Introducing FLO: A Federated Learning cOllector [*]

Tony Young - 20161423
CS 848 - Fall 2004

December 3, 2004

Abstract

In traditional (single-site) database systems, many methods of estimating statistics and performing cost-based optimization have been proposed. These methods make use of direct access to table and catalog data. Thus, accurate statistics can be computed and stored for use during optimization.

In distributed (multiple homogeneous site) database systems, methods unique to the distributed environment have been proposed to collect statistics and perform cost-based optimization. These methods however, still rely on direct access to table and catalog data. In fact, many optimization techniques that are used in traditional systems can be easily extended to distributed systems.

In federated (multiple heterogeneous site) database systems, new methods of collecting statistics and performing query optimization must be developed. These systems cannot assume they will have direct access to table or catalog data as some data sources do not create a catalog, or allow access to it. Thus, statistics must come from queries that applications perform. Further, methods employed in traditional and distributed systems cannot be directly applied or even extended to a federation because of local site autonomy.

In this paper, we present FLO: a Federated Learning cOllector. FLO "learns" and stores statistics in its catalog for later use in query optimization. A framework and algorithms for FLO are outlined and FLO's roots in current literature are presented. A body of experiments and unanswered research questions are discussed. As well, a discussion of current statistics collection and optimization methods will be presented. FLO will be compared to these methods.

[*]This research was supported by iAnywhere Solutions, 445 Wes Graham Way, Waterloo, Ontario, N2L 6R2

Contents

1 Introduction to Federation

It is 4:00 PM. John is sitting at his desk at Credit Co. finishing up a report regarding the Acme account. John seeks to provide Acme with a consolidated report about its various financial holdings around the world. Since John manages Acme's financial investments, he is trying to locate all of the company's holdings around the world. John has a painstaking task ahead of him. He must contact each firm that Credit Co. invests in and see if any of Acme's funds exist there. However, John has a secret weapon up his sleeve: a federated database system. Through the federated system, John can query all the firms that Credit Co. invests in around the world at once.

This paper provides an introduction to the various query optimization and statistics collection techniques employed today in federated (multi) database systems. A framework and algorithms for a new statistics gathering system, FLO, are also presented. A body of experiments and unanswered research questions are discussed. Throughout this document, federated database systems (FDBS's) and multidatabase systems (MDBS's) will be used to refer to one and the same concept: a system for integrating multiple heterogeneous (local) data sources into one global data source [27]. These products combine data from multiple database management systems (DBMS's) and make the data usable as though it resided on one central system. For John, this means being able to see Acme's holdings at each firm around the world simultaneously. John may believe he is looking at one database, but he is in fact looking at many databases that have been consolidated to give a global view of many pieces of local data.

1.1 Motivating Factors

As noted in [9, 13, 14], several factors motivated the development of federated database systems.

- Proliferation of Heterogeneous Databases Within an Organization: It is not uncommon for different departments within an organization to make use of their own database servers. Often, departments do not coordinate to ensure that a corporation is using a homogeneous DBMS to store data. As well, there is no guarantee that the schema individual departments use to store data will be homogeneous.

- Data Sharing Within Organizations: Many organizations seek to share data between different departments. For example, the finance department may require information regarding projects in progress in the marketing department. Such information sharing is difficult without the guarantee of schema, model or language homogeneity.

- Differing Rates of Technology Adoption: Different departments will adopt technology at different paces. For example, the information technology department is usually tech savvy. They will typically adopt new technology rapidly to store inventory information, service calls, etc. Alternatively, the human resources department is usually not very tech savvy. They may adopt technology slowly as they use many paper forms that are hard to replace efficiently with electronic counterparts. Further, newer systems that are deployed might come from the vender that is considered the "market leader" at the time of purchase.

- Mergers and Acquisitions: When companies join forces, their information systems must be joined as well. Old applications used by individual business units will depend on old (pre-merger/pre-acquisition) system. As well, users might be reluctant to learn an entirely new system. If the individual systems can be merged, each user can make use of their old applications and old access language, and the two systems can be viewed as one when required by a new application.

- Geographic Separation of Teams: Different teams may be broken up across geographic locations. For example, a company may have engineering teams located in Ontario and California. Different sites will host teams working on different projects. Each site may have their own information technology staff and make their own purchasing and installation decisions. Often there is no coordination between information technology departments at different sites. However, there are situations when databases need to be queried across sites. For example, human resources might want to know what projects each employee works on regardless of the site at which they work.

1.2 Query Optimization Challenges

Federated systems pose many new challenges for query optimization. Some of the challenges presented in [5, 19, 20, 31, 33] are presented here:

- Local Autonomy: There are several types of autonomy:

  1. *Data Autonomy* - Local database administrators have direct and complete control over the schemas, data types, relationships, etc. at their sites. This information cannot be modified in any way to make it more convenient for consolidation.
  2. *Design Autonomy* - Local database administrators decide when and how to replicate and fragment data.
  3. *Communication Autonomy* - Each site decides locally whether or not to communicate with other sites in the federation and the FDBS.
  4. *Execution Autonomy* - Each site can determine how, when and whether to execute global queries, as well as how queries are prioritized.

- Local Parameters: Local cost parameters for individual sites are not always available to the FDBS. For example, the FDBS may not know what indices are available for relations at local sites[1] as catalog data is not always accessible by the federation, and some data sources do not generate or export a catalog at all.

- Translation: Queries must be translated to and from the local schema, query language, and data model on-the-fly. This requires additional query processing time. As well, translations must be built and maintained by a database administrator.

[1]Nor can they predict which access methods will be used to execute a given query

- Heterogeneous Capabilities: Not all local sites have the same capabilities. For example, some sites may not implement any ranking operations. This means that intermediate results might have to be shifted to sites that can provide these capabilities, further increasing processing time.

- Additional Costs: Cost-based optimization at the federation needs to take into consideration some additional factors such as transmission speeds, network loads, local site configurations, etc. As with local parameters, this information is not always available to the FDBS.

Overall, one cannot assume any control over, or source of information for, local site cost parameters.

The rest of this article is organized as follows. Section 2 outlines the two main approaches to designing a FDBS. Section 3 gives an overview of FLO, the Federated Learning cOllector. Section 4 discusses some query optimization techniques employed by FDBS's. Statistics collection methods proposed for federated systems are presented in Section 5. Section 6 provides some concluding remarks.

2 Approaches to Federation

It is important for the reader to understand how data is organized and accessed in a federated system. This understanding will help to illustrate how posed queries can benefit from optimization, and statistics gathering can be performed.

As we see in [22, 31], there are two main approaches to designing a federated database system. The *multidatabase language approach* makes use of a specialized multidatabase language to submit and optimize queries. Thus, the burden of integration is left to the system users and, to a lesser extent, the local database administrators. On the other hand, the *global schema approach* makes use of a global database schema composed of a translated view of local database schemas. Thus, integration falls to a global database administrator who defines the global schema and data transformations to be applied.

2.1 Multidatabase Language Approach

As mentioned above, the multidatabase language approach makes use of a specialized query language to submit and optimize global queries [7]. Thus, the burden of integration is left to the system users and, to a lesser extent, the local database administrators. Local database administrators must provide schemas and semantic information about the data at their sites. Users must then formulate their queries in order to access data from these multiple sites at once. With a multidatabase language, users have much more control over the data they are querying. The heterogeneity of local DBMS's can be exploited to make use of language features implemented by one system that are not implemented by another. For example, some systems may not implement ranking. However, data can be gathered and ranked at one site before being returned to the user.

Since no global schema or translations need to be created, no global database administrator is necessary. Once the system is setup, it can be used without significant maintenance being performed. However, the multidatabase language approach requires users to have a significant amount of information about local DBMS's. Users must know the structure and semantics of each local schema in the federated system in order to make use of it. As well, explicit use of these sites is required when writing a query.

2.2 Global Schema Approach

As mentioned above, the global schema approach makes use of a global database schema composed of a translated view of local database schemas [2]. Thus, integration falls to a global database administrator who defines the global schema and data transformations/translations to be applied when submitting a query to local sites. With this approach, users see a unified view of all federated data. They do not need to know how data is represented at the local sites they are accessing. The database administrator combines the local schemas into a global one and generates the translation rules between global and local column names, relation names and data formats. Functional compensation can also be provided by the FDBS to perform operations on data from sites that are unable to perform some operations themselves (ex: ranking, grouping, etc.).

Although a global database administrator is necessary for this approach, the burden of integration is lifted from the user (i.e. users do not need to know semantic information about the underlying local DBMS's). However, unlike multidatabase language systems, a global schema will need to be maintained over time as sites, databases, relations and columns are added or updated.

3 Meet FLO

The Federated Learning cOllector (FLO) builds on statistics collection methods proposed in [3, 19, 20, 26, 34]. FLO has the ability to "learn" statistical information through the queries performed by users of the system.

3.1 Design Goals

The design goals of FLO are simple:

1. *Missing Catalogs*: The main design gaol of FLO is to provide statistics when none are directly available from data sources. Thus, FLO aims to provide statistics in those instances that local parameters are not available, such as when a catalog file does not exist or cannot be queried.

2. *Overhead at Local Sites*: FLO aims to avoid imposing extra overhead at local sites. We do not want to disrupt the operation of local sites or upset their administrators and users. Thus, FLO has been designed to exploit user query data to avoid imposing additional overhead.

3. *Overhead at the Federation*: FLO aims to avoid imposing extra overhead at the federation. We do not want to make users wait any longer for their query result than they have to. Thus, FLO has been designed to calculate statistics offline to avoid imposing additional processing delays on user queries.

3.2 Input from Current Literature

As mentioned above, FLO's roots are in the current literature. This subsection explains how FLO builds on various techniques.

- *LEO*: IBM's LEarning Optimizer learns from mistakes made in selectivity and other statistical estimations [26]. When a statistic is calculated, LEO stores the calculated value and the actual value in a table for later analysis. Upon analysis, LEO might discover its estimations of statistic x are off by a factor of y. LEO might then decide to augment all estimations for x by factor y in an attempt to make more accurate optimization decisions.

  Although it does not learn from mistakes, FLO "learns" all statistics through analysis of data that was retrieved through user queries. Statistics will constantly be refined as new queries are performed. Also, FLO uses a decay function to keep statistics relatively constant as statistics extracted from query results fluctuate. Thus, high variances found between some queries can be smoothed out over time.

- *SIT's*: Researchers proposed a method, currently employed by Microsoft, to "learn" cardinality and other statistics of joins, etc. from query execution in much the same way as LEO does [3]. With this method, statistics from query execution can be kept and reused the next time a query is submitted. This means statistics do not need to be recalculated, and the same calculation mistakes are not repeated, each time a query is run.

  The Microsoft algorithm assumes that statistics are already available in order to calculate selectivities, etc. In addition, this algorithm learns about mistakes made in the estimation process in much the same way LEO learns. FLO introduces a way to gather base statistics, not selectivities.

- *Subquery Execution Monitor*: A subquery execution monitor was proposed by Lu et al [19, 20]. This monitor keeps track of the amount of time queries take to execute at local sites, as well as the predicted execution time of those queries. If the monitor finds that the estimates are not accurate for a site, it adjusts the execution plan in order to avoid sending data and queries to it (because it is operating slowly and might be under heavy load). Thus, the query plan will be altered on-the-fly in an attempt to decrease processing time. The monitor also gathers statistics such as the type of DBMS at a local site, subquery type, and actual and estimated subquery completion times. This information is used in future optimizations to make better site choices.

  Our algorithm learns from subqueries performed at local sites in much the same way as the subquery execution monitor does. However, our algorithm captures and maintains different statistics on data, and does not alter the query plan in any way.

- *Piggyback Statistics Collection*: Piggyback collection requests additional data from sites during query execution in order to calculate statistics [34].

  As with piggyback collection, the data returned by queries will be used to calculate statistics, and statistics will be stored for use in future optimizations. However, FLO does not request any additional column or row data be returned with a query. This avoids the increased overhead incurred with such requests (see the section on piggyback statistics collection for more information).

FLO is implemented at the source wrapper level and can thus be used in parallel across sites. The statistics gathered by FLO can be used as input to any cost-based optimizer.

3.3 Collection

FLO captures data returned from queries performed at a local site. Data is then stored in a temporary table at the FDBS for analysis. Once the table has been created, analysis begins offline. FLO calculates counts, maximum and minimum values, histograms, etc. as described below. The statistics are stored in a catalog file for use by the federated optimizer. When queries are rerun, the optimizer has direct access to the statistics that have been extracted from previous queries performed on the same relations. No data is retained by the FDBS. Once analysis is complete, temporary tables are deleted.

It should be obvious that statistics may be incomplete at the time a query is run (i.e. a histogram might be used, but might only have data for entries over a subset of the range required). In this case, FLO cannot aid in the optimization of the present request; it is upon subsequent execution that FLO will provide performance gains.

It should also be obvious that statistics may become stale very quickly. Thus, as FLO is operating, it will periodically analyze data returned by a query to update statistics. When a statistic is recalculated, the new value is compared with the old value. The difference between values (i.e. Δ) is used to decide when the next recalculation will take place. If FLO determines that its estimates are very inaccurate, the frequency with which statistics are recalculated is increased. Conversely, if FLO determines that its estimates are fairly accurate, the frequency with which statistics are recalculated is decreased. In this manner, the changing state of the database is captured but statistics are not calculated each time a query is run (unless statistics have not been gathered previously, or are changing significantly between successive queries). So, from Table 1, a statistic that has changed by 10% will be recalculated again in 2 hours. Thus, the closer the value of the old statistic to the new statistic, the more accurate it is, meaning it must be recalculated less often.

3.4 Gathering Base Statistics

When a wrapper is first created, initialized statistics will be empty. Thus, some method of determining initial statistics is necessary. FLO supports four different methods of loading initial statistics:

Statistic Δ	*Next Recalculation*
≥ 20 %	At Next Chance (i.e. when new/useful data is available)
≥ 10 %	2 hrs.
≥ 5 %	6 hrs.
≥ 2 %	12 hrs.
≥ 1 %	24 hrs.
< 1 %	48 hrs.

Table 1: Recalculation time by change in statistic value

1. *Entered by Global DBA*: The global database administrator can enter the initial statistics directly. This method is somewhat tedious and prone to human error, but does not impose any additional overhead or loss of performance at the local sites or FDBS.

2. *Probe Upon Wrapper Creation*: The wrapper can run probing queries to get initial statistics when it is created. In this manner, the initial statistics will be completely accurate at the time of wrapper creation. However, this may impose overhead and decrease performance at the local sites.

3. *Probe Upon First Use*: The wrapper can run probing queries to get initial statistics when the statistic is first used. In this manner, the initial statistics will be accurate at the time the first query is submitted. However, this will impose overhead on the first query to be executed that uses a particular statistic, as well as possible decreases in performance at the local sites. In addition, this operation would be blocking, as we need the statistic before we can optimize the query.

4. *Learn from Scratch*: The wrapper can learn statistics from scratch (i.e. the initial statistics are learned). In this manner, the first few queries to be run will essentially run unoptimized. This is because no statistics will be available to perform global optimization. However, no additional overhead or decreases in performance will take place. Further, the value of the statistic might take some time to align with the actual statistic value, meaning that many queries could potentially run poorly optimized.

3.5 Calculating Statistics

As mentioned above, FLO calculates table counts, column maximum and minimum values, histograms, number of distinct values, and round trip time for each relation. A discussion of how FLO calculates each of these values is presented in this section.

3.5.1 Round Trip Time (RTT)

Figure 1 presents an algorithm that can be run constantly during query execution. Calculations are based on the RFC 793 estimation algorithm of round trip times for Transmission Control

```
$RTT_N$ = RTT returned from ping of site S
$RTT_S$ = The RTT stored for site S
if($RTT_S$ is empty)
        $RTT_S = RTT_N$
else
        $RTT_S = RTT_N/8 + 7*RTT_S/8$
```

Figure 1: Algorithm for calculation of round trip time

```
1.   If the user requests a statistic unfiltered from one relation; take the statistic directly
     from the query result
2.   If the user requests a statistic filtered from one relation
2a.  & it is filtered on one predicate column; use the histogram and result set to calculate
     a new count = (result + number of tuples in the histogram of the predicate column
     outside the predicate range) / 8 + 7*(old count) / 8
2b.  & it is filtered on multiple predicate columns; use the selectivity estimate to revise
     the count = (result + (1-selectivity %)*(old count)) / 8 + 7*(old count) / 8
3.   If the user requests data unfiltered from one relation; count the result set
4.   If the user requests data filtered from one relation
4a.  & it is filtered on one predicate column; use the histogram and result set to calculate
     a new count = (count(result) + number of tuples in the histogram of the predicate
     column outside the predicate range) / 8 + 7*(old count) / 8
4b.  & it is filtered on multiple predicate columns; use the selectivity estimate to revise
     the count = (count(result) + (1-selectivity %)*(old count)) / 8 + 7*(old count) / 8
```

Figure 2: Algorithm for calculation of table count (cardinality)

Protocol congestion avoidance [15, 16]. A decay formula is used for calculation. This decay formula is used as the basis for many of FLO's statistics calculations. Round trip time is useful to determine transmission time to local sites.

3.5.2 Table Count

Figure 2 presents an algorithm for determining the number of tuples in a table (i.e. table cardinality). In this algorithm, the query result set is used to calculate cardinalities. However, cardinalities can only be calculated from data that has not been joined. If data is joined, we can only calculate cardinalities for the tables if we can assume a 1:1 relationship between tuples in the primary and foreign key tables. This assumption rarely holds.

3.5.3 Maximum and Minimum Column Values

Figure 3 presents an algorithm for calculating the maximum and minimum values in a column. In the algorithm, the query result set is used to calculate maximum and minimum values. In

1. If the user requests a statistic unfiltered from one relation; replace the old max or min with the query result
2. If the user requests a filtered statistic from one relation;
   -If old max or min was not filtered by the query, we can replace them with new values
   -Else if filter is >; we can replace max always, but only replace min if we have a min that is lower than the current min;
   -Else if filter is <; we can replace min always, but only replace max if we have a max that is higher than the current max;
   -Else if range query; we can only replace max or min if we have a value higher or lower (respectively) than the current max or min
3. If the user requests data unfiltered from one relation; replace the old max or min with a new max or min for all columns in the result set
4. If the user requests filtered data from one relation;
   -If old max or min was not filtered by the query, we can replace them with new values;
   -Else if filter is >; we can replace max always, but only replace min if we have a min that is lower than the current min;
   -Else if filter is <; we can replace min always, but only replace max if we have a max that is higher than the current max;
   -Else if range query; we can only replace max or min if we have a value higher or lower (respectively) than the current max or min
5. If the user requests an (un)filtered statistic from a joined relation; use case 2
6. If the user requests (un)filtered data from a joined relation; use case 4

Figure 3: Algorithm for calculation of column maximum and minimum values

```
1.  If the user requests a statistic unfiltered from one relation; take the statistic directly from the query result
2.  If the user requests a statistic filtered from one relation
2a.     & the statistic is filtered on one predicate column; use the histogram and result set to calculate a new distinct = (result + number of distinct in the histogram of the predicate column outside the predicate range) / 8 + 7*(old distinct) / 8
2b.     & the statistic is filtered on multiple predicate columns; use the selectivity estimate to revise the distinct = (result + (1-selectivity %)*(old distinct)) / 8 + 7*(old distinct) / 8
3.  If the user requests data unfiltered from one relation; count the distinct tuples in the result set for each returned column
4.  If the user requests data filtered from one relation
4a.     & the data is filtered on one predicate column; use the histogram and result set to calculate a new distinct = (distinct(result) + number of distinct tuples in the histogram of the predicate column outside the predicate range) / 8 + 7*(old distinct) / 8 for each returned column
4b.     & the data is filtered on multiple predicate columns; use the selectivity estimate to revise the distinct = (distinct(result) + (1-selectivity %)*(old count)) / 8 + 7*(old count) / 8 for each returned column
```

Figure 4: Algorithm for calculation of the number of distinct column values

case 2, 4 - 6, we only replace the values if we have a new maximum or minimum, or the old values were not filtered by a predicate. This is because the old value might still exist in the database, but could have been filtered out by the predicates. However, we can always replace the values if we have a new maximum or minimum, as the old value is definitely lower than the new globally maximum or minimum value.

3.5.4 Number of Distinct Column Values

Figure 4 shows the algorithm used to calculate the number of distinct column values. In the algorithm, the query result set is used to calculate the number of distinct values. However, the number of distinct values can only be calculated from data that has not been joined. If data is joined, we can only calculate the number of distinct values for the columns if we can assume a 1:1 relationship between tuples in the primary and foreign key tables. This assumption rarely holds.

3.5.5 Histogram of Satisfying and Distinct Values

Figure 5 shows the algorithm used to calculate a histogram of satisfying and distinct tuple values. In the algorithm, the query result set is used to calculate the value of each bucket. However, the values can only be calculated from data that has not been joined. If data is joined, we can only calculate the number of distinct values for the columns if we can assume a 1:1 relationship between tuples in the primary and foreign key tables. This assumption rarely

1. If the user requests a statistic (un)filtered from one relation; we cannot use statistics unless we have requested a count or distinct count grouped by the bucket size we have set on a column
2. If the user requests data unfiltered from one relation; update the histograms for each column returned as bucket count or distinct = (new value)/8 + 7*(old value)/8
3. If the user requests data filtered from one relation on one predicate column; update the buckets that are inside the predicate range (in the predicate column only) as is done in 2 (we can't make any assumptions about the spread of data in other columns, i.e. we might have filtered out an entire bucket of values, or a large portion of values from a bucket in a non-predicate column

Figure 5: Algorithm for calculation of a histogram of distinct and satisfying tuples

-If table count < 20 then $b = \lfloor \text{count} / 5 \rfloor$
-Else if table count < 1000 then $b = \lfloor \text{count} / 10 \rfloor$
-Else if table count ≥ 1000 then $b = 100$

Figure 6: Algorithm for calculation of the initial number of buckets

holds. As well, data filtered on multiple predicate columns cannot be used for analysis because we can make no assumptions about the accuracy of the numbers we will calculate from the data that is returned.

There are three additional issues to discuss surrounding histogram generation and maintenance:

1. *Initial number of buckets*: We must decide how many buckets, b, to generate when the histogram algorithm runs on a column for the first time. The algorithm presented in Figure 6 is used to determine the number of buckets used. The reasoning behind this algorithm is that we want at most 20% of the tuple values to be within one bucket. As well, to make calculations faster, we want no more than 100 buckets.

2. *Increasing the number of buckets*: If any bucket has a frequency greater than 20%, double b and split each bucket in half (i.e. assume uniform distribution within each bucket and allow the values to become more realistic over time as we analyze more query data). Again, we want no more than 20% of the data values within one bucket.

3. *Calculating the bucket size*: The bucket size, s = (column max - column min) / b. This ensures that each bucket has an even portion of the spread of column values (i.e. we are generating an equi-width histogram).

3.6 Experiments and Research Questions

There are several questions we wish to answer using our proposed model:

1. *Does FLO impose any overhead at the federation?* In order to answer this question, queries will be posed over TPC benchmark data with statistics gathering turned on and turned off. Query response times will be compared statistically to see if any significant difference in response times can be attributed to FLO.

2. *Does the decay formula used by FLO to calculate the statistics reduce variances experienced with simple value replacement?* Some data sources experience variance in values returned by queries. For example, streaming network data will vary depending on network load, etc. Thus, we wish to smooth this variance seen in this fluctuating data over time, as well as produce a protocol that will provide accurate statistics in environments where data changes less frequently (i.e. a company database, etc.). In order to answer this question, queries will be posed over TPC benchmark data. Statistics will be computed using direct calculation after each query performed, as well as using the FLO technique. Statistics will be graphed to determine what slight variations in statistic values FLO can smooth out.

3. *Are statistics computed by FLO actually useful/accurate?* The authors of LEO claim that the FLO method of statistics collection was not useful because it makes assumptions about distribution of unknown data and distribution of error within data. However, we are not convinced that this claim is true. We wish to determine if the FLO method collects useful statistics. To answer this question, queries will be posed over TPC benchmark data. Statistics will be computed using direct calculation and again using FLO based on the collection schedule that FLO sets out. Statistics will be graphed to determine whether or not the claims made in [26] are true.

4. *How long does FLO take to align to correct statistic values?* We wish to have FLO align to the correct statistic value as quickly as possible. It would be useful to know approximately how many queries must be run before this alignment takes place[2]. The maximum, minimum and average alignment times, in number of queries performed before alignment within 2%, will be calculated using a set of queries over TPC benchmark data with statistics determined empirically and using the FLO method.

5. *Are refinements to the algorithm possible?* Is there any way that the algorithm can be made better? Can we calculate any reasonably accurate statistics from joined data? Running FLO on joined result sets in an attempt to extract statistics will help us answer this question. If statistics gathered from join data are relatively useful, perhaps our algorithm can be refined

6. *How often must the collector run in order to maintain accurate statistics?* In order to make sure our statistics are always as accurate as possible, we must ensure that a "good" schedule is determined for how often FLO must be run on user query data. We wish to balance the amount of computing resources given to FLO with the accuracy of statistics generated. Using TPC benchmark data, we will investigate how stable the relations

[2]We recognize that this is entirely implementation and data specific. Although an exact value for all situations cannot be determined, it should be possible to determine a reasonable approximation

are, and attempt to analyze how often FLO needs to run to keep statistics within each accuracy ranges from Table 1.

7. *How long does the algorithm take to run?* It is important to determine the amount of processing required to calculate statistics from the data so that we may understand the requirements FLO imposes on hardware and software implementations. To answer this question, statistics will be calculated on data sets of varying sizes from the TPC benchmarks in order to determine the average, maximum and minimum run times for the algorithm. All run times will be plotted in order to graphically organize the data to determine any alarming trends in run times.

We believe the experiments and questions outlined in this section will allow us to determine the usefulness of this algorithm. We also expect many more questions to present themselves during implementation and evaluation of the algorithm.

3.7 Implementation Proposal

FLO would be implemented at the source wrapper level of a federated database system. Thus, FLO would be able to calculate statistics in parallel across sites. FLO would require minimal code to be written in order to extend current wrapper implementations to calculate statistics. FLO could be implemented to intercept data as it is passed from the wrapper to the federated system. Then, FLO could cache the data and run its collection algorithm.

In this section, we have outlined a complete statistics collector that relies only on user queries for data collection. We believe our algorithm imposes minimal additional overhead to user queries and the system as a whole during analysis. Some experiments and unanswered research questions have also been presented. Due to time constraints, it was not possible to implement our algorithm and test it.

4 Optimization Techniques

Many query optimization techniques for FDBS's that rely on statistics have been proposed in the literature [1, 2, 4, 8, 10, 11, 18, 25, 28, 30, 31, 33, 37]. This section provides an overview of how FLO can provide statistics for these algorithms.

4.1 Parallel Reduction Algorithms

Some algorithms process data in parallel at local sites. This data is also reduced at those sites before being transferred for final assembly [2].

4.1.1 Semijoin Optimization Algorithm

The semijoin algorithm was proposed by Brill et al [2] and assumes that the cost of data transfer through a network outweighs local site CPU overhead. Therefore, this algorithm seeks to reduce the size of relations required for a query at local sites before transferring data back to the FDBS for query execution. It consists of four steps:

1. Site Selection - A set of sites that will be used to perform a query must first be chosen. This requires finding a set of minimal size that includes one copy of each local, partitioned and replicated relation (i.e. each site holding a data fragment must be in the set, but only one replica of a relation must be in the set). Some sites may hold more than one relation required by the query, allowing us to further reduce the size of the site set.

2. Local Reduction - In parallel at each local site in the chosen site set, reduce each relation by performing selections and projections. Parameters used to perform these operations are taken from select, where and join conditions in the original query[3].

3. Global Reduction - Find and execute an efficient sequence of semijoins that will reduce the set of records to be transmitted. The original proposal uses a hill-climbing algorithm to determine this set. Once the semijoins are performed, the smallest amount of data required to answer the query is ready for transport.

4. Assembly - Transfer the data to one central query site and generate the result set. Return the result set to the user[4].

This algorithm exploits the capabilities of the DBMS's in the federation, and attempts to reduce the transmission overhead required to send data between sites. However, statistics are computed on-the-fly and are discarded once the query is complete. No attempt is made to store or make statistics more accurate. So, the same computation errors or false assumptions may be made each time statistics are computed. FLO seeks to make statistics more accurate over time through constant refinement and recalculation.

4.1.2 Replicate Optimization Algorithm

The replicate algorithm was also proposed by Brill et al [2] but assumes that CPU overhead at local sites outweighs transfer costs between them. Therefore, this algorithm seeks to transfer data to local sites in order to exploit the differences in processing speeds of each system. It consists of four steps:

1. Site Selection - As with the semijoin algorithm, we choose a minimal site set. However, instead of choosing only one replica for each replicated relation, we include all replicas of the data. This allows us to run queries in parallel at each replica.

2. Data Transfer - Copy each relation to each site where it is to be used to process a subquery, but does not already exist (i.e. if site 1 holds relation A and requires relation B, transfer B to site 1). This may require composing fragmented relations into one large relation. After this step, each site should have a copy of the relations that are to be used to form the partial query result for which that site is responsible (as per the subquery sent to it by the controller).

[3]It might be possible to optimize the order in which site reduction queries are performed by exploiting network traffic and speed, CPU load at local sites, etc. (i.e. submit queries over slow links or to slow sites first in an attempt to minimize their impact on overall execution time).

[4]It may be less costly to generate the result set at one central site and then transfer the data back to the user. It may also be less costly to assemble the result set at the user's site.

3. Query Execution - Once each site has the data it needs to run its partial query, the queries are executed. After this step, each site should have a partial answer to the user's query.

4. Assembly - Transfer the partial answers from local sites and create the final result set at the user's home site. Return the results to the user.

This algorithm exploits the varying hardware capabilities of the DBMS's in the federation, and attempts to reduce response time to a query by performing it in parallel at different sites. The two main differences between the semijoin and replicate algorithms are:

1. *Assumptions*: depending on which assumptions are valid at execution time, either algorithm could be used.

2. *Execution Location*: replicate queries are executed at local sites in parallel while semijoin queries are executed at the FDBS (or another central site) once all data has been reduced.

As with the semijoin algorithm, no attempt to store or increase the accuracy of statistics for future use is made, and FLO seeks to improve on this.

4.2 Single-Run Algorithms

Some algorithms are run only once. The cheapest or most reduced plan that is executed at the end of the algorithm is not altered after compilation or in-flight to adjust to runtime parameters or other changes in system state [4, 8, 10, 11, 18, 25, 28, 33, 37].

4.2.1 Semantic Query Optimization

Semantic Query Optimization uses database integrity constraints to optimize queries. Using the global schema and some heuristic transformation rules, a query is converted into a semantically equivalent form that has a lower total processing cost. This can be done through communication with agents running at local sites that can take into account local site parameters [4, 28].

While this method is useful, one cannot assume that local sites will allow agents to be installed to gather statistics such as CPU load, memory, indexing information, etc. Thus, another method of gathering needed parameters must be found in order to make this optimization algorithm useful in all federated environments. FLO provides that alternative.

4.2.2 Statistical Sampling Optimization

Using statistical sampling optimization, plans in the plan space are classified according to the relations they access and the operations they perform [33, 37]. The reason for classification is that queries with similar access plans have similar costs associated with them. In this manner, it is possible to take a statistical sample from each class in the plan space, estimate an average cost for that class, and perform a query from the class with the lowest cost.

As with many other optimization algorithms, statistics are not saved during optimization in an attempt to make optimization of future queries faster and more accurate. Thus, much of the overhead of this method can be saved using FLO to gather statistics at runtime.

4.2.3 The Garlic Optimizer

The Garlic optimizer uses wrappers to gather cost information for developing a query plan [8, 10, 11, 18, 25]. Wrappers that communicate subqueries to the local sites provide costs for those subqueries to the optimizer. The goal is to allow Garlic to find a good plan without knowledge of the capabilities of local sites (i.e. the wrappers must know the capabilities of sites and provide a good cost estimate for any given subquery). The Garlic optimizer uses a set of strategy alternative rules (STAR's) that are used to generate and rewrite plans. Plans consist of a set of plan operators (POP's) that compose the query plan tree (sort, filter, scan, etc). A generic "pushdown" POP that encapsulates work to be done at a local site (i.e. table scans, etc) is also included.

STAR's are "fired" over the query string in order to generate POP's. Thus, STAR's can be seen as grammatical production rules. STAR's generate cost and cardinality information using input from wrappers. The Garlic optimizer works in three phases:

1. Fire Access STAR's - "Access" STAR's are applied in order to enumerate plans that read data from a source. The plan space is then pruned in order to remove plans that have the same or weaker cost properties (i.e. remove plans of higher cost that might not provide some interesting order to future joins, etc.).

2. Fire Join STAR's - "Join" STAR's are applied in order to enumerate all plans involving joins. The plan space is then filled with all possible join orders. Garlic considers bushy plans and left-deep plans. Bushy plans are considered because collocated data may make a bushy plan more efficient (i.e. a join could be performed cheaply at a local site instead of sending the data back to the FDBS for joining). The plan space is then pruned in order to remove those plans that have the same or weaker cost properties.

3. Fire FinishRoot STAR - The "FinishRoot" STAR is applied to provide orderings, selects, projects, etc. that were not already completed by some local site (i.e. the local site did not have the proper capabilities or sufficient data to perform that operation). The plan with the lowest cost is then chosen for execution.

4.3 Multi-Run Algorithms

Some algorithms are run at several points before and during query execution. These algorithms can adapt a compiled query plan just before or even during execution if some statistics have changed significantly or some unforeseeable circumstances present themselves [1, 30, 31].

4.3.1 2-Phase Optimization

Zhu [31] proposed optimization be performed in order to exploit cost information available or calculable only at runtime. The algorithm consists of two phases of optimization as follows:

1. Compile-time Optimization - Perform as much optimization as possible with the information that is available at compile time. This optimized, but possibly incomplete, plan is stored in the system. Plans might be incomplete if, for example, a query requires a runtime parameter to execute.

2. Run-time Optimization - Perform remaining optimization or re-optimization when the stored plan is executed. Re-optimization might be necessary if, for example, the statistics used to calculate the phase 1 optimization are significantly out of date.

This method, while quite applicable, requires some statistics collection methods to be used. However, no mention of what collection methods are appropriate is made. FLO can be used to provide relatively accurate statistics to both phases of this algorithm that model the changing state of the data in the federation.

4.3.2 Adaptive Query Optimization

Adaptive query optimization makes use of statistics during execution to change a query plan in-flight [1, 30]. A plan is altered to make it more optimal if some calculations appear to be incorrect as intermediate results are retrieved. Thus, a plan that appeared to be optimal may be discarded during execution for another plan based on some runtime information (ex. a server goes down, high load at some site, etc.).

Although this method makes use of statistical information, as with the semijoin algorithm, it does not save these statistics. Thus, the resources for computing or capturing statistics are wasted and future queries will not benefit. As stated above, FLO's learning allows it to incrementally improve statistics throughout the life of the federation.

Garlic makes use of wrappers to gather statistics. They are assumed able to gather statistics directly from the local sites (from catalogs or base tables, etc.). However, this might not be possible in all situations. For example, an XML file does not contain a catalog. FLO provides a method of calculating statistics even in environments where the wrapper has no direct access to catalogs or data tables.

5 Statistics Collection Methods

Cost-based query optimizers rely quite heavily on statistics in order to determine which plan is most appropriate for execution. As such, several methods have been proposed for collecting or calculating statistics in a federated system [12, 17, 21, 23, 24, 29, 32, 34, 35, 36]. This section provides a discussion of these algorithms as well as details of their deficiencies. Ways in which FLO improves on these algorithms will be presented.

5.1 Calculation Methods

Some statistics collection methods perform complicated calculations or build complex models in order to calculate statistics and rely heavily on base statistics. They may not generate the base statistics themselves, but their calculations are not based on the data they summarize [23, 32, 35]

5.1.1 Fuzzy Cost Estimation

Fuzzy cost estimation makes use of a fuzzy cost model in order to make predictions about statistics [23, 35]. A fuzzy cost model assigns probabilities to parameter values. Then, the final estimated statistic value will be estimated as a weighted average, by probability, of each of the possible values that make up the fuzzy model. The model is built by "experts" (people knowledgeable about the field) who program it into the optimizer. A model is provided for each parameter to be used in cost estimation.

This method requires models be computed and built offline, and then programmed into the optimizer. Also, the model will need to be updated regularly by hand to capture the changing state of the database. This requires an expert to perform a significant amount of additional work. FLO does not require an expert.

5.1.2 Uniform Selectivity Estimation

Zhu [32] presented a method of calculating selectivity estimates for columns based on a number of formulas. These formulas take into account the number of distinct values, record counts, etc. The formulas can be used only in situations where the data is uniformly distributed over the predicate columns.

This method of calculating selectivity estimates is quite useful. However, it assumes data uniformity. If data is uniformly distributed, selectivity estimates will be accurate. FLO can provide the input statistics necessary to calculate selectivities with these formulas. Thus, the two can be seen as complimentary.

5.1.3 Trapezoid Selectivity Estimation

Also presented by Zhu [32] is a method of calculating selectivity estimates for columns using the trapezoid integration rule. In this method, a histogram can be used to generate a cost formula. This formula is then integrated and evaluated in the range required by the predicate to determine the selectivity of that column.

This method of calculating selectivity estimates is quite good as it does not assume data uniformity[5], and can be computed quickly using available information about a column's values. Again, FLO can provide the histogram inputs to this method.

5.2 Data Sampling Techniques

Several methods to generate statistics through sampling of data stored in base tables have been proposed [12, 21, 24, 36].

5.2.1 Sequential Sampling

Haas and Swami [12] presented a sampling method where tuples are divided into groups (each group may contain only one tuple). Then, random sampling is performed within groups (i.e. simple or stratified random sampling), and statistics and error estimates are calculated on-the-fly. The algorithm stops drawing samples when the error estimate is below some confidence level.

Although this method may compute a statistic meeting some confidence level, it is possible that many more samples than required for a statistically significant sample size are drawn.

[5]In fact, no data shape is assumed

Conversely, it is also possible that too few samples will be drawn in order to calculate an accurate statistic. This is because the calculation of error is only an estimate.

5.2.2 Adaptive Sampling

Lipton et al [17] proposed a sampling method where a pre-selected number of samples is drawn from the tuple space in order to compute statistics. In this manner, a statistically significant sample can be guaranteed using statistical theory to choose the number of samples to be drawn, or the application can provide a base error level and the system will compute statistics to have an accuracy above that error level. Thus, the sample size is adapted to meet an application-required confidence level, or to consider the changing size of relations.

Adaptive sampling avoids the problems associated with sequential sampling, but suffers from a widely varying overhead (i.e. the overhead required to calculate statistics to an application-defined confidence level will change significantly from table to table). Because of this, it is difficult to make accurate predictions of the time it will take to perform the analysis required to generate the statistics. Again, FLO does not incur extra overhead.

5.2.3 Systematic Sampling

Ngu et al [21] detailed a sampling method where a specific percentage (application specific or hard coded) of tuples in a table are sampled for statistics computation. The algorithm then divides the table into n logical bins, where n is the number of tuples to be sampled. One tuple is sampled at random from each bin. Thus, the algorithm attains a sample of tuples that covers the entire data file. Statistics can then be computed on the drawn sample, and confidence levels can be given.

This algorithm, at first glance, appears to take a more accurate sample of tuples from the table. However, it is possible to receive a biased sample if data in the table is sorted. For example, if the algorithm chooses a tuple from the beginning of bin i and the end of bin $i+1$, it is possible that because of sorting, the tuples at the end of bin i and the beginning of bin $i+1$ are of a different value and are missed by the algorithm. Also, the overhead imposed might be similar to a full table scan as each sample could potentially come from a different disk page. The statistics calculated by FLO do not suffer from sample bias as they are calculated on full data sets returned by a user query.

5.2.4 Query Class Sampling

Zhu and Larson [36] outlined a sampling method based on running probing queries from query classes. Queries are grouped according to the "class" of functions they perform. Some random queries are generated in each class and run across the data. The time taken to perform each query is recorded. Then, regression models of the observed query execution times are formed in an attempt to fit a general cost model for each query class. Statistics are extracted from the regression models. If required, statistics extracted from the regression model could be adjusted up or down (using transformations) so that they more closely fit the regression model.

Although this algorithm appears to provide solid timing estimates, it relies on knowledge of the indices and access methods used by a local site. This information might not always be available to the FDBS. Also, it is possible that a regression model might be inaccurate if the sample is taken at a time of unusually high or low local site load.

5.2.5 Shifting Method

Rahal et al [24] proposed a method to calculate (and update) statistics using regression techniques. In this case, old values are shifted out of a cost model and new values take their place in a matrix that is later used to perform regression analysis. Thus, statistics are be evolved as new samples arrive.

This method has two major flaws. First, the algorithm runs in $\Theta(n^2)$ time (where n is the degree of the regression equation). This makes it nearly impossible to perform online if many queries are being processed at once. Second, the calculated statistics might not be accurate. Regression analysis can introduce a significant amount of error, especially if the samples experience large variance. Thus, special care must be taken to ensure that an appropriately large number of samples are included. FLO does not rely on regression in order to calculate statistics. Thus, it avoids the overhead and error involved with this technique.

5.2.6 Block-moving Method

Rahal et al [24] also proposed a method to recalculate statistics for a group of new samples instead of a single new sample. In this case, the old statistics are shifted out of the model in "blocks". Regression analysis is then performed on the new matrix containing the new values. The new statistics can then be used for optimization.

This method is more appropriate for use in environments where statistics are changing slowly (i.e. the old statistic does not deviate significantly from the actual value). Thus, either method (shifting or block-moving) can be used depending on the environment the system is running in. However, the block-moving method still shares the two flaws seen in the shifting method.

In all sampling methods, regression analysis is performed in order to calculate the statistics from the observations. Sequential sampling draws samples until the desired confidence level is attained. Adaptive sampling draws a predetermined number of samples to attain some user specified (and possibly shifting) confidence level. Thus, sequential sampling may draw fewer samples than required to attain a confidence level as dictated by statistical theory, where as adaptive sampling is always guaranteed to draw a statistically significant sample size as dictated by statistical theory.

Although sampling is mainly applied to selectivity estimation (except in the case of query class sampling, which is used to determine the time taken to access data at remote sites using indexes, etc), the above methods may be extended quite easily to determine such things as the number of distinct column values, maximum and minimum column values, histograms, etc. The overriding problem with sampling methods is that they require direct access to the data tables in order to perform efficient sampling. In the case of a federation, the statistics generator will not have direct access to data tables because of local site autonomy. Queries will have to be run in order to gather sample data, and this can be slow and incur a large amount of overhead at local sites. As stated earlier, FLO avoids these problems.

5.3 Probing Methods

Some statistics gathering methods request data or statistics directly from a data source. These methods can be broadly classified as probing methods [29, 31, 34].

5.3.1 Probing Queries

Probing Query methods dispatch queries to local sites during the compilation of stored procedures [31]. These queries execute operations to gather necessary statistics for optimization. For example, a probing query may request the maximum and minimum column values for relations in a join operation.

Although this method is useful for optimization of compiled queries, it is not suitable for use with dynamic queries as significant additional overhead may be incurred to dispatch the necessary probes to local sites. In addition, the statistics gathered will become stale over time, requiring the query to be re-optimized regularly. FLO does not incur additional overhead to gather statistics as they are computed from user query result sets.

5.3.2 Utility-based Collection

Utilities that contact local sites and perform probing queries can be run by database administrators [29]. They can collect completely accurate statistics, but have the unfortunate side effect of high cost. For example, building a histogram on a relation could potentially require a query to ask for all values in a certain column. This may impose significant load on a local site and cause performance problems for local queries.

While this method will collect completely accurate statistics, it potentially imposes large amounts of overhead. For example, if the entire table must be locked for each probe, local queries may experience significant slowdowns. As well, a database administrator must run the utility at regular intervals to keep statistics up to date. Database administrators might not have time or will to perform such tasks. FLO does not require input from a database administrator, nor does it impose performance hits or additional overhead.

5.3.3 Piggyback Collection

Statistics collection operations can be included (piggybacked) with user submitted queries [34]. In this fashion, more data than is required to answer a user query is requested from the local site. This data is returned to the FDBS where statistics are calculated. The data is then altered to contain only the results that will answer the user's query. In this manner, a small additional overhead is incurred with the aim of avoiding plans of high cost in the future. In this fashion, very accurate statistics can be kept on remote data.

This method requires additional processing time be spent running queries in order to gather statistics. In some cases, adding an extra column could potentially double query execution time. Thus, the performance hit to user queries might be unacceptable. FLO does not request additional data from queries to calculate its statistics and hence imposes no performance hit on user queries.

All of the methods presented in this section have relied on the collection of additional data or imposition of additional overhead in order to calculate the statistics they provide. In many cases, this is not acceptable. The decrease in performance might not be tolerable by an application or a user. FLO alleviates this problem by calculating statistics from user query results. Thus, no extra work is required by database administrators, and no additional overhead or performance hits are introduced.

6 Conclusion

In this paper, we have presented FLO: a Federated Learning cOllector. FLO ascertains the value of statistics required for optimization from user queries. Thus, FLO does not impose any additional overhead on local sites, and imposes only minimal overhead at the FDBS for analysis. FLO imposes no performance penalties on user queries. Also given was a discussion of current proposals for statistics gathering and query optimization techniques. Ways that FLO differs from current statistics collection methods, and aids in optimization, were also presented.

In future work, we would like to implement our method and determine how well FLO performs. Several performance questions need to be answered. How accurate are the statistics calculated by FLO? How long does FLO take to align to correct statistic values? Are optimizations to the algorithm possible? How often must the collector run in order to maintain accurate statistics? How long does the algorithm take to run? These questions, among others, must be answered before the usefulness of FLO can be determined.

Further future work will include an extension to our idea in order to include the collection of other statistics such as network bandwidth, local site loads, etc. These statistics can be used in site selection and other optimization tasks. To our knowledge, no other statistics collector addresses these issues.

References

[1] P. Bodorik, J. Pyra, and J. S. Riordon. Correcting execution of distributed queries. In *Proceedings of the second international symposium on Databases in parallel and distributed systems*, pages 192–201. ACM Press, 1990.

[2] David Brill, Marjorie Templeton, and Clement T. Yu. Distributed query processing strategies in mermaid, a frontend to data management systems. In *Proceedings of the First International Conference on Data Engineering*, pages 211–218. IEEE Computer Society, 1984.

[3] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 263–274. ACM Press, 2002.

[4] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, 1990.

[5] Neil Coburn and Per-Ake Larson. Multidatabase services: issues and architectural design. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 57–66, Toronto, Ontario, Canada, 1992. IBM Press.

[6] The Transaction Processing Performance Council. Transaction processing performance council. Accessed November 23rd, 2004 from http://www.tpc.org/.

[7] John Grant, Witold Litwin, Nick Roussopoulos, and Timos Sellis. Query languages for relational multidatabases. *The VLDB Journal*, 2(2):153–172, 1993.

[8] L. M. Haas, P. Kodali, J. E. Rice, P. M. Schwarz, and W. C. Swope. Integrating life sciences data-with a little garlic. In *Proceedings of the 1st IEEE International Symposium on Bioinformatics and Biomedical Engineering*, page 5. IEEE Computer Society, 2000.

[9] L. M. Haas, E. T. Lin, and M. A. Roth. Data integration through database federation. *IBM Systems Journal*, 41(4):578–596, 2002.

[10] L. M. Haas, P. M. Schwarz, P. Kodali, E. Kotlar, J. E. Rice, and W. C. Swope. Discoverylink: a system for integrated access to life sciences data sources. *IBM Syst. J.*, 40(2):489–511, 2001.

[11] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 276–285. Morgan Kaufmann Publishers Inc., 1997.

[12] Peter J. Haas and Arun N. Swami. Sequential sampling procedures for query size estimation. *SIGMOD Rec.*, 21(2):341–350, 1992.

[13] David K. Hsiao. Federated databases and systems: part i — a tutorial on their data sharing. *The VLDB Journal*, 1(1):127–180, 1992.

[14] David K. Hsiao. Federated databases and systems: part ii — a tutorial on their resource consolidation. *The VLDB Journal*, 1(2):285–310, 1992.

[15] V. Jacobson. Congestion avoidance and control. In *Symposium proceedings on Communications architectures and protocols*, pages 314–329. ACM Press, 1988.

[16] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison Wesley, second edition edition, 2003.

[17] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 1–11. ACM Press, 1990.

[18] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 18–27. ACM Press, 1988.

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

[20] Hongjun Lu, Beng-Chin Ooi, and Cheng-Hian Goh. On global multidatabase query optimization. *SIGMOD Rec.*, 21(4):6–11, 1992.

[21] A. H. H. Ngu, B. Harangsri, and J. Shepherd. Query size estimation for joins using systematic sampling. *Distrib. Parallel Databases*, 15(3):237–275, 2004.

[22] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1999.

[23] Per-Ake Larson Qiang Zhu. Establishing a fuzzy cost model for query optimization in a multidatabase system. In *System Sciences, 1994. Vol.II: Software Technology, Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 2, pages 263–272, 1994.

[24] Amira Rahal, Qiang Zhu, and Per-Ake Larson. Evolutionary techniques for updating query cost models in a dynamic multidatabase environment. *The VLDB Journal*, 13(2):162–176, 2004.

[25] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *The VLDB Journal*, pages 599–610, 1999.

[26] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 19–28. Morgan Kaufmann Publishers Inc., 2001.

[27] Marjorie Templeton, Herbert Henley, Edward Maros, and Darrel J. Van Buer. Interviso: dealing with the complexity of federated database access. *The VLDB Journal*, 4(2):287–318, 1995.

[28] H. J. A. van Kuijk, F. H. E. Pijpers, and Peter M. G. Apers. Semantic query optimization in distributed databases. In *International Conference Proceedings of Advances in Computing and Information*, pages 295–303, May 1990.

[29] E. Whalen. *Oracle Performance Tuning and Optimization*. SAMS Publishing, 1996.

[30] C. T. Yu, L. Lilien, K. C. Guh, and M. Templeton. Adaptive techniques for distributed query optimization. In *IEEE 1986 International Conference on Data Engineering*, pages 86–93, 1986.

[31] Qiang Zhu. Query optimization in multidatabase systems. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 111–127. IBM Press, 1992.

[32] Qiang Zhu. An integrated method for estimating selectivities in a multidatabase system. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 832–847. IBM Press, 1993.

[33] Qiang Zhu. *Estimating Local Cost Parameters for Global Query Optimization in a Multidatabase System*. PhD thesis, University of Waterloo, 1995.

[34] Qiang Zhu, Brian Dunkel, Wing Lau, Suyun Chen, and Berni Schiefer. Piggyback statistics collection for query optimization: Towards a self-maintaining database management system. *The Computer Journal*, 47(2):221–244, March 2004.

[35] Qiang Zhu and P. A. Larson. Query optimization using fuzzy set theory for multidatabase systems. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 848–859, Toronto, Ontario, Canada, 1993. IBM Press.

[36] Qiang Zhu and Per-Ake Larson. A query sampling method of estimating local cost parameters in a multidatabase system. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 144–153. IEEE Computer Society, 1994.

[37] Qiang Zhu and Per-Ake Larson. Solving local cost estimation problem for global query optimization in multidatabase systems. *Distrib. Parallel Databases*, 6(4):373–421, 1998.