.
so that various system characteristics, such as remote system loads and network latencies, are implicitly taken into consideration while estimating query fragment processing costs. In addition to such transparent statistics collection, QCC also uses daemon programs that periodically access remote sources, through MW, to ensure their availability. The daemon programs are also used to derive initial query cost calibration factors by exploring the network latency and processing latency at remote sources.

When wrappers do not provide cost estimation and accessing remote servers to get cost statistics is not feasible, QCC deploys a simulated federated system that has the same II, meta-wrapper, and wrappers as the original run time system as well as the simulated catalog and virtual tables, to capture database statistics and server characteristics without storing the actual data. The simulated federated system allows QCC to derive alternative query plans and perform “what-if” analysis for query routing and data placement.

After query compilation at II, only the global query plan with the lowest cost is stored in the explain table. When queries are unique, this approach of choosing low cost plans is suitable. However, if there is a large number of similar queries that use the same plan, then the remote servers involved in this plan can get overloaded, rendering the original statistics invalid. To prevent such hot-spots and achieve proper load distribution, through the calibration and query routing of QCC, II is enabled to use alternative (maybe not the lowest cost, but close) global query plans in addition to the lowest-cost query plan.

In addition to collecting and using cost statistics, QCC also records error messages (if any) from accessing remote servers for assessing their availability and reliability. This information is later used to compute the reliability factor for cost calibration. Consequently, QCC influences II to access not only high performance but also highly available remote servers.

3 Query Cost Calibrator

In this section, we describe the design and functions of the Query Cost Calibrator. The parameters associated with cost functions in II include first tuple cost, next tuple cost, and cardinality, and total cost (i.e. first tuple cost + next tuple cost × cardinality). QCC calibrates first tuple cost, next tuple cost, and total cost. These costs and cardinality are returned to II and are then converted to the parameters that the optimizer uses internally for the cost functions. The cost referred in this paper is total cost for simplifying the presentation.

3.1 Query Cost Calibration for System Load and Network Latency

Figure 3 illustrates processing steps of a federated query, Q1, integrating data from two sources, S1 and S2 (underlined numbers in the figure indicate sequence numbers of the processing steps). II accesses the individual sources with query fragments and merges the results locally. During the compile time, Q1 is transformed into two query fragments, QF1 and QF2, for S1 and S2 respectively, and both query fragments are forwarded to the meta-wrapper (MW). MW forwards QF1 and QF2 to the corresponding wrappers for cost estimation.

The wrappers compute and return possible execution plans for these query fragments along with their estimated costs. In this example, two plans (QF1,p1 and QF1,p2) are returned for QF1 and two plans (QF2,p1 and QF2,p2)
are returned for $QF_2$. In addition to forwarding this information to II, MW passes these plans and the associated costs to QCCs (shown at the right hand side of Figure 3), which record it for future use.

The II query optimizer then selects a set of query fragments in the global query plan. Say, $QF_1p_1$ and $QF_2p_2$ are selected. At the compile time phase, neither MW nor QCC knows which query fragment plans are selected by the II query optimizer. After the selection of the query execution plan by the II optimizer, the run time phase of query processing starts (Figure 4). During the execution, the selected fragment plans, $QF_1p_1$ and $QF_2p_2$, are sent to MW, which then forwards the corresponding execution descriptors to appropriate remote servers through wrappers. After the execution of the query fragments are completed, once again, MW not only passes the results back to II, but also keeps track of the response times of the individual query fragments (in this example, 8 and 7 for $QF_1p_1$ and $QF_2p_2$ respectively) and passes this information to QCC which stores it for future use.

At this point, QCC has the record of two sets of costs for each query fragment (if selected by the II query optimizer): the estimated cost (obtained in the compile phase) and the observed cost (response time obtained in the runtime phase). Assuming that the original cost estimates are valid, any significant difference between these two sets of values has to be caused by variations in the network latencies or processing cost variations at the remote sources due to their local loads.

These external (and dynamic) factors are not explicitly known to II and are not included in the cost model. However, their combined effects can be captured using a single query fragment processing cost calibration factor per data source (and query fragment if runtime statistics is available), defined as the ratio of the average runtime cost vs. the average estimated cost. This allows II query optimizer to consider network and process latencies at the remote servers without having to observe these factors explicitly. Once the calibration factors are calculated, they can be used for calibrating estimated costs for future, yet-unseen query fragments. For this purpose, QCC computes per source query fragment processing cost calibration factors. In our simplified running example, the calibration factors for $S_1$ and $S_2$ can be calculated as 1.6 (i.e. $8/5$) and 1.4 (i.e. $7/5$) respectively.

Using these server cost calibration factors, II can calibrate other query fragments which have no runtime cost records. This process is depicted in Figure 5. Here a new federated query, $Q_5$, is issued to II. II transforms the query into two query fragments, $QF_1$ (seen before) and $QF_3$ (not seen before). At the compile phase, for $QF_3$ MW consults the wrapper to get a query plan, $QF_3p_1$ and its estimated cost, 8. However, instead of returning this estimated cost directly, MW calibrates the cost to 11.2 by multiplying the estimated cost, 8, by the per server query fragment processing cost calibration factor, 1.4. On the other hand, since we already have a plan and an estimated cost for $QF_1$, MW can compute the calibrated runtime cost without having to consult the wrapper as MW performs for $QF_1$ in this example.

### 3.2 Cost Calibration for Information Integration

II query optimizer uses a cost model, which includes the size of the query results at the remote sources as well as the additional cost of merging and aggregating the results locally, to estimate the cost of global query plans. In this process, II considers various database statistics and the physical characteristics of the server where II is located. However, the standard II cost model does not consider the impact of system load on II in local integration. To improve the quality of the estimates for the response times seen by end users or applications, we further calibrate the cost observed at the II level, using a workload cost calibration factor. For this purpose, we use the calibrated cost of the sources instead of the estimated cost, since the calibrated cost is closer to the runtime response time. In Figure 6, we illustrate this process with a simplified example in which we use execution history at II to calculate the work load cost calibration factor. Note that the table maintained in QCC for II query cost calibration factors is different from the table maintained for query fragment processing cost calibration factors.

### 3.3 Consideration of Remote Source Availability

QCC uses daemon programs to periodically access remote sources to verify their availability. In addition, from the query execution log provided by the query patroller and MW, QCC can detect the system down event at remote sources. When a system down event is detected, QCC temporally adjusts cost functions for these unavailable servers to infinity so that no query fragments will be forwarded to these remote sources. Note that the runtime log provided by the query patroller and MW enables QCC to influence II not to route queries to the unavailable remote sources. On the other hand, the status reports from the daemon programs allow QCC to make unavailable remote sources be considered by II again once the remote resources become available.

### 3.4 Dynamic Adjustment of Calibration Cycle

Since each remote server has different network and processing latencies, the query fragment processing cost calibration factors are calculated per remote server. Furthermore, dynamic nature of the network and processing latencies at each remote server can vary dramatically. Thus, the frequency of re-calibration does have impact to effectiveness of QCC in influencing II query optimization.
Figure 3. Interaction between QCC and Meta-wrapper (Compile Time)

Figure 4. Interaction between QCC and Meta-wrapper (Runtime)

Figure 5. Interaction between QCC and Meta-wrapper with Calibrated Cost (Compile Time)
QCC maintains aggregated histories of the various dynamic values associated with the remote source access costs to compute and maintain running averages. The information enables QCC to dynamically adjust calibration cycles for the reliability factor, query fragment processing cost calibration factor, II processing cost calibration factor, and simulated catalog refreshes.

### 3.5 Adaptive Query Routing

Above, we have described the novel functions of the query cost calibrator. By monitoring the runtime environment and query execution status, QCC dynamically calibrates the cost functions of the II query optimizer and thus influences the selection of global query execution plans. The introduction of QCC to II query optimization allows adaptive query routing based on current runtime environment instead of fixed query routing plans that are predetermined when the mapping between nicknames and remote sources and network latency values are defined. QCC can detect unavailability of the remote sources and adjust the cost functions of the remote sources so that II would not consider routing queries to these unavailable sources. QCC also incorporates reliability into the decision process of remote source selection.

Note that QCC influences the II optimizer to select the global query execution plans that have the lowest cost according to current runtime environment instead of introducing a new component or requiring the modification of the code at II optimizer. This transparent design gives QCC great flexibility in customizing cost functions for different business applications that may demand incorporation of unique business logic, such as QoS goal and reliability, outside of DB2 and II without modification of the DBMS.

In the next section, we will describe more advanced query routing schemes for load distribution.

### 4 Load Distribution

The query optimizer optimizes incoming queries individually, by selecting the cheapest global query plan, rather treating the workload as a whole. Selecting a low cost global query plan and applying this plan to all similar queries is not necessarily ideal since it tends to overload a small group of servers. In some scenarios, the workload needs to be distributed in a balanced way among alternative servers for better overall system performance. In this section, we describe a load balancing scheme to overcome this obstacle.

Figures 7 and 8 illustrate an example scenario which depicts how the proposed load distribution scheme works. In this scenario, there are four remote servers, $S_1$, $S_2$, $R_1$, and $R_2$. $R_1$ is a replica of $S_1$ and $R_2$ is a replica of $S_2$. A federated query, $Q_6$, is submitted to II and the query requires join operations across the two sources.

Query routing for load balancing can be performed at the query fragment processing or at the global query processing levels. We describe these two alternatives in the following sections.

#### 4.1 Load Balance at Query Fragment Level

In this example, $Q_6$ has two query fragments: $QF_3$ and $QF_4$. As shown in Figure 7, the calibrated costs of two query fragment plans, $QF_3_{\phi_1}$ using $S_1$ and $QF_3_{\phi_3}$ using $R_1$, are close to each other. If $QF_3_{\phi_1}$ and $QF_3_{\phi_3}$ are identical, they are exchangeable for global query processing purposes since they correspond to the same query fragment. When $QF_3_{\phi_1}$ is selected by II in the global plan, QCC can distribute the load across $S_1$ and $R_1$ by clustering $QF_3_{\phi_1}$ and $QF_3_{\phi_3}$ together and using them interchangeably. Note that exchangeable query fragment processing plans need to be identical, since two different query fragment processing plans may result in different global processing plans with dramatically different costs even they have an identical calibrated cost.

The guidelines QCC follows for load distribution can be summarized as follows:

- For each query fragment processing plan selected by II, QCC locates alternative plans that can execute the same query fragment with close calibrated costs (e.g. within 20%).
- QCC clusters these plans in a set and rotates in a round robin fashion for the future requests to the corresponding query fragment.

Note that the workload of the fragment query (i.e. calibrated cost times frequency of queries issued in a given period) must be greater than a preset threshold value in order for the fragment query to be considered load distribution. The process is repeated periodically as calibrated costs may change.

The load balancing at the query fragment processing plan level provides a simple implementation, but may miss certain global opportunities for load distribution. For example, this approach cannot provide load balance for federated queries with join operations across multiple remote sources.
Furthermore, it cannot provide load balancing for substitution of different query fragment processing plans of similar cost that result in similar global query processing costs.

4.2 Load Balance at Global Query Level

As illustrated in Figure 7, during the compile time, MW records possible execution plans and their costs for the two query fragments, QF3 and QF4, of Q6. The origin servers S1 and S2 can support two query fragment execution plans respectively while the replica servers R1 and R2 can support only single query fragment execution plans. As a result, the II query optimizer has nine global query execution plans to choose from. QCC has information of all query fragments and their estimated costs. However, since II query optimizer only stores the "winner plan" in the explain table, QCC does not have knowledge about other alternative plans and their costs.

To carry out load balance at the global query level, QCC needs to derive all possible global execution plans. QCC utilizes the simulated federated system shown in Figure 2 to generate all alternative global execution plans, Q6_p1 to Q6_p9, and estimate their calibrated costs. QCC achieves this by iterating through possible query fragment pairs one at a time at the wrapper level as shown on the right hand side of Figure 7. Calibrated costs of query fragments are used to estimate the cost of the global query plan. The cost of the alternative global query plans are then calibrated by the information integration cost calibration factor.

Once the calibration costs of all alternative query plans are derived, QCC can eliminate plans that are not promising. In our example, Q6_p1, Q6_p2, Q6_p3, Q6_p4, and Q6_p7 are eliminated since there are cheaper plans running on exactly the same set of remote servers. For example, both Q6_p1 and Q6_p5 are executed at S1 and S2. We would certainly select Q6_p5 over Q6_p1 since it is cheaper.

Next, QCC identifies that Q6_p5, Q6_p6, and Q6_p8 have similar costs (i.e., within 20%) and are executed on different sets of servers. Therefore, QCC can select Q6_p5, Q6_p6, and Q6_p8 to form a group of plans to recommends to II in a round robin fashion for Q6. By selecting the plans in this way, II distributes the load to multiple servers in a balanced way instead of to few servers.

The guidelines QCC uses to select query plans for round robin load distribution are as follows:

- For global query plans whose fragment queries are executed on the same set of servers, QCC picks the cheapest plan.
- QCC selects the cheapest plan and other alternative plans whose costs are close to that of the cheapest plan.

**Figure 7. Deriving and Costing Alternative Federated Query Plans**

**Figure 8. Selection of Federated Query Plans for Load Balance**
(e.g. within 20%) as a set of query plans to rotate in the round robin fashion.

The workload of a query (i.e. calibrated cost times frequency of queries issued in a period) must be greater than a preset threshold value in order for the query to be considered load distribution.

QCC can use additional information to reduce the cost of identifying an effective load distribution scheme. For instance, although in Figure 8 we show that there are nine alternative global query plans. The query fragments are labeled with the remote servers that return the fragments. In the actual implementation, QCC needs to execute $Q6$ in the explain mode only four times:

1. MW provides only the query fragment processing plans at $S1$ and $S2$ to II. The implementation is done by adjusting cost functions of $R1$ and $R2$ to infinity so that only the query fragment processing plans at $S1$ and $S2$ will be considered. $Q6,p5$ will be selected while $Q6,p1$, $Q6,p2$, and $Q6,p4$ will be eliminated.

2. MW provides only the query fragment processing plans at $S1$ and $R2$. $Q6,p8$ will be selected while $Q6,p7$ will be eliminated.

3. MW provides only the query fragment processing plan at $R1$ and $S2$. $Q6,p5$ will be selected while $Q6,p3$ will be eliminated.

4. MW provides only the query fragment processing plans at $R1$ and $R2$. $Q6,p8$ will be selected.

Furthermore, QCC does not need to include query fragments from all remote sources in deriving alternative global plans. Since QCC maintains the server cost calibration factors for all remote sources, it can exclude those remote sources with very high server cost calibration factors from being considered as candidates for query routing destinations.

5 Experiments

In this section, we experimentally evaluate the performance and reliability gains obtained using QCC in information integration. More specifically, we aim to observe whether (1) as the server loads change, the QCC can learn the query-cost calibration factors, successfully, (2) whether the cost factors learned by the QCC help II to pick better plans than those it would pick without transparent cost calibration by QCC, and whether (3) QCC can help the system achieve load balance by inducing the II and the MW to use different plans during each iteration.

In order to observe these three aspects of QCC deployment, we created an information integration scenario with one II server and three remote servers, each hosting an IBM DB2 DBMS. We populated the remote servers with tables from the sample database schema provided along with regular DB2 installments. Each table has been populated with randomly generated data. Furthermore, the tables are replicated and distributed on the three remote servers such that each server is involved in a diverse set of queries.

In order to observe and interpret the performance of the proposed query calibration mechanism, we experimented with a diverse set of query types, each with different remote source integration needs: (1) different query types need different numbers of tables, (2) the amount and type of II processing needed to merge data from multiple sources differ for different queries, and (3) the number of replicas available for the required tables varies from query to query. The tables sizes also varied, with small tables having on to order of 1000s of tuples and large tables having on the order of 100000s of tuples.

5.1 Procedure

Given these scenarios, we carried out the experiments as follows:

- **Step 1. Query fragment generation**: Queries in the workload are processed by II and the relevant query fragments are generated.
- **Step 2. Query-fragment cost estimation**: Query fragments obtained in the previous step are forwarded to the available servers in the explain mode and the corresponding costs are observed.
- **Step 3. Baseline query-fragment cost observation**: Query fragments obtained in the first step are forwarded to the available servers and the corresponding server response times are observed.
- **Step 4. Heavy-server-load query-fragment cost observation**: Servers are hit with a heavy update load, and the query fragments obtained in the first step are re-forwarded to the available servers and the corresponding heavy-load server response times are observed. Comparisons of heavy-load and light-load costs tell us whether the cost-factors monotonically increase as the load to the remote servers change. In addition, the analysis of the heavy-load costs tell us for what type of query fragments QCC can reliably generate cost factors.
- **Step 5. Workload execution based on estimated costs under high-server-load**: This step illustrates how II would perform under heavy remote server load without feedback from QCC.
- **Step 6. Workload execution based on observed costs under high-server-load**: Comparison of the results from this step with the results from the previous step illustrates the performance gains obtained with query cost calibration under heavy remote server load.
5.2 The Need for Dynamic Adjustment of Query Routing

In this experiment, we used different query fragment types (with different performance characteristics and resource requirements) to observe the sensitivity of query processing cost at the remote to its system load. The four query fragment types we include in Figure 9 are as follows: QT1 (i.e., query type 1) include queries with equijoin on two large tables (100000 tuples) followed by a “greater than” selection on the input parameter and an aggregation operation; QT2 is similar except that the selection table is small (1000 tuples). QT3 is similar to QT1 except that the selection condition is much more selective. Finally, QT4 joins has three tables and has a highly selective predicate. Figure 9 shows the response time of the three remote servers, under low and high load conditions, for different instances of the query fragment types.

The first thing to notice from these figures is that the three servers function differently from each other. Overall, S3 functions better than the others in most situations. Therefore, a naive system would pick S3 over the others most of the time, blindly of the load conditions and query
<table>
<thead>
<tr>
<th>Server</th>
<th>Phase1</th>
<th>Phase2</th>
<th>Phase3</th>
<th>Phase4</th>
<th>Phase5</th>
<th>Phase6</th>
<th>Phase7</th>
<th>Phase8</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Base</td>
<td>Base</td>
<td>Base</td>
<td>Base</td>
<td>Load</td>
<td>Load</td>
<td>Load</td>
<td>Load</td>
</tr>
<tr>
<td>S2</td>
<td>Base</td>
<td>Base</td>
<td>Load</td>
<td>Load</td>
<td>Base</td>
<td>Base</td>
<td>Load</td>
<td>Load</td>
</tr>
<tr>
<td>S3</td>
<td>Base</td>
<td>Load</td>
<td>Base</td>
<td>Load</td>
<td>Load</td>
<td>Base</td>
<td>Load</td>
<td>Load</td>
</tr>
</tbody>
</table>

Table 1. Combinations of Server Load Conditions

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Server Assignment</th>
<th>Phase1</th>
<th>Phase2</th>
<th>Phase3</th>
<th>Phase4</th>
<th>Phase5</th>
<th>Phase6</th>
<th>Phase7</th>
<th>Phase8</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT1</td>
<td>S1</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
</tr>
<tr>
<td>QT2</td>
<td>S2</td>
<td>S3</td>
<td>S2</td>
<td>S3</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S3</td>
<td>S3</td>
</tr>
<tr>
<td>QT3</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S3</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S3</td>
<td>S3</td>
</tr>
<tr>
<td>QT4</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
</tr>
</tbody>
</table>

Table 2. Comparison between Fixed Server Assignment and Dynamic Assignment (for Each Phase)

Figure 10. Benefits of QCC in Performance Gain over Fixed Assignment 1

Figure 11. Benefits of QCC in Performance Gain over Fixed Assignment 2

fragment types. However, we notice that for one of the costlier query types (QT2), S3 is much more sensitive to load than the others; as a consequence, if S3 is the only loaded server (which would be the case if S3 is blindly chosen as the preferred server), then the performance for query fragments of type QT2 will degrade so much that S1 and S2 will be more desirable. In fact, since the response times in this situation are fairly close to each other, a load balancing between these three servers for query type 2 would be desirable.

Another thing to notice is that in query type 3, S3 is the cheapest server, even when it is highly loaded and the other two are not loaded. Therefore, a static solution which chooses server simply based on the load on the server is also not acceptable. The performance highly depends on the real-time performance behavior of the servers for different query fragment types, which can be observed and used by QCC effectively as shown in the next subsection.

5.3 Benefit of QCC in Performance Gain

The next evaluation is conducted to validate the benefits of QCC in performance during various remote server load conditions. In Table 1, we show eight phases that the three remote servers are placed with different levels of load.

We assume a typical federated information system in which how federated queries are distributed to remote servers are fixed and predetermined in the phase of nickname definition registration (possibly with the help of explain mode help). We compare the response time of a prototype version of DB2 Information Integrator and of the same DB2 II with deployment of QCC for query routing in adapting to runtime environment changes. We construct a workload consistent of four query types (each with 10 different query instances) and the queries in the workload is uniformly distributed among four query types.

In the typical federated information system, the queries of query type 1 and query type 3 are registered to be forwarded to S1, the queries of query type 2 are registered to be forwarded to S2, and queries of query type 4 are registered to be forwarded to S3. The query routing decision is fixed at the nickname registration phase in contrast to the dynamic server assignment by QCC as shown on the right of Table 2.

Figure 10 shows the benefit of QCC in all phases in performance gain. With deployment of QCC, a typical federated information system can achieve an average of almost 50% performance gain in the experimental scenario. Note...
that even when all remote servers are heavily loaded, QCC still can improve the average response time by almost 60%.

As we notice earlier, S3 is the most powerful machine among the three available servers. One nature way of load distribution is to pick S3 as the default server. As shown in Figure 11, this assignment performs well most of the time. However, in three combinations of server load conditions, the system with deployment of QCC can still achieve an average of almost 20% performance gain.

The experimental results show that QCC is highly effective in adapting to dynamic changes in the runtime environment as well as in making accurate and timely decisions in query routing for performance. Furthermore, the federated information system with QCC can also benefit from improved fault tolerance as QCC can detect unavailability of the remote servers for query routing. These functions can not be done by manual monitoring and adjustment by the system administrators.

6 Related Work

The system described in this paper build on top of DB2’s federated query processing capability [10], which in turn derives from the literature on wrappers and federated query processing [3, 8, 21, 15, 19].

The closest prior art on measuring historical performance and using these measurements to adjust database cost models is the LEO project [18]. But the method of LEO as described is over a single database, whereas our work applies to federated databases. Specifically, the method of LEO assumes that a database system can be instrumented to get feedback, whereas we are dealing with remote data sources which are not easy to instrument directly (and which may disallow instrumentation). Our work instead infers their characteristics indirectly via the overall query behavior, such as response time and cardinality.

The DB2 II optimizer, as well as the SQL-MED standard [10, 11], allow users writing wrappers for external data sources to provide a cost function for accesses to that data source. The DB2 II optimizer has explicit logic for taking a total access cost from a remote data source provided by such a cost function, and incorporates it into the optimizer cost model, by breaking it down into individual components such as CPU, I/O cost [9]. But neither describes a method to implement this cost function. Our work complements this prior art well, because our method can be used to implement the cost function for a remote data source.

Another area of related prior art is work on calibrating the detailed cost characteristics of each operator in relational DBMSs, such as the cost of sequential scan, cost of merge join, etc. For example, [6] calibrates such detailed cost parameters by developing a ”cost-tuner component”, which a data source provider has to run in order to collect cost statistics. [5] creates a ”calibration database” on the data source, and runs a series of predefined queries over it. There are two problems with this approach which make it inapplicable to our situation. First, it assumes that the remote data source is willing to run this specialized software (such as the cost tuner component), or is willing to install a special calibration database, on the remote computer system. This is highly intrusive on an autonomous data source, and has serious security problems. Second, this approach assumes that the remote data source is relational. In contrast, the method of our invention is designed to calibrate a model of access characteristics of arbitrary data sources (e.g., relational, XML, file, web service, or otherwise). Moreover we are not allowed to run any custom program or change the database at the computer system housing the data source. This characteristic makes our method applicable to autonomous data sources.

The query processing and query optimization literature has focused mainly on queries that explicitly specify the data sources to be accessed. The closest analogy to our work in the literature is views (e.g., see [13]). The materialized view literature deals with substitution of query fragments with precomputed views to improve query performance (e.g., [23, 4, 22]). The serious limitation with views is that they are resolved syntactically, at query optimization time. This early-stage binding rules out any opportunity for routing queries to alternate data sources based on execution-time load or network conditions.

A recent paper [7] describes a method to substitute replicas for tables in a query if their staleness is within an application’s tolerance. However this is also an optimization-time method, and suffers from the same problem with views described above.

Some of the recent papers on adaptive query processing do tackle this issue. For example, Eddies [2] and distributed Eddies [20] continually monitor the speed of query operators and use this information to adjust the query plan by changing the ways tuples are routed. Our work is complementary to Eddies because we adapt query plans by changing the data source being accessed (whereas Eddies adapt the order of query operators). The work on dynamic adaptation of access methods in DEC RDB and in SteMs [1, 14] is also similar, in the sense that switching to an alternate data source is analogous to switching to an alternate access method. Unlike Eddies, our method does not involve any ongoing run-time overhead, the load at data sources is only verified the first time a source is accessed. For very long-running or continuous queries, we could extend our method to periodically re-check the load and switch data sources if needed; the open question is how we deal with duplicates.

In Mariposa [19] the federated optimizer solicits bids for performing query fragments. While Mariposa did such negotiation at optimization-time, one future direction for our project is to dynamically solicit bids during
query-execution, rather than simply calibrate the optimizer-estimated with runtime load conditions.

Finally, the work on multi-query optimization (e.g., [12, 17, 16]) is related to this paper in the sense that both optimize for a query workload, rather than a single query instance. While the multi-query optimization literature deals with methods to identify shared work in queries and reuse them, we take a different stab at the problem, by balancing loads on shared resources.

7 Conclusion

In this paper, we described a novel component query cost calibrator (QCC) to transparently calibrate the cost functions based on process and network latencies at the remote sources, as well as reliability and data availability of these sources. QCC adapts the cost functions used by a federated query processor to the runtime properties of the environment, and therefore indirectly influences the federated query optimizer, to enable the selection of the right remote sources. We have conducted experiments and the experimental results show the adaptiveness and effectiveness of the query cost calibration method. The future work includes dynamic tuning of the re-calibration cycles and incorporation of data placement strategies in conjunction with QCC into the proposed architecture.

Acknowledgments

The authors would like to acknowledge the contribution of Jun Rao at IBM Almaden and Daniel Zilio at IBM Toronto for their valuable suggestions on this work as well as Pedro Morais at IBM Silicon Valley Laboratories for his assistance in setting up the sample databases used in our experience.

References