Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System *

Mary Tork Roth† Fatma Özcan‡ Laura M. Haas§

IBM Almaden Research Center, San Jose CA 95120

Abstract

An important issue for federated systems of diverse data sources is optimizing cross-source queries, without building knowledge of individual sources into the optimizer. This paper describes a framework through which a federated system can obtain the necessary cost and cardinality information for optimization. Our framework makes it easy to provide cost information for diverse data sources, requires few changes to a conventional optimizer and is easily extensible to a broad range of sources. We believe our framework for costing is the first to allow accurate cost estimates for diverse sources within the context of a traditional cost-based optimizer.

1 Introduction

Increasingly, companies need to be able to interrelate information from diverse data sources such as document management systems, web sites, image management systems, and domain-specific application systems (e.g., chemical structure stores, CAD/CAM systems) in ways that exploit these systems' special search capabilities. They need applications that not only access multiple sources, but that ask queries over the entire pool of available data as if it were all part of one virtual database. One important issue for such federated systems is how to optimize cross-source queries to ensure that they are processed efficiently. To make good decisions about join strategies, join orders, etc., an optimizer must consider both the capabilities of the data sources and the costs of operations performed by those sources. Standard database optimizers have built-in knowledge of their (sole) store's capabilities and performance characteristics. However, in a world where the optimizer must deal with a great diversity of sources, this detailed, built-in modeling is clearly impractical.

Garlic is a federated system for diverse data sources. Garlic's architecture is typical of many heterogeneous database systems, such as TSIMMIS [PGMW95], DISCO [TRV96], and HERMES [ACPS96]. Garlic is a query processor [HFLP89]; it optimizes and executes queries over diverse data sources posed in an extension of SQL. Data sources are integrated by means of a *wrapper* [RS97]. In [HKWY97, RS97], we described how the optimizer and wrappers cooperate to determine alternative plans for a query, and how the optimizer can select the least cost plan, assuming it has accurate information on the costs of each alternative plan. This paper addresses how wrappers supply information on the costs and cardinalities of their portions of a query plan and describes the framework that we provide to ease that task. This information allows the optimizer to compute the cost of a plan without modifying its cost formulas or building in knowledge of the execution strategies of the external sources. We also show that cost-based optimization *is* necessary in a heterogeneous environment; heuristic approaches that push as much work as possible to the data sources can err dramatically.

Our approach has several advantages. It provides sufficient information for an optimizer to choose good plans, but requires minimal work from wrapper writers. Wrappers for simple sources can provide cost information without writing any code, and wrappers for more complex sources build on the facilities provided to produce more accurate information as needed. Our framework requires few changes to a conventional bottom-up optimizer. As a result, in addition to examining the full space of possible plans, we get the benefits of any advances in optimizer technology for free. The framework is flexible enough to accommodate a broad range of sources easily, and does not assume that sources conform to any particular execution model. We believe that our framework for costing is the first to allow accurate cost estimates for diverse sources within the context of a traditional cost-based optimizer.

The remainder of the paper is structured as follows. Sec-

*This work was partially supported by DARPA contract F33615-93-1-1339.

†*torkroth@almaden.ibm.com*

‡*fatma@cs.umd.edu; current address: Department of Computer Science, University of Maryland; partial funding provided by Army Research Laboratory contract DAAL01-97-K0135.*

§*laura@almaden.ibm.com*

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

tion 2 discusses the traditional approach to costing query plans. In Section 3, we present a framework by which these costing techniques can be extended to a heterogeneous environment. Section 4 shows how a set of four wrappers with diverse capabilities adapt this framework to provide cost information for their data sources. In Section 5, we present experiments that demonstrate the importance of cost information in choosing good plans, the flexibility of our framework, the accuracy it allows, and finally, that it works – the optimizer is able to choose good plans even for complex cross-source queries. Section 6 discusses related work, and in Section 7 we conclude with some thoughts about future directions.

2 Costing in a Traditional Optimizer

In a traditional bottom-up query optimizer [SAC+79], the cost of a query plan is the cumulative cost of the operators in the plan (plan operators, or *POPs*). Since every operator in the plan is the root of a subplan, its cost includes the cost of its input operators. Hence, the cost of a plan is the cost of the topmost operator in the plan. Likewise, the cardinality of a plan operator is derived from the cardinality of its inputs, and the cardinality of the topmost operator represents the cardinality of the query result.

In order to derive the cumulative costs and cardinality estimates for a query plan, three important cost numbers are tracked for each POP: *total cost* (the cost in seconds to execute that operator and get a complete set of results), *re-execution cost* (the cost in seconds to execute the POP a second time), and *cardinality* (the estimated result cardinality of the POP). The difference between total and re-execution cost is the cost of any initialization that may need to occur the first time an operator is executed. For example, the total cost of a POP to scan a temporary collection includes both the cost to populate and scan the collection, but its re-execution cost includes only the scan cost.

The total cost, re-execution cost, and cardinality of a POP are computed using *cost formulas* that model the runtime behavior of the operator. Cost formulas model the details of CPU usage and I/O (and in some systems, messages) as accurately as possible. A special subset of the formulas estimates predicate selectivity.

Cost formulas, of course, have variables that must be instantiated to arrive at a cost. These include the cardinality of the input streams to the operator, and *statistics* about the data to which the operator is being applied. Cardinality of the input streams is either computed using cost formulas for the input operators or is a statistic if the input is a base table. Hence, statistics are at the heart of any cost-based optimizer. Typically, these statistics include information about collections, such as the base cardinality, and about attributes, such as information about the distribution of data values. A traditional optimizer also has statistics about the physical system on which the data is stored, usually captured as a set of constant weights (e.g., CPU speed, disk transfer rate, etc.).

Figure 1 summarizes this flow of information. At the core

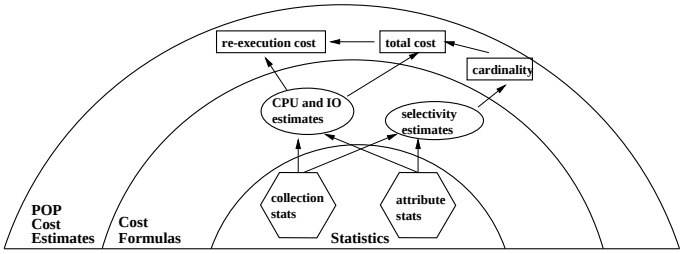

Figure 1: Traditional ost-based optimization

is a set of statistics that describe the data. At the next layer, these statistics feed cost formulas to compute selectivity estimates, CPU and I/O costs. Finally, in the outer layer, operator costs are computed from the cost formulas, and these operator costs ultimately result in plan costs.

3 Costing Query Plans in a Heterogeneous Environment

This section focuses on the process of costing query plans in a heterogeneous environment. Two significant challenges in adapting a traditional cost-based optimizer to a heterogenous environment are first, to identify *what* additional information is required to cost the portions of a query plan executed by remote sources, and second, *how* to obtain such information. Section 3.1 addresses the *what*, by introducing a framework for wrappers to provide information necessary to extend traditional plan costing to a heterogeneous environment. Section 3.2 addresses the *how*, by describing a default adaptation of the framework and facilities that a wrapper may use to compute cost and cardinality information for its data source.

3.1 A Framework for Costing in a Heterogeneous Environment

While the flow of information from base statistics to plan operator costs described in Section 2 works well in a traditional (relational) environment, it is incomplete for a heterogeneous environment. Given the diversity of data sources involved in a query, it is impossible to build cost formulas into the optimizer to compute the costs of operations performed by those data sources. Furthermore, since the data sources are autonomous, a single strategy cannot be used to scan the base data to gather and store the statistics the optimizer needs to feed its formulas. Clearly, a cost-based optimizer cannot accurately cost plans without cooperation from wrappers. In this section, we describe what information is needed from wrappers to extend cost-based optimization to a heterogeneous environment.

3.1.1 Cost Model

The first challenge for an optimizer in a heterogeneous environment is to integrate the costs of work done by a remote data source into the cost of the query plan. In Garlic, the portions of a query plan executed by data sources are encapsulated as `PUSHDOWN` POPs. Such POPs show up as leaves of the query plan tree. As a result, total cost, re-execution

cost, and result cardinality are all that is needed to integrate the costs of a PUSHDOWN POP into the cost of the query plan.

Fortunately, these three estimates provide an intuitive level of abstraction for wrappers to provide cost information about their plans to the optimizer. On one hand, these estimates give the optimizer enough information to integrate the cost of a PUSHDOWN POP into the cost of the global query plan without having to modify any of its cost formulas or understand anything about the execution strategy of the external data source. On the other hand, wrappers can compute total cost, re-execution cost, and result cardinality in whatever way is appropriate for their sources, without having to comprehend the details of the optimizer's internal cost formulas.

3.1.2 Cost Formulas

Wrappers will need cost formulas to compute their plan costs, and most formulas tailored to the execution models of the built-in operators will typically not be appropriate. On the other hand, some of the optimizer's formulas may be widely applicable. For example, the formula to compute the selectivity of a set of predicates depends on the predicates and attribute value distributions, and not on the execution model. Wrappers should be able to pick from among available cost formulas those that are appropriate for their data sources, and if necessary, develop their own formulas to model the execution strategies of their data sources more accurately.

Additionally, wrappers may need to provide formulas to help the optimizer cost new built-in POPs specific to a heterogeneous environment. For example, traditional query processors often assume that all required attributes can be extracted from a base collection at the same time. In Garlic, wrappers are not required to perform arbitrary projections in their plans. However, they must be able to retrieve any attribute of an object given the object's id. If a wrapper is unable to supply all requested attributes as part of its plan, the optimizer attaches a FETCH operator to retrieve the missing attributes. The retrieval cost may vary greatly between data sources, and even between attributes of the same object, making it impossible to estimate using standard cost formulas. Thus, to allow the optimizer to estimate the cost of this FETCH operator, wrappers are asked to provide a cost formula that captures the *access cost* to retrieve the attributes of its objects.

As another example, wrappers are allowed to export methods that model the non-traditional capabilities of their data sources, and such methods can be invoked by Garlic's query engine. Methods may be extremely complex, and their costs may vary greatly depending on the input arguments. Again, accurately estimating such costs using generic formulas is impossible. Wrappers are asked to provide two formulas to measure a method's costs: *total method cost* (the cost to execute the method once), and *re-execution method cost* (the cost to execute the method a second time). These formulas provide an intuitive level of abstraction for the wrapper, yet give the optimizer enough information to integrate method invocation costs into its operator costs.

3.1.3 Statistics

Both the optimizer and the wrappers need statistics as input to their cost formulas. In a heterogeneous environment, the base data is managed by external data sources, and so it becomes the wrapper's task to gather these statistics. Since wrappers provide the cost estimates for operations performed by their data sources, the optimizer requires only *logical* statistics about the external data. Statistics that describe the physical characteristics of either the data or the hardware of the underlying systems are not necessary or even helpful; unless the optimizer actually models the operations of the data sources, it would not know how to use such statistics.

A traditional optimizer's collection statistics include base cardinality, as well as physical characteristics (such as the number of pages it occupies), which are used to estimate the I/O required to read the collection. In a heterogeneous environment, the optimizer still needs base cardinality statistics to compute cardinality estimates for its operators.

For attributes, optimizers typically keep statistics that can be used to compute predicate selectivity assuming a uniform distribution of values, and some physical statistics such as the average length of the attribute's values. More sophisticated optimizers keep detailed distribution statistics for oft-queried attributes. In a heterogeneous environment, an optimizer still needs some attribute statistics in order to compute accurate cardinality estimates. In Garlic, wrappers are asked to provide uniform distribution statistics (number of distinct values, second highest and second lowest values). They may optionally provide more detailed distribution statistics, and the optimizer will make use of them. Physical statistics such as average column length are not required, although they may be helpful to estimate the cost to operate on the data once it is returned to Garlic. If not provided, the optimizer estimates these costs based on data type.

Not only are these statistics needed for the optimizer's formulas, but wrappers may need them as input to their private cost formulas. In addition, wrappers may introduce *new* statistics that only their cost formulas use. Such statistics may be for collections, attributes, or methods. For example, the cost formulas a wrapper must provide to estimate the total and re-execution costs of its methods are likely to require some information as input. Thus, as with cost formulas, the set of statistics in a heterogeneous environment must be extensible.

To summarize, Figure 2 shows the extended flow of information needed for an optimizer in a heterogeneous environment. White objects represent information that is produced and used by the optimizer. Objects with horizontal lines (e.g., the formula to compute predicate selectivity) are provided by the optimizer and made available to the wrappers. Those with vertical lines are provided by the wrappers, and used by both the optimizer and the wrappers. Statistics and cost formulas shown shaded in gray are introduced by and available only to wrappers. The outer circle shows that wrappers are asked to report the total cost, re-execution cost, and result cardi-

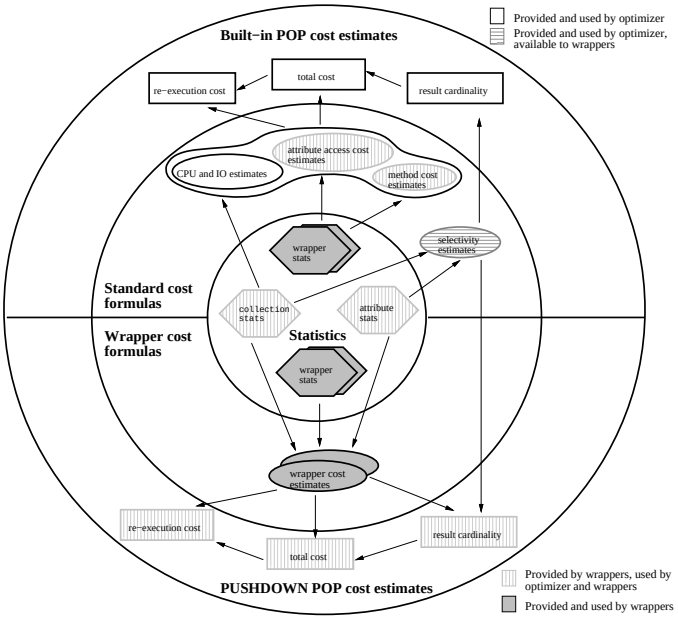

Figure 2: Heterogeneous cost-based optimization

nality for their plans. Armed with this information, the optimizer can combine the costs of PUSHDOWN POPs with the costs of built-in POPs to compute the cost of the query plan. In the next circle, wrappers are asked to provide formulas to compute attribute access costs and method costs. In addition, they can make use of some existing cost formulas, and add new formulas to model the execution strategies of their data sources. Finally, in the inner circle, wrappers are asked to provide the basic statistics about their collections and the attributes of their objects that the optimizer needs as input to its formulas. They may also compute and store statistics that are required by their own formulas.

3.2 Completing the Framework

Figure 2 shows how our framework extends the traditional flow of cost information to include wrapper input at all levels. To make it easy to provide such information (particularly for simple data sources), the framework also provides a default cost model, default cost formulas, and a facility to gather statistics. The framework is completely flexible; wrappers may use any of the defaults provided, or choose to provide their own implementations.

3.2.1 Extending the Cost Model

As described in Section 3.1.1, a wrapper's first job is to report total cost, re-execution cost, and result cardinality for its plans. To make this task as easy as possible, the framework includes a default cost model which wrappers can use to model the execution strategies of their data sources. Wrappers can take advantage of this cost model, or, if it is not sufficient, replace it with a cost model of their own.

The default cost model was designed with simple data sources in mind. We chose this approach for two important reasons. First, simple data sources have very basic capabilities. They can iterate over the objects in their collections, and perhaps apply basic predicates. They do not perform joins or other complex SQL operations. This limited set of capabilities often means that their execution strategy is both straightforward and predictable. These characteristics make it easy to develop a general purpose cost model. Second, an important goal of Garlic is to ensure that writing a wrapper is as easy as possible. If the default cost model is complete enough to model very basic capabilities, then wrapper writers for simple data sources need not provide *any* code for computing costs.

	Cost Model
P1	$plan\_total\_cost = reset\_cost + advance\_cost \times ((result\_cardinality + 1)/BLOCK\_SIZE)$
P2	$plan\_reexecution\_cost = plan\_total\_cost - reset\_cost$
P3	$plan\_result\_cardinality = \prod_{i=1}^{n} BASE\_CARD_i \times applied\_predicates\_selectivity$

Table 1: Default cost model estimates for wrapper plans

The default cost model is anchored around the execution model of a runtime operator. Regardless of whether a runtime operator represents a built-in POP or a PUSHDOWN POP, its work can be divided into two basic tasks[1]: *reset*, which represents the work that is necessary to initialize the operator, and *advance*, which represents the work necessary to retrieve the next result. Thus, the total cost of a POP can be computed as a combination of the reset and advance costs. As shown in Table 1, the default model exploits this observation to compute the total and re-execution costs of a wrapper plan.

(P1), the formula to compute the total cost of a plan, captures the behavior of executing a PUSHDOWN POP once. The operator must be reset once, and advanced to retrieve the complete result set (plus an additional test to determine that all results have been retrieved). *BLOCK_SIZE* represents the number of results that are retrieved at a time. Default formulas to compute reset and advance costs are described in Section 3.2.2 below. The re-execution cost (P2) is computed by factoring out the initialization costs from the total cost estimate. Since PUSHDOWN POPs are leaf POPs of the query plan tree, the result cardinality estimate (P3) is computed by multiplying the cross product of the *n* collection base cardinalities accessed by the plan by the selectivity of the applied predicates. As described in Section 3.2.3, *BASE_CARD* is the basic collection cardinality statistic, and *applied_predicates_selectivity* can be computed using the standard selectivity formula provided by the optimizer.

3.2.2 Extended Cost Formulas

Our framework provides default implementations of all the cost formulas wrappers need to supply (including those intro-

[1]Our model actually has three tasks; we are omitting discussion of the bind task to simplify exposition. *Bind* represents the work needed to provide the next set of parameter values to a data source, and can be used, for example, to push join predicate evaluation down to a wrapper. However, simple sources typically don't accept bindings, as they cannot handle parameterized queries. Our relational wrapper does accept bindings, and provides cost formulas to calculate their cost.

	Cost formula
F1	$access\_cost(A) = AVG\_ACCESS\_COST_{max} + (n-1) \times OVERHEAD \times AVG\_ACCESS\_COST_{max}$
F2	$method\_total\_cost_i = AVG\_TOTAL\_METH\_COST_i$
F3	$method\_reexecution\_cost_i = AVG\_REEX\_METH\_COST_i$
F4	$reset\_cost = AVG\_RESET\_COST_c$
F5	$advance\_cost = AVG\_ADVANCE\_COST_c$

Table 2: Default cost formulas

duced in Section 3.1.2) as well as those needed by the default cost model of Section 3.2.1. These formulas are summarized in Table 2, and we will describe each formula in greater detail below. They rely on a new set of statistics, and Section 3.2.3 describes how these statistics are computed and stored.

(F1) is the default definition of the attribute access cost formula. *A* represents a set of *n* attributes to be retrieved by a FETCH POP. Typically there is a significant charge to retrieve the first attribute, but only an incremental charge to retrieve additional attributes once the first attribute has been retrieved. $AVG\_ACCESS\_COST_i$ is a new attribute statistic that measures the cost to retrieve attribute *i*, and $AVG\_ACCESS\_COST_{max}$ is the most expensive attribute retrieved by the FETCH. *OVERHEAD* is a constant multiplier between 0 and 1 that represents the additional cost to retrieve an attribute, assuming that the most expensive attribute in *A* has already been retrieved. Wrappers may adjust this value as appropriate.

(F2) and (F3) represent the default definitions provided by the framework for the optimizer's method cost formulas. *AVG_TOTAL_METH_COST* and *AVG_REEX_METH_COST* are new statistics that represent measures of the average total and re-execution costs to invoke a method. This information is similar to the information standard optimizers keep for user-defined functions [Cor97]. These statistics are extremely simple, and do not, for example, take into account the set of arguments that are passed in to the method. As we will illustrate in Section 4.2, wrappers that use methods to export the nontraditional capabilities of a data source may provide new definitions that match the execution strategy of their underlying data source more accurately, including any dependency on the method's arguments.

Formulas (F4) and (F5) are the default cost formulas used by the default cost model to compute plan costs. For simple wrappers with limited capabilities, computing average times over the range of queries that the data source supports may often be sufficient to obtain accurate cost estimates. The default cost formulas to compute plan costs use this approach. While these formulas are admittedly naive, in Section 5.2 we show that they work remarkably well for the simple data sources in our experiments. Since simple wrappers typically do not perform joins, the reset and advance costs are computed to be the average reset and advance costs of the single collection *c* accessed by the plan. *AVG_RESET_COST* and *AVG_ADVANCE_COST* are new collection statistics that represent measures of the average time spent initializating and retrieving the results for queries executed against a collection.

If these default cost formulas are not sufficient for a particular data source (and they won't be for more capable sources), a wrapper writer may provide formulas that more accurately reflect the execution strategy of the data source. In fact, our framework for providing cost formulas is completely extensible; wrappers may use the optimizer's predicate selectivity formulas, any of the default formulas used by the default cost model, or add their own formulas. Wrapper-specific formulas can feed the formulas that compute operator costs and cardinalities, and their implementations can make use of the base statistics, and any statistics wrappers choose to introduce.

Category	Statistic	Query template
Collection	BASE_CARD	select count(*) from collection
	AVG_RESET_COST*, AVG_ADVANCE_COST*	select c.OID from collection c
Attribute	NUM_DISTINCT_VALUES	select count(distinct c.attribute) from collection c
	2ND_HIGH_VALUE	select c.attribute from collection c order by 1 desc
	2ND_LOW_VALUE	select c.attribute from collection c order by 1 asc
	AVG_ACCESS_COST*	select c.attribute from collection c
Method	AVG_TOTAL_METH_COST*, AVG_REEX_METH_COST*	select c.method(args) from collection c

Table 3: Statistics generated by update_statistics facility

3.2.3 Gathering Statistics

As described in Section 3.1.3, both the standard cost formulas and wrapper-provided cost formulas are fueled by statistics about the base data. Garlic provides a generic *update_statistics* facility that wrappers can use to gather and store the necessary statistics. The update_statistics facility includes a set of routines that execute a workload of queries against the data managed by a wrapper, and uses the results of these queries to compute various statistics. Wrappers may use this facility "as-is", tailor the query workload to gain a more representative view of the data source's capabilities and data, or augment the facility with their own routines to compute the standard set of statistics and their own statistics.

Table 3 describes the default query workloads that are used to compute various statistics. From the table, it should be clear how the standard statistics are computed. However, the calculations for the newly introduced statistics (marked with an asterisk) bear further description. For collections, the default cost model described in Section 3.2.1 relies on statistics that measure the average time to initialize and advance a wrapper operator. These measures are derived by executing a workload of single-collection queries defined by the wrapper writer (or DBA) that characterizes the wrapper's capabilities. For example, for a simple wrapper, the workload may contain a single simple "`select c.OID from collection c`" query, executed multiple times. Running averages of the time spent in reset and advance

Wrapper	Code	Cost model	Cost formulas	Statistics
ObjectStore	0	default	default	default
Lotus Notes	0	default	default	default
QBIC	700	default	replaces method cost, reset, advance formulas	added method statistics
Relational	400	default	replaces reset, advance formulas	added collection statistics

Table 4: Wrapper adaptations of framework

of the wrapper's runtime operator are computed for this workload of queries, and those measures are stored as the *AVG_RESET_COST* and *AVG_ADVANCE_COST* statistics. Note that these times include network costs, so the plan cost formulas do not have to consider network costs explicitly.

To compute the new attribute statistic *AVG_ACCESS_COST* (used by the default cost formula to compute attribute access cost), a single query which projects the attribute is executed, and the optimizer is forced to choose a plan that includes a `FETCH` operator to retrieve the attribute. This query is executed multiple times, and a running average of the time spent retrieving the attribute is computed, including network costs.

For the new method statistics, a workload of queries which invoke the method with representative sets of arguments (supplied by the wrapper writer) is executed multiple times. Running averages are computed to track the average time (including network costs) to execute the method both the first time, and multiple subsequent times. These averages are stored as the *AVG_TOTAL_METH_COST* and *AVG_REEX_METH_COST* statistics, respectively.

4 Wrapper Adaptations of the Framework

Figure 2 shows how our framework enables wrapper input in each concentric circle. In this section, we describe how a set of wrappers have adapted this framework to report cost information about their data sources to the optimizer. These wrappers represent a broad spectrum of capabilities, from the very limited capabilities of our complex object repository wrapper to the very powerful capabilities of our wrapper for relational databases. Table 4 summarizes how these different wrappers adapted the framework to their data sources, as well as the number of lines of code involved in the effort.

4.1 Simple Data Sources

We use ObjectStore as a repository to store complex objects, which Garlic clients can use to bind together existing objects from other data sources. This wrapper is intentionally simple, as we want to be able to replace the underlying data source without much effort. This wrapper will only generate plans that return the objects in the collections it manages, i.e., it will only produce plans for the simple query "`select c.OID from collection c`". The optimizer must append Garlic POPs to the ObjectStore wrapper's plans to fetch any required attributes and apply any relevant predicates. Because of its simplicity, the wrapper uses the default cost model to compute its plan costs and cardinality estimates, and uses the update_statistics facility "as-is" to compute and store the statistics required to fuel the default formulas, as well as those used by the optimizer to cost the appended Garlic POPs. We will see in Section 5.2 that the default model is indeed well-suited to this very basic wrapper.

We also implemented a more capable wrapper for Lotus Notes databases. It can project an arbitrary set of attributes and apply combinations of predicates that contain logical, comparison and arithmetic operators. It cannot perform joins. We have observed that the execution strategy for Lotus Notes is fairly predictable. For any given query with a set of attributes to project and set of predicates to apply, Lotus will retrieve each object from the collection, apply the predicates, and return the requested set of attributes from those objects that survive the predicates.

We intended to demonstrate with this wrapper that only a few modifications by the wrapper were needed to tailor the default cost model to a more capable wrapper. However, as we will show in Section 5.2, we discovered that although the wrapper for Lotus Notes is much more capable than the ObjectStore wrapper, the behavior of its underlying data source is predictable enough that the simple default cost model is still suitable. The wrapper writer was only required to tailor the workload of queries used to generate the *AVG_RESET_COST* and *AVG_ADVANCE_COST* collection statistics so that they more accurately represented the data source's capabilities. We used a simple workload of queries; techniques described in [ZL98] might do a better job of choosing the appropriate sample queries.

4.2 Data Sources with Interesting Capabilities

QBIC [N+93] is an image server that manages collections of images. These images may be retrieved and ordered according to features such as average color, color histogram, shape, texture, etc. We have built a wrapper for QBIC that models the average color and color histogram feature searches as methods on image objects. Each method takes a sample image as an argument, and returns a "score" that indicates how well the image object on which the method was invoked matched the sample image; the lower the score, the better the match. These methods may be applied to individual image objects via the Garlic method invocation mechanism. In addition, the QBIC wrapper will produce plans that apply these methods to all image objects in a collection, and return the objects ordered from best match to worst match. It can also produce plans that return the image objects in an arbitrary order. It does not apply predicates, project arbitrary sets of attributes, or perform joins.

Both the average color feature searches and color histogram feature searches are executed in a two-step process by the QBIC image server. In the first step, an appropriate 'color value' is computed for the sample image. In the second step, this value is compared to corresponding pre-computed

Feature	*sample_eval_cost*	*comparison_cost*
Average color	*AVG_COLOR_SLOPE* × *sample_size* + *AVG_COLOR_INTERCEPT*	*AVG_COLOR_COMPARE*
Color histogram	*AVG_HISTOGRAM_EVAL*	*HISTOGRAM_SLOPE* × (*number_of_colors*) + *HISTOGRAM_INTERCEPT*

Table 5: QBIC wrapper cost formulas

values for the images in the scope of the search. In our implementation, the scope is a single object if the feature is being computed via method invocation, or all objects in the collection if the feature is being computed via a QBIC query plan. For each image in the collection, the relationship between the sample image's color value and the image's color value determines the score for that image. Hence, the cost of both feature searches can be computed using the following general formula:

$$search\_cost = sample\_eval\_cost + m \times comparison\_cost$$

In this formula, *sample_eval_cost* represents the cost to compute the color value of the sample image, *comparison_cost* represents the cost to compare that value to a collection image object's corresponding value, and *m* is the number of images in the scope of the search.

In an average color feature search, the color value represents the average color value of the image. The execution time of an average color feature search is dominated by the first step and depends upon the x-y dimensions of the sample image; the larger the image, the more time it takes to compute its average color value. However, the comparison time is relatively constant per image object. The first entry in Table 5 shows the formulas the wrapper uses to estimate the cost of an average color feature search. *AVG_COLOR_SLOPE*, *AVG_COLOR_INTERCEPT*, and *AVG_COLOR_COMPARE* are statistics the wrapper gathers and stores using the update_statistics facility. The wrapper uses curve fitting techniques and a workload of queries with different sizes for the sample image to compute both the *AVG_COLOR_SLOPE* and *AVG_COLOR_INTERCEPT* statistics. *AVG_COLOR_COMPARE* represents the average time to compare an image, and an estimate for it is derived using the same workload of queries.

In a color histogram feature search, the color value represents a histogram distribution of the colors in an image. In this case, the execution time is dominated by the second step. For the typical case in which the number of colors is less than six, QBIC employs an algorithm in which the execution time of the first step is relatively constant, and the execution time of the second step is linear in the number of colors in the sample image. The second entry of Table 5 shows the formulas the wrapper uses to compute the cost of color histogram searches. *AVG_HISTOGRAM_EVAL*, *HISTOGRAM_SLOPE*, and *HISTOGRAM_INTERCEPT* are new statistics, and *number_of_colors* represents the number of colors in the sample image. Again, the wrapper uses curve fitting and a workload of queries with different numbers of colors for the sample image to compute *HISTOGRAM_SLOPE* and *HISTOGRAM_INTERCEPT*. It uses the same workload of queries to compute an average cost for *AVG_HISTOGRAM_EVAL*.

The wrapper uses these formulas to provide both method cost estimates to the optimizer and to compute the costs of its own plans. The effort to provide these formulas and compute the necessary statistics was about 700 lines of code.

4.3 Sophisticated Data Sources

Our relational wrapper is a "high-end wrapper"; it exposes as much of the sophisticated query processing capabilities of a relational database as possible. Clearly, the default formulas are not sufficient for this wrapper. Vast amounts of legacy data are stored in relational databases, and we expect performance to be critically important. The time invested in implementing a more complete cost model and cost formulas for a relational query processor is well-worth the effort.

However, decades of research in query optimization show that modelling the costs of a relational query processor is not a simple task, and creating such a detailed model is not within the scope of this paper. We believe that an important first step is to implement a set of formulas that provide reasonable ball-park cost estimates, as such estimates may be sufficient for the optimizer to make good choices in many cases. With this goal in mind, for the relational wrapper, we chose to use the default cost model, and implement a very simple set of formulas to compute the reset and advance costs that use an oversimplified view of relational query processing:

$$reset\_cost = prep\_cost$$
$$advance\_cost = execution\_cost + fetch\_cost$$

In these formulas, *prep_cost* represents the cost to compile the relational query, *execution_cost* represents the cost to execute the query, and $fetch\_cost$ represents the cost to fetch the results. At a high level, *prep_cost* and *execution_cost* depend on the number of collections involved in the query, and $fetch\_cost$ depends on the number of results. The relational wrapper used curve fitting techniques and the update_statistics facility to compute and store cost coefficients for these formulas. The total number of lines to implement this was less than 400.

While this is an admittedly naive implementation, the estimates produced by this formula are more accurate than those from the default model, and provide the optimizer with accurate enough information to make the right plan choices in many cases. However, we do not claim that our implementation is sufficient for the general case. We believe many of the techniques applied in [DKS92] and the approaches of more recent work of [ZL98] could be adapted to work within the context of the relational wrapper, and present an interesting area of research to pursue.

5 Experiments and Results

In this section, we describe the results of experiments that show that the information provided by wrappers through our

	Query	Pushdown join time (secs)	Garlic join time (secs)	Card
Q1	**select** p.id, p1.id **from** professor p, professor p1 **where** p.id < p1.id **and** p.status = p1.status **and** p.aysalary > 115000 **and** p1.aysalary > 115000	369.47	1550.52	605401
Q2	**select** p.id, p1.id **from** professor p, professor p1 **where** p.id < p1.id **and** p.status < p1.status **and** p.aysalary > 115000 **and** p1.aysalary > 115000	6332.14	1766.11	2783677

Table 6: A comparison of execution times for 2 join queries

framework is critical for the optimizer to choose quality execution plans. Without wrapper input, the optimizer can (and will) choose bad plans. However, with wrapper input, the optimizer is able to accurately estimate the cost of plans. As with any traditional cost-based optimizer, it may not always choose the optimal plan. However, for most cases, it chooses a good plan and avoids bad plans.

We adapted the schema and data from the BUCKY benchmark [C+97] to a scenario suitable for a federated system. We used only the relational schema, distributed it across a number of sources, and added to it a collection of images representing department buildings. We developed our own set of test queries that focus on showing how the optimizer performs when data is distributed among a diverse set of data sources. The test data is distributed among four data sources: an IBM DB2 Universal Database (UDB) relational database, a Lotus Notes version 4.5 database, a QBIC image server, and an ObjectStore version 4.0 object database. For the experiments, the query engine executed on one machine, the UDB database, QBIC image server, and ObjectStore database all resided on a second server machine, and the Notes server resided on a third machine. All were connected via a high-speed network. When an execution plan included a join, we limited the optimizer's choices to nested loop join and push-down join. This did not affect performance, and allowed us to illustrate the tradeoffs in executing a join in Garlic or at a data source without having to consider countless alternative plans. It should be noted that Garlic is an experimental prototype, and as such, the Garlic execution engine is slower than most commercial relational database product engines. However, it is significantly faster than Notes. Hence, we believe our test environment is representative of a real world environment in which some sources are slower and some faster than the middleware, and hence, is a fair testbed for our study.

5.1 The Need for Wrapper Input

This first set of experiments addresses the need for cost-based optimization in an extensible federated database system. It has been suggested [ACPS96, EDNO97, LOG93, ONK+96, SBM95] that heuristics that push as much work as possible to the data sources are sufficient. Consider the two queries

ID	Department predicate	Professor predicate	Cardinality
Q3	dno < 1	id = 101	0
Q4	dno < 51	id = 101	50
Q5	dno < 101	id = 101	100
Q6	dno < 151	id = 101	150
Q7	dno < 201	id = 101	200
Q8		id < 102	250
Q9		id < 103	500
Q10		id < 105	1000
Q11		id < 107	1500
Q12		id < 109	2000

Table 7: UDB professorxdepartment predicates

defined in Table 6. (Q1) finds all pairs of similarly ranked professors that make more than $115,000 a year. (Q2) finds, for all professors that make at least $115,000 a year, the set of professors of a lower rank that also make at least $115,000 a year. The professor collection is managed by the relational wrapper. There are two obvious plans to execute these queries: push both the join and the predicate evaluation down to the UDB wrapper, or push the predicate evaluation down to the wrapper but perform the join in Garlic. Table 6 also shows the result cardinality and execution times for these 2 plans. In (Q1), the equi-join predicate on status restricts the amount of data retrieved from the data source, so the pushdown join is a better plan. However, in (Q2), the join predicates actually increase the amount of data retrieved from the data source, so it is faster to execute the join in Garlic. These queries represent two points in a query family that ranges from an equi-join (p.status = p1.status) to a cross product (no predicate on status). At some point in this query family, there is a *crossover point* at which it no longer makes sense to push the join down. The crossover point depends on different factors, such as the amount of data, the distribution of data values, the performance of both query engines, network costs, etc. Cost-based optimizers use such information to compare plan alternatives to identify where such crossover points exist, while heuristic approaches can only guess.

5.1.1 Working without wrapper input

The previous example motivated the need for cost-based optimization in a federated system by showing that pushing down as much work as possible to the data sources is not always a winning strategy. In this experiment, we show that accurate information is crucial for a cost-based optimizer to identify crossover points. For this set of experiments, we chose a family of queries over the UDB department and professor collections. To control result cardinality, we used a cross product with local predicates (shown in Table 7) on each table.

To predict plan costs accurately, a cost-based optimizer depends heavily on the availability and accuracy of statistics. If statistics are not available, the optimizer uses default values for these parameters. Without accurate information, the optimizer will sometimes choose a good plan, and sometimes it will not. In our environment, in the absence of wrapper input,

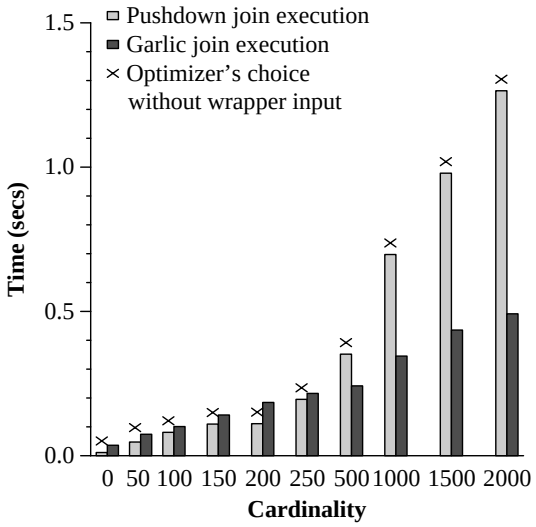

Figure 3: Optimizer choices without wrapper input

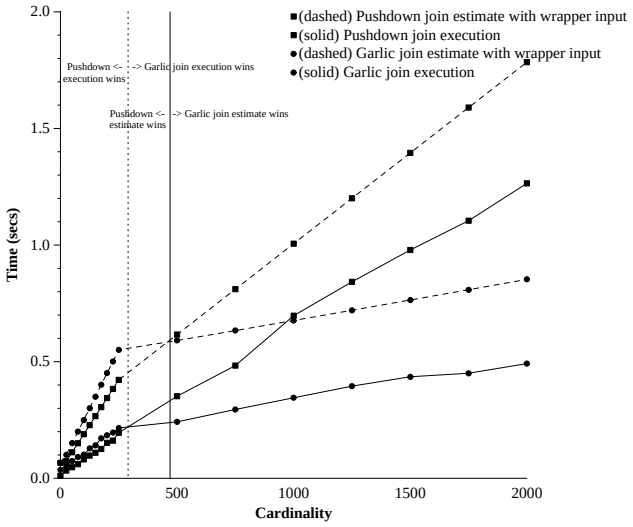

Figure 4: Optimizer estimates with statistics

the optimizer's parameters have been tuned to favor pushing as much work down to the data sources as possible.

For the set of queries in Table 7, the crossover point at which it makes sense to execute the join in Garlic occurs between queries (Q8) and (Q9), or when the result cardinality is between 250 and 500. Figure 3 shows the execution times for executing these queries with both the pushdown join and Garlic join plans. For each query, an x marks the plan that was chosen by the optimizer. Since the optimizer does not have the benefit of wrapper input, it relies on its defaults, and favors the pushdown join plan in all cases. With only default values, the cardinalities of the base collections look the same, and all local predicates (e.g., d.dno $<$ 101 or p.id $<$ 102) have the same selectivity estimates. Without more accurate information, the optimizer cannot easily discriminate between plans.

5.1.2 Working with wrapper input

Consider the same set of queries, only this time with input from the UDB wrapper, using the cost model and formulas described in Section 4.3. Figure 4 shows both the optimizer's estimates and the execution times for both the pushdown and Garlic join plans. The graph shows that while the optimizer's estimates differ by 10% to 45% from the actual execution costs, the wrapper input allows the optimizer to compare the *relative* cost of the two plans. Keep in mind that the cost formulas implemented by the UDB wrapper are fairly naive; if the wrapper writer invested more effort in implementing cost formulas reflecting the execution strategies of UDB, the optimizer's estimates would be more accurate.

Now instead of favoring the pushdown plan in all cases, the optimizer recognizes a crossover point in which it makes sense to execute the join in Garlic. The vertical dotted line on the graph shows the actual crossover point. The vertical solid line on the graph shows the optimizer's estimate of the crossover point. The area between the two lines represents the range in which the optimizer may make the wrong choice. Since we didn't have a data point in this area of the graph, we ran further experiments to identify the range more accurately. These experiments used a few more predicates to allow us to control the result cardinality more precisely. We found that the execution crossover point is at cardinality $= 251$, and the optimizer identifies the crossover point in the 278-298 range. Thus, the range in which the optimizer will make the wrong choice is between 251 and at most 298. In this narrow range, the execution times of the plans are so close that the wrong plan choice is not significant.

5.2 Adaptability of the Framework

In the previous section, we showed that wrapper input is critical for the optimizer to choose good plans. In this section, we show that our framework makes it easy for wrappers to provide accurate input. We look at 3 wrappers in particular: ObjectStore, Notes, and QBIC.

5.2.1 Wrappers that Use the Default Cost Model

As described in Section 4.1, the ObjectStore wrapper is our most basic wrapper and uses the default cost model without modification. [ROH99] shows the optimizer's estimates and actual execution times for a set of queries that exercise the wrapper's capabilities. The experiments show that the defaults are well suited for the ObjectStore wrapper; the optimizer's estimates differ from the actual execution time by no more than 10%.

Recall that although Notes is a more capable wrapper, the Notes wrapper also uses the default cost model, formulas, and statistics. Again, [ROH99] shows that for a set of queries that exercise the wrapper's capabilities, the optimizer's estimates are "in the ballpark", ranging from a 13% to 40% difference from the actual execution time. For the experiments with more complicated queries, the optimizer's estimates are off by more than 30%. Further analysis showed that a significant percentage of this difference can be attributed to result cardinality underestimates, which were off by 21% for both of these queries. Such inaccuracies are not unusual for cost-

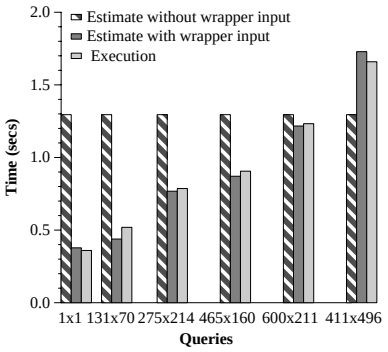

Figure 5: QBIC avg color query plan

based optimizers, and are the result of imperfect cost formulas and deviations in the data from the distribution assumptions. To make up the difference between the estimate and the actual execution time that cannot be attributed to inaccurate result cardinality estimates, the wrapper writer could provide formulas that model the predicate application strategy of Lotus Notes more accurately. However, we do not believe such effort is necessary. Analysis to be presented in section 5.3 shows that even for this more capable wrapper, the default cost formulas provide estimates that are close enough for the optimizer to choose good plans in most instances.

5.2.2 Wrappers with Interesting Capabilities

For data sources with unusual capabilities, such as QBIC, the default model is not sufficient. As described in Section 4.2, the execution time for an average color search depends on the size of the sample image. Figure 5 shows optimizer estimates and actual execution times for a family of average color queries with increasingly larger predicate images. The x-axis shows the size of the sample image. The first bar for each query represents the optimizer's cost estimate without wrapper input, the second bar shows the optimizer's cost estimate with wrapper input, and the third bar shows the actual execution time.

Without wrapper input, the optimizer has no knowledge of how much an average color search costs, nor is it aware that the cost depends on the size of the sample image. Thus, it must rely on default estimates, which can in no way approximate the real cost of the search or the plan. However, with wrapper input, the optimizer's estimates do reflect the dependency on the image predicate size, and its estimates are extremely accurate, with most being within 4% of the actual cost. An analysis of color histogram queries yields similar results. As we will see in Section 5.3, such input from wrappers with unusual capabilities is crucial for the optimizer to choose good plans when data from that source is joined with data from other sources.

5.3 Cross-Repository Optimization

Our final experiment shows that our framework provides sufficient information for the optimizer to choose good plans for complex queries. For this experiment, we used the query template given in Table 8 to generate a query family. The

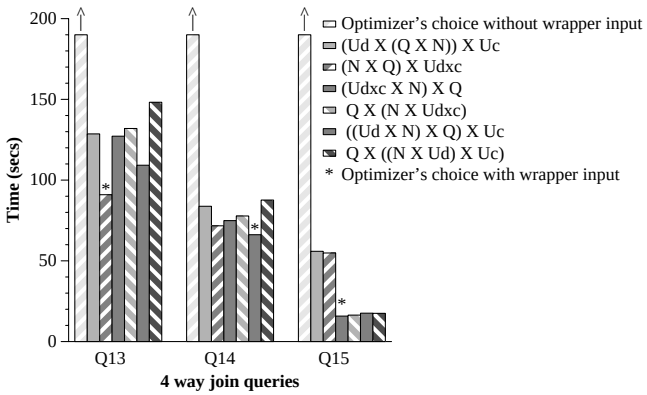

Figure 6: 4-way cross-repository join queries

Query template
select i.OID, i.avg_color('767x589_image.gif'), i.avg_color('1x1_image.gif') **from** images i, notes_departments n_d, udb_course u_c, udb_department u_d **where** n_d.building = i.image_file_name **and** u_c.dno = u_d.OID **and** u_d.dno = n_d.dno

Table 8: 4-way join query template

query template is a 4-way join between the department and course collections managed by the UDB wrapper, the Notes department collection, and the QBIC image collection. To generate the family, we added predicates on the UDB department collection and the UDB course collection that control the cardinality of the results. These predicates and the result cardinalities are shown in Table 9. The queries also contain 2 average color image searches, one of which is for a 1x1 image (cheap), while the other is for a 767x598 image (expensive).

The number of possible plans for executing this query family is over 200. However, a large number of these plans are clearly bad choices, as they would require computing large cross-products. We enumerated and forced the execution of the 20 most promising plans, including the ones the optimizer itself selected. In any plan, the optimizer is forced to push one average color search down and evaluate the other by method invocation because the QBIC wrapper returns plans that execute only one search at a time.

Figure 6 shows the execution time of 7 plans for each query. The first bar represents the plan the optimizer chose without statistics or wrapper input. The other 6 bars are representative plans from the set that we analyzed. The plans are

ID	Predicates	Card
Q13	u_d.budget $<$ 10000000 and u_c.cno $<$ 102	456
Q14	u_d.budget $<$ 6000000 and u_c.cno $<$ 102	258
Q15	u_d.budget $<$ 2000000 and u_c.cno $=$ 102	23

Table 9: 4-way join query predicates

denoted by the order in which the joins are evaluated. A collection is identified by the first character of the wrapper that manages it. The UDB collections are further marked by the first character of each collection. An upper case X indicates that the join was done in Garlic, and a lower case x indicates the join was pushed down to the UDB wrapper. A * over a bar indicates that the optimizer, working with wrapper input, chose the corresponding plan.

For all three queries, the optimizer picked the best plan of the alternatives we studied, and, we believe, of all possible plans. Note that this may not happen in general; the purpose of a cost-based optimizer is not to choose the optimal plan for every query, but to consistently choose good plans and avoid bad ones.

The graph once again reinforces the assertion that wrapper input is crucial for the optimizer to choose the right plan. Without wrapper input, the optimizer chose the same plan for all three queries, which was to push the join between the UDB collections down to the UDB wrapper, join the result of that with the Notes department collection, and join that result with the image collection. Without information from the QBIC wrapper about the relative costs of the two image searches, it arbitrarily picked one of them to push down, and the other to perform via method invocation. In this case, the optimizer made a bad choice, pushing the cheap search of the 1x1 image down to the QBIC wrapper, and executing the expensive search via method invocation on the objects that survive the join predicates. This plan is a bad choice for all three queries, with execution times well over 1000 seconds.

When the optimizer was given input from the QBIC wrapper about the relative cost of the two average color searches, it chose correctly to push the expensive search down to the QBIC wrapper and perform the cheap search via method invocation. This is true for all plans we looked at for all queries, and brings the execution times for all of our sample plans to under 200 seconds.

This experiment also shows that pushing down as much work as possible to the data sources does not always lead to the best plan. For (Q13) and (Q15), the best plan did in fact include pushing the join between the UDB collections down to the UDB wrapper. However, for (Q14), the best plan actually split these two collections, and joined UDB department with Notes department as soon as possible. In this plan, the predicate on the UDB department collection (u_d.budget $<$ 6000000) restricted the number of UDB department tuples by 50%. Joining this collection with the Notes department collection first also reduced the number of tuples that needed to be joined with the image collection by 50%. For (Q13), the UDB department predicate (u_d.budget $<$ 10000000) was not as restrictive. In this case, it would have only reduced the number of tuples that needed to be joined with the image collection by 9%, which was not a significant enough savings to make this alternative attractive. Instead, it was better to group UDB department and UDB course together and push the join down to the UDB wrapper.

For (Q15), the UDB department predicate is even more restrictive, filtering out over 90% of the tuples. In this case, it is a good idea to use it to filter out both the Notes department tuples and UDB course tuples as soon as possible. Thus, the two best plans push the join between the UDB collections down to the wrapper, and immediately join the result with Notes. The two worst plans failed to take advantage of this. Plan 2 in the figure arranged these collections out of order, and plan 3 joined the entire Notes department collection with QBIC image before the join with the UDB collections.

These experiments show that cost-based optimization is indeed critical to choose quality execution plans in a heterogeneous environment. Using our framework, wrappers can provide enough information for the optimizer to cost wrapper plans with a sufficient degree of accuracy. By combining such cost information with standard cost formulas for built-in operators, traditional costing techniques are easily extended to cost complex cross-source queries in a heterogeneous environment.

6 Related Work

As federated systems have gained in popularity, researchers have given greater attention to the problem of optimizing queries over diverse sources. Relevant work in this area includes work on multidatabase query optimization [LOG93, DSD95, SBM95, EDNO97, ONK+96] and early work on heterogeneous optimization [Day85, SC94], both of which focus on approaches to reduce the flow of data for cross-source queries, and not on estimation of costs. More recent approaches [PGH96, LRO96] describe various methods to represent source capabilities. Optimizing queries with foreign functions[CS93, HS93] is related, but these papers have focused on optimization algorithms, and again, not on estimating costs. [UFA98] describes orthogonal work to incorporate cost-based query optimization into query scrambling.

Work on frameworks for providing cost information and on developing cost models for data sources is, of course, highly relevant. OLE DB [Bla96] defines a protocol by which federated systems can interact with external data sources, but it does not address cross-source query optimization, and presumes a common execution model. The most complete framework for providing cost information to date is Informix's DataBlades [Cor97] architecture. DataBlades integrates individual tables, rather than data sources, and the optimizer computes the cost of an external scan using formulas that assume the same execution model as for built-in scans.

Various approaches have been proposed to develop cost models for external data sources. These approaches can be grouped into four categories: calibration [DKS92, GST96], regression [ZL98], caching [ACPS96], and hybrid techniques [NGT98]. The calibration and regression approaches typically assume a common execution model for their sources (which doesn't work for heterogeneous federations), but may be useful in developing wrapper cost models for particular

sources. Both [ACPS96] and [NGT98] deal with diverse data sources, but neither approach employs standard dynamic programming optimization techniques.

7 Conclusion

We have demonstrated the need for cost-based optimization in federated systems of diverse data sources, and we presented a complete yet simple framework that extends the benefits of a traditional cost-based optimizer to such a federated system. Our approach requires only minor changes to traditional cost-based optimization techniques, allowing us to easily take advantage of advances in optimization technology. Our framework provides enough information to the optimizer for it to make good plan choices, and yet, it is easy for wrappers to adapt. In the future, we intend to continue testing our framework on a broad range of data sources. We would like to add templates to support classes of data sources that share a common execution model, and test our framework for how well it handles object-relational features such as path expressions and nested sets.

8 Acknowledgements

We thank Peter Haas, Donald Kossmann, Mike Carey, Eugene Shekita, Peter Schwarz, Jim Hafner, Ioana Ursu and Bart Niswonger for their help in preparing this paper, and V.S. Subrahmanian for his support.

References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 137–148, Montreal, Canada, June 1996.

[Bla96] J. Blakely. Data access for the masses through ole db. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996.

[C+97] M. Carey et al. The bucky object-relational benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 135–146, Tucson, Arizona, US, May 1997.

[Cor97] Informix Corporation. Guide to the virtual table interface. Manual, 1997.

[CS93] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 529–542, Dublin, Ireland, 1993.

[Day85] U. Dayal. Query processing in a multidatabase system. In W. Kim, D. S. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*, pages 81–108. Springer, 1985.

[DKS92] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in heterogeneous DBMS. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 277–291, Vancouver, Canada, 1992.

[DSD95] W. Du, M.-C. Shan, and U. Dayal. Reducing multidatabase query response time by tree balancing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 293–303, San Jose, CA, USA, May 1995.

[EDNO97] C. Evrendilek, A. Dogac, S. Nural, and F. Ozcan. Multidatabase query optimization. *Distributed and Parallel Databases*, 5(1):77–114, 1997.

[GST96] G. Gardarin, F. Sha, and Z.-H. Tang. Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 378–389, Bombay, India, September 1996.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 377–388, Portland, OR, USA, May 1989.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J .Yang. Optimizing queries across diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, August 1997.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–276, Washington, DC, USA, May 1993.

[LOG93] H. Lu, B.C. Ooi, and C.H. Goh. Multidatabase query optimization: Issues and solutions. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 137–143, 1993.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 251–262, Bombay, India, September 1996.

[N+93] W. Niblack et al. The QBIC project: Querying images by content using color, texture and shape. In *Proc. SPIE*, San Jose, CA, USA, February 1993.

[NGT98] H. Naacke, G. Gardarin, and A. Tomasic. Leveraging mediator cost models with heterogeneous data sources. In *Proc. IEEE Conf. on Data Engineering*, Orlando, Florida, USA, 1998.

[ONK+96] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac. Dynamic query optimization on a distributed object management platform. In *Proc. of the International Conference on Information and Knowledge Management (CIKM)*, pages 117–124, Rockville, MD, USA, 1996.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, Miami, Fl, USA, December 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE Conf. on Data Engineering*, pages 251–260, Taipeh, Taiwan, 1995.

[ROH99] M. Tork Roth, F. Ozcan, and L. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. *IBM Technical Report RJ10141*, 1999.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, August 1997.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.

[SBM95] S. Salza, G. Barone, and T. Morzy. Distributed query optimization in loosly coupled multidatabase systems. In *Proc. of the Intl. Conf. on Database Theory (ICDT)*, pages 40–53, Prague, Czech Republic, January 1995.

[SC94] P. Scheuermann and E. I. Chong. Role-based query processing in multidatabase systems. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, pages 95–108, Cambridge, England, March 1994.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *Proc. of the Intl. Conf on Distributed Computing Systems (ICDCS)*, Amsterdam, The Netherlands, 1996.

[UFA98] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 130–141, Seattle, WA, USA, June 1998.

[ZL98] Q. Zhu and P. Larson. Solving local cost estimation problem for global query optimization in multidatabase systems. *Distributed and Parallel Databases*, 6:1–51, 1998.