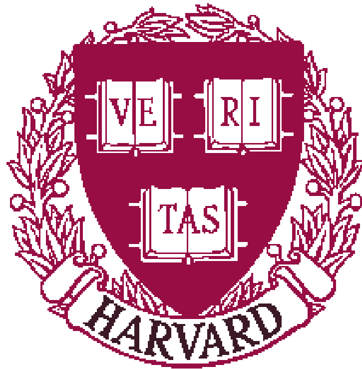
Feedback-Directed Query Optimization

Kim Hazelwood

TR-03-03

2003



Computer Science Group
Harvard University
Cambridge, Massachusetts

Feedback-Directed Query Optimization

Kim Hazelwood
Division of Engineering and Applied Sciences
Harvard University
hazelwood@eecs.harvard.edu

Abstract

Current database systems employ static heuristics for estimating the access time of a particular query. These heuristics are based on several parameters, such as relation size and number of tuples. Yet these parameters are only updated intermittently, and the heuristics themselves are hand-tuned. As trends in database systems aim toward self-tuning systems, we can apply the experience of the feedback-directed compiler world to provide robust, self-tuning query optimizers. This paper presents the design and evaluation of a feedback-directed query optimization infrastructure. Using trace-driven simulation, we conclude that dynamic feedback can be quite effective at improving the accuracy of a query optimizer, and adapting to predictable query overhead.

1 Introduction

The basic functionality of a database system includes support for data storage and retrieval. While a user is unambiguous about the data requested from the database, there are several alternatives available when it comes to retrieving the data internally. For instance, in complex queries, there is a certain amount of latitude when deciding the execution order of each of the sub-queries. In modern database systems, it is the role of the query optimizer to determine the best strategy for carrying out a query. Alternative strategies are evaluated and compared based on their expected cost [6], and the query plan with the lowest cost is executed.

Current query optimization systems use static heuristics for estimating the cost of various equivalent plans for evaluating queries. These heuristics calculate the average-case overhead of query evaluation. The expected cost is calculated using statistics about the database and the query being performed, to the extent that the statistics are known. Yet, in many cases, these average-case estimations may not be representative of the actual cost of performing the query. There are several reasons for this.

First, average cost may not be indicative of the cost of a particular query instance. For example, a binary search is expected to find a particular value in *log n* time, however a particular value may be found in as little as *time=1*. And while on average, a binary search will find a value faster than a linear scan, we can certainly envision the case where the value is located in the first slot of an array, thus it is better to use a linear scan. In fact, only rarely is an average case estimate equal the the actual result.

Second, query cost heuristics may miss a key bottleneck in the system, such as network and processor contention, or a system administrator may underestimate the cost of one or more bottleneck.

Third, many of the heuristics are based on dynamic database attributes, such as size and fanout, and although these values may change frequently, the values used in the heuristics are only updated during query downtime, if at all.

Finally, there are no sanity checks incorporated into the query estimation system to detect *bad* estimations. A feedback-directed system can not only recognize bad query cost estimates, but it can provide compensation factors to correct the estimates.

Dynamic feedback-directed query optimization attempts to bridge the gap between estimated and actual query times. By maintaining a list of actual query times acquired during execution, we can make more informed decisions on which query optimization to choose at a particular instance in time. Actual query times may be incorporated into a query optimizer in several ways. First, we can use the information as a tie-breaker for multiple query optimizations that appear to achieve similar benefits. Second, we can use actual times to invalidate query estimations that appear to be non-representative for a given execution. Third, we can even envision a system that always chooses past query performance as the estimator of future query performance, and

only resorts to the former method of estimation when there is no dynamic information available for a query.

In this paper, we explore a system that utilizes feedback from previous queries to guide future query optimization decisions. The internal design is based on similar research performed in the feedback-directed compilation world. The contributions of this paper are as follows:

- We present the design architecture of a feedback-directed query processing system.
- We describe some of the design choices of building a feedback-directed query processor.
- We present a query execution simulator for exploring the problem domain.
- We discuss the results achieved by simulation.

The remainder of the paper is organized as follows. Section 2 introduces query optimization, describes the existing query processor architecture, and provides the motivation for our work. Section 3 then describes the system extensions required for introducing feedback-directed cost profiling into a query processor, and the issues and design considerations we encountered. Section 4 introduces our query engine simulator and trace generator. Section 5 presents and discusses our results. Section 6 discusses related research, Section 7 concludes, and finally Section 8 discusses areas of future work.

2 Query Processing

A query processor has the responsibility of performing two main tasks–query optimization and query execution. Often, there are several alternative strategies for query execution, all of which produce the same result. Query optimization is the process of choosing the best strategy from the available options. Figure 1 shows a sample input query. As we can see, there are two alternative methods for performing the query, *option A* and *option B*. By estimating the *cost* of each of the two options, the query optimizer will decide which strategy should be performed.

Figure 2 shows a typical query processing engine. After a query has been submitted, parsing and analysis is performed in order to generate a set of equivalent plans for performing the data access. Each plan is sent through a cost estimator, where cost calculations are performed using knowledge about the database and the query being performed. Finally, the plan with the minimum cost is executed.

select balance
from account
where balance < 2500

option A: $\sigma_{balance<2500}(\Pi_{balance}(account))$
option B: $\Pi_{balance}(\sigma_{balance<2500}(account))$

Figure 1: Sample Query

Currently, the state-of-the-art in database query optimization involves the use of static heuristics for estimating query cost. These costs are used to guide decisions between equivalent options for performing a query. Query cost can be defined to describe several attributes of query overhead. Typically, cost is defined as one of the following:

- disk accesses
- cpu cycles
- network communication
- wall-clock response time

The costs are estimated using both a catalog of profile information about the database, and a set of weights for each profiled parameter [6]. In some systems, the database profile information is automatically gathered during query execution, and the catalog is updated periodically during query downtime. In other systems, it is the responsibility of the database administrator to update the catalog. For all systems, the set of *weights* for each profiled parameter is manually defined and tuned by a local database administrator.

The current procedure of manual parameter definition by a local administrator is problematic because (a) no sanity checks are currently in place to recognize or correct for bad decisions, and (b) in the case of a distributed database, a local administrator knows little about the attributes and costs associated with a remote database.

Trends in database systems are moving away from static heuristics, and toward adaptive, self-tuning systems. In Section 3, we describe simple extensions to the static query processor that will open the doors for automated query optimization.

3 Dynamic Feedback

In Figure 2, we depicted the static query processing engine that has been extensively described in

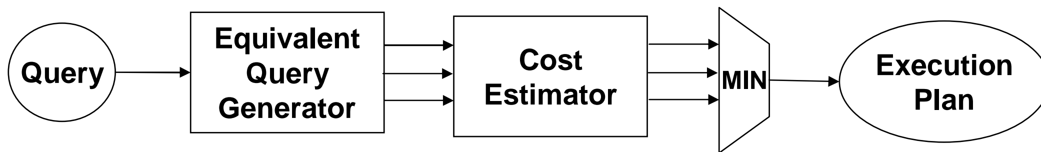


Figure 2: Static Query Processing Engine

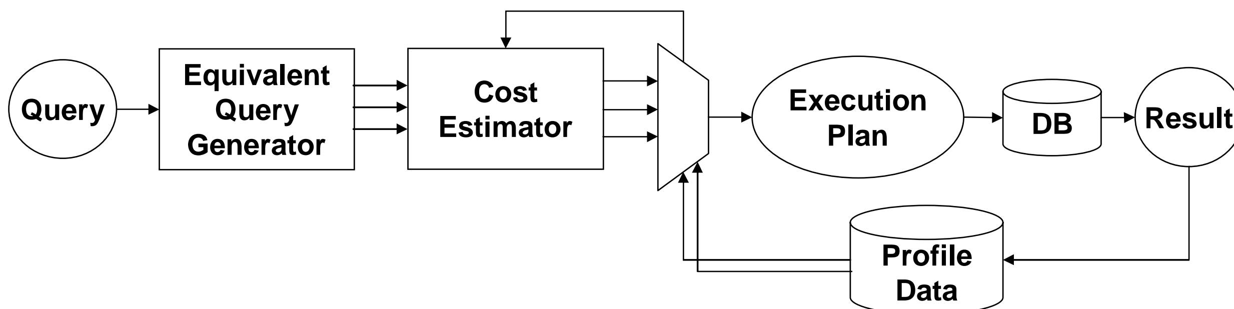


Figure 3: Query Optimization with Dynamic Feedback

literature. Over the years, a great deal of research has focused on the cost estimation portion of the query processor. Yet, however fine tuned the cost estimates may become, they are still just estimates. In fact, they are estimates of the average case, and their accuracy is highly dependent on the accuracy of the profile data and the heuristics in place. On the other hand, it is entirely possible to collect *actual cost* statistics from queries during execution, particularly if the cost is measured in response time. And in many cases, performing a lookup on the profile information is much more lightweight than recalculating the cost for each query.

In a query optimizer, dynamic feedback of the actual time it took to perform a query can be used in several ways, including:

- Providing assistance in future cost estimations for identical or similar queries.

- Signaling a problem with the current static heuristics. This signal may either notify a database administrator of a flaw in the current static cost heuristics, or trigger an automatic update of the cost heuristics.

Figure 3 depicts the design of a query processing engine that has been modified to incorporate dynamic feedback. In this system, the resulting cost of each query is stored into a profile database. This database is then queried during cost estimation. Finally, abnormalities in the cost estimations, and large divergences between estimated and actual query costs are reported back to the cost estimator. The following subsections describe each of the three major extensions in greater detail.

3.1 Profiling Cost Data

Several interesting design challenges arise as we extend the query processing engine to collect cost profiles. First, we must consider the problem of query matching. This concerns the statistical profile data stored in the profile database. For a given query, we must search the profile database, and decide what profile information can be considered applicable to the given query. Will we only allow exact query matches, or would it be possible to leverage the information from *partial query matches*?

Our solution to partial query matches is a hybrid approach. Complex queries are broken down into their respective *atomic sub-queries*. A query atom is a single database operation, such as a selection, projection, or join operation. Each query atom is stored separately in the profile database. During future cost estimations, only the profile information from exact atom matches will apply. Similarly, the profile information stored in the profile database will only be updated upon exact execution matches to each of the sub-query atoms. The notion of *similar* and *close-enough* matches are beyond the scope of

this paper. Yet it is important to point out that each of the query atoms can be combined to form a complex query. Therefore, while we may not have an exact match of a complex query in the database, we may combine several atomic sub-queries for a final cost estimation.

The next major challenge in extending the system to collect cost profiles is that the overhead of accessing the profile database must be considered. Therefore, it was necessary to design an efficient indexing scheme for storing and retrieving data from the profile database. Our current implementation is a multi-level hashing mechanism. We first index by the type of query (selection, projection, etc), then by attribute (account, balance, etc.)

A final note concerning the profile database is that there are no restrictions on the granularity of the collected data. Cost can be measured at the level of any or all of the standard cost measures listed in Section 2, i.e. disk accesses, cpu cycles, response time, etc.

3.2 Query Plan Selection

Another notable extension of the standard query processing engine shown in Figure 3 is the use of the profile data in the selection of an execution plan. While the previous system exclusively used static cost estimations, we now allow hybrid functionality in the query selection process by incorporating the profile data from previous query executions to guide our decisions.

The profiled cost statistics are used in the following manner:

- To break *ties* between queries with equivalent estimated costs
- To override the estimated cost of a query with the actual cost
- To recognize trends in cost estimation error

Each of these schemes for using the feedback from earlier runs to guide future cost estimations is investigated in Section 5.

3.3 Cost-Estimation Feedback

The final extension shown in Figure 3 is the feedback-loop into the Cost Estimator. Here, information from the profile database is compared to the static query cost estimations. Habitual errors in the cost estimation model trigger feedback to the cost estimator along the path shown in the figure. An automated system uses the error feedback in order to create an additional parameter to the cost estimation equation.

$$cost = cost + \Delta_{error} \tag{1}$$

We can view the Δ_{error} value as an auto-tune parameter that will compensate for an overlooked cost factor, or an out-of-date set of cost heuristics. Along with constant Δ_{error} offsets, we may also experience multiplicative offsets, where the entire cost equation must be increased. This may occur do to a slow microprocessor, or for memory-intensive queries, slow memory access time.

$$cost = cost * \Delta_{error} \tag{2}$$

A final case occurs when one cost factor isn't factored as heavily as it should, perhaps because of out-of-date profile information about the database itself. In this case, a more complex compensation factor is required. This compensation factor is applied to the offending input to the cost equation.

$$cost = costCalc(cf_1, cf_2, \ldots cf_n * \Delta_{error}) \tag{3}$$

In this section, we have described the feedback-directed query optimization architecture at a high level. Next, we will present the details of the simulator we used to evaluate our design.

4 Research Testbed

Building a complete system as described in Section 3 is beyond the scope of this paper. Instead, we evaluate our ideas using a query engine simulator. The simulator is a means for gathering initial data and determining whether it is worthwhile to build such functionality into a real system. Results from the simulator will provide insight into the conditions under which the system performs well, and often more importantly, the conditions under which it performs poorly. Both sets of information may serve as a guide when making design decisions for a final product.

4.1 Trace-Driven Simulation

The simulator we developed for investigating the responsiveness and adaptiveness of our system is depicted in Figure 4. The simulator models the behavior of two query cost estimators: our FDQO model and the standard non-FDQO model, then compares

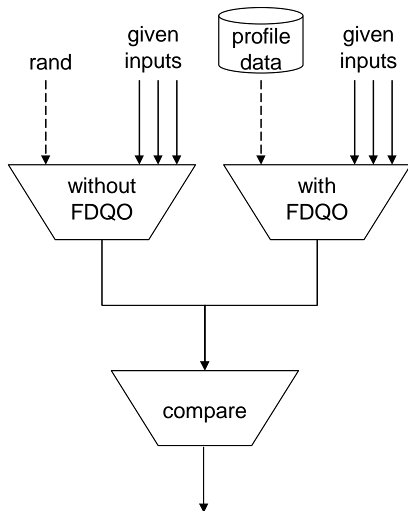


Figure 4: Feedback-Directed Query Optimization Simulator

these cost estimations with the actual cost of performing the query. (The actual cost is then fed back into the FDQO system to update its internal cost-estimation algorithms.) Several data points are collected during the cost estimation comparison process, such as the cost-estimation error and variance of each system.

In the top-left portion of Figure 4, we show the standard query processor without feedback-directed query optimization. This query processor takes several known parameters as inputs, and calculates the expected cost based on those parameters. To simulate reality, we add an additional randomized parameter that the query processor knows nothing about. The combination of all of the parameters results in the actual cost.

On the top-right, we show our feedback-directed query processor. Our query processor also isn't aware of the current value of the randomized parameter. However it is able to access the cost profile information that has been collected thus far to tune its cost calculations. We compare the cost estimates produced by both of the query processors to determine if feedback-directed optimization aids our query cost calculation accuracy.

The simulator is trace driven and is written in C. As it is a functional simulator, we do not model the overhead introduced by feedback-directed query optimization. While this can be considered a weakness in our model, we can argue that overhead values collected via simulation are not at all indicative of actual system overhead. In fact, even on a real system, the overhead can be altered via clever tuning. Therefore, we have not incorporated overhead calculations into our simulator as we view such calculations as dubious results.

4.2 Query Traces

As mentioned in Section 4.1, our simulator is trace driven. The query traces and input parameters that drive the simulator are generated by a query generation program. The query generator is responsible for creating the individual query details, along with the access patterns of the entire trace.

Each query in a query trace consists of numerous generated parameters. The first value represents the type of query. Our trace generator currently generates eight query types - linear search, binary search, selection, select+compare, sort, hash-join, projection, and aggregation. The query type is used by the cost estimator to determine the equation for the expected cost of the operation.

The second value generated for each query is a unique identifier that distinguishes similar and dissimilar queries. The remaining values are the required input parameters for the particular query cost estimation. The number and type of these cost parameters varies by query type, and can be inferred from Table 1 which lists the query cost calculations for each query type.

Finally, the trace generator creates the randomized cost parameter (described in Section 4.1) that is used to simulate cost parameters that are missing or under-weighted in the static cost equations. Each of the cost input parameters produced by the trace generator is used to estimate the cost of each query, while the final randomized parameter is reserved for use in calculating the *actual cost* of the query.

A typical generated query appears below. Each of the cost input parameters is defined in the caption of Table 1.

```
      lsearch id=1 br=50000 lambda=10
hashJoin id=2 br=4800 bs=235 p=402 lambda=8
selection id=3 hti=260 sc=631 fr=4 lambda=3
                    ⋮
```

The trace generator is written in C and communicates with the simulator via operating system pipes. It is designed to be parameterizable. For example, we can vary the number of distinct queries in a query sequence, the number of total queries, and even the odds of generating a particular query type. This allows us to study the amount of repetition necessary for our techniques to be most effective, the required warm-up period, and the individual impact

of feedback-directed query optimization on each individual query type.

The trace generator is pseudo-random and deterministic. The features of the `drand()` random number generating system call, which is used to generate the query input values, are such that the random-number sequence is always repeated during subsequent executions. This allows us to make apples-to-apples comparisons of different policies, and draw concrete conclusions about the effectiveness of one policy over another.

4.3 Query Cost Calculation

As discussed in Section 4.2, the query trace generator produces various queries for use by the simulator. The simulator then calculates the cost of each query. In this section, we discuss the cost equations used to estimate the expected and actual query cost.

Each of the generated query types and their respective static cost estimations are listed in Table 1. These cost estimations are widely accepted and described in further detail in [6]. In our simulations, the existing Non-FDQO model will estimate the query cost using the exact equations listed in Table 1. Our FDQO model, on the other hand, will use feedback from earlier query executions (if available) to adjust the query cost equations listed in the table.

Calculating the *actual cost* of a query (in order to compare the accuracy of the two competing models) forms the backbone of our results. As mentioned earlier, each query generated by the trace generator includes a randomized input parameter for simulating real-life contention in the database, which will result in inaccurate cost estimations. We shall refer to the randomized input parameter as the *lambda value*. An interesting research question is how to factor this lambda value into the cost equations to produce an *actual cost* equation. For this paper, we chose six distinct policies for factoring the lambda value into the actual cost equations. The six policies are listed in Table 2. Each policy is named to reflect the trace pattern it is attempting to simulate.

The first two policies, `additive` and `multiplicative` attempt to represent a stable overlooked cost factor. In this case, the actual cost equations become $[estimatedcost + lambda]$ and $[estimatedcost * lambda]$, respectively, where the lambda value does not change during subsequent executions of the same query. This policy is designed to test how well FDQO and Non-FDQO adapt to an overlooked cost factor in their cost equations.

Query Type	**Cost Calculation**
`linear search`	$\frac{b_r}{2}$
`binary search`	$log(b_r)$
`selection`	$HT_i + \frac{SC(A,r)}{f_r}$
`select+compare`	$HT_i + \frac{b_r}{2}$
`sort`	$b_r * \log_{M-1}(\frac{b_r}{M}) + 1$
`hash+join`	$3(b_r + b_s) + 4p_i$
`projection`	$V(A,r)$
`aggregation`	b_r

Table 1: Static cost calculations for each of our queries. b_r: blocks containing tuples of relation r. f_r: blocking factor of relation r. $V(A,r)$: distinct values in r for attribute A. $SC(A,r)$: selection cardinality of attribute A. HT_i: height of index i. M: page frames in memory buffer. p_i: partitions.

Lambda Policy	**Description**
`additive`	Constant overlooked factor
`multiplicative`	Constant overlooked multiplier
`randomized add`	Varying overlooked factor
`randomized mult`	Varying overlooked multiplier
`increasing`	Overlooked increasing trend
`value change`	Underestimated input factor

Table 2: This table describes the lambda policies simulated. Each item describes how the lambda cost factor is incorporated to calculate the actual query cost.

The third and fourth policies, `randomized add` and `randomized mult`, attempt to simulate overlooked cost factors that do not remain stable. More succinctly, these policies represent noise in the cost of a query. These policies are designed to test how FDQO and Non-FDQO respond to cost overhead noise. An ideal situation would be that neither policy is affected by noise, as it is not predictable and should not be incorporated into the cost equations.

The fifth policy, `increasing`, represents an overlooked cost factor which is not stable across queries, but it does follow a predictable pattern. The actual cost calculation is determined as $[estimatedcost + (previouslambda + 1)]$. An ideal system will recognize that there is an overlooked cost factor that is not noise, and should be incorporated into cost calculations.

The final policy, `value change`, represents the case where one of the input parameters to the sta-

tic cost equation was underestimated (perhaps due to out-of-date tuning by the database administrator.) In the case where there are three inputs to the static cost equation, the actual cost is calculated as $[staticcostcalc(input1, input2 + lambda, input3)]$. For each query, we choose to adjust a non-trivial input to the cost equation, if available. For example, we would adjust the b_r value in the sort query, as it appears inside a logarithm function in the calculation. The purpose of this policy is to determine how well the two systems adapt to non-trivial changes in the factors that contribute to query cost.

In summary, our six lambda policies represent realistic overhead that may be encountered during a database query. They also serve as the backbone for drawing conclusions on the effectiveness of the existing and proposed query optimization engines at adapting to a changing environment.

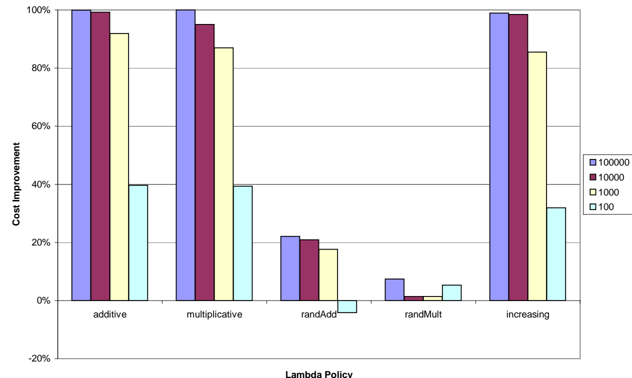


Figure 5: Cost estimate improvement of the FDQO query processor over the existing Non-FDQO query processor for various overhead trends. These results represent traces with all eight query types included.

5 Experimental Results

Using the simulator[1] described in Section 4, we collected results on a dual-Pentium4 2.5-GHz machine with 2GB RAM running Red Hat Linux 7.2. During our simulations, we focused on query cost as measured by wall-clock response time, and this choice is reflected in our cost equations listed in Table 1. We felt that response time is the most important cost measure, as it encompasses all of the overheads of disk access and network communication.

Results were collected by executing the simulator once for each of the six lambda policies, while varying the length of the query trace from 100, 1000, 10000, to 100000. Each of these tests was repeated, this time varying the make-up of each of the queries in the query trace. We first experimented with a query trace where each of the eight query types (linear search, binary search, selection, select+compare, sort, hash+join, projection, aggregation) had an equal chance of being produced in the trace. Next we instructed the trace generator to generate traces that contained only one type of query. This would allow us to observe the particular query types that FDQO predicts particularly well.

5.1 Query Estimate Improvements

As described in Section 4.1, our simulator compares the cost estimate error of the existing and proposed architecture. Figures 5-8 show the resulting cost error reduction achieved by the proposed FDQO query optimization system for each of the six lambda overhead policies.

Figure 5 depicts the mean reduction in cost error for a random query trace, where each of our eight query types has an equal probability of being generated. These results attempt to indicate the average query accuracy increase we can expect during a typical sequence of queries. There are several key observations to note from Figure 5. First of all, our system does well across the board for the predictable overhead patterns, such as `additive`, `multiplicative`, and `increasing`. This is to be expected, as each of these patterns require trivial changes to the cost equation to improve accuracy. For the traces that simulate noise, `randomized add` and `randomized mult`, we see that our system had very little improvement over the existing model, and in one case, cost accuracy was even degraded. This is not surprising because one of the major drawbacks of an adaptive system is that it may be overaggressive when adjusting its cost equations, due to the difficulty of recognizing the difference between overhead trends and noise.

Next, Figure 6, takes a closer look at our sixth lambda policy, `value change`. We chose to present this policy in terms of its performance for each individual query type, in order to portray our effectiveness as it becomes more difficult to estimate the lambda value encountered during actual execution. We were surprised find that the accuracy of the cost equations produced by the FDQO system was fairly consistent across all query types, regardless of the complexity of the lambda value. (For example, we would expect a parameter adjustment to a complex

[1]Source code for our simulator is contained in the appendix and available for download at our project web site.

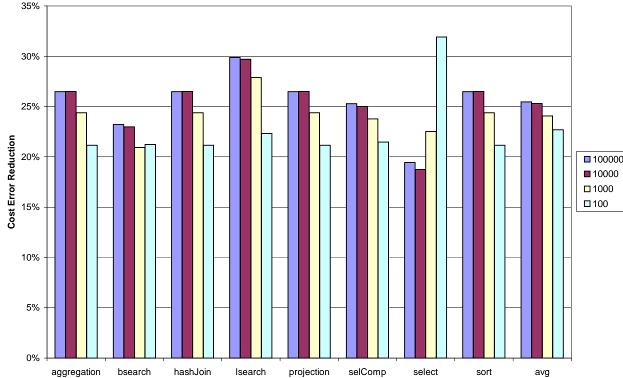


Figure 6: Cost estimate improvement of FDQO over Non-FDQO as we apply the `value change` policy (lambda costs are added to random inputs to the cost equations.) Results are separated by query type.

query cost equation to be much more difficult for our system to predict than a simple cost equation.) Nevertheless, our system consistently performed 20% better than the existing system. Upon closer inspection, we can see that for one of the outliers, `selection`, performance degrades as the trace size is increased. As selection is one of our more complex cost equations, we can probably attribute this degradation to our inability to accurately predict complex cost factors catching up with us. It may have been the case that for shorter query traces, we may have been lucky and achieved improvements.

Finally, Figures 7 and 8 break down the results from Figure 5 into the effects for each type of query. The main observation from these graphs is that we are consistently effective on predictable cost trends across all queries. Furthermore, we are consistently marginal in our improvement of queries containing overhead noise for the same reasons we described earlier for Figure 5.

5.2 System Adaptability

As described earlier, some of our experiments were specifically designed to test how our system responds to *overhead noise*. A system that overaggressively adjusts its cost equations will not only produce inaccurate cost estimates, but it will also incur unnecessary overhead. Based on the results in Figures 7 and 8, we can conclude that with the exception of Hash-Join, Linear Search, and Projection, our design is marginally effective at improving the query estimation performance in light of overhead noise. Our problems with the three query exceptions we mentioned were based on the fact that the cost estimations for these queries have relatively simple algorithms, therefore overhead noise becomes a dominating factor in the cost equation.

5.3 System Responsiveness

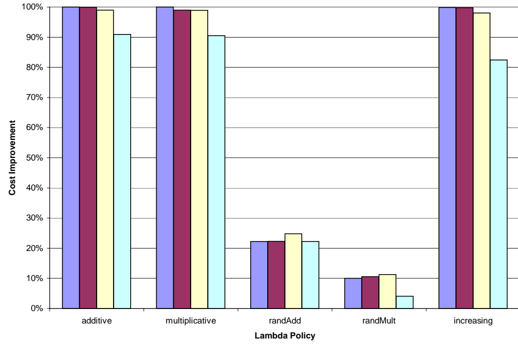
A notable observation from all of our resulting graphs (Figures 5-8) is that all policies perform much better after a significant warm-up period is achieved. This is because the cost history structures have learned the overhead patterns of the queries. Yet, even in the case of a query trace of length=100, when most of the query history structures are still empty, up to 40% improvement is possible. In general, we can conclude that our system is quickly responsive to cost changes, but is much more effective on long query traces.

6 Related Work

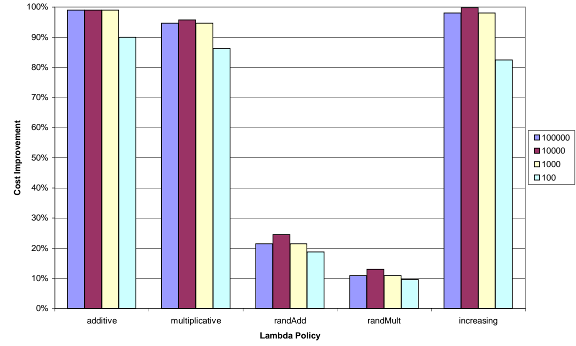
While much database research has focused on improving static query cost heuristics [4, 8], only the most recent work has attempted to tackle the problem of adaptive heuristics and self-tuning systems.

A great deal of recent query optimization research has focused on the concept known as piggybacking [2, 9]. Piggybacking refers to the technique where profile collectors are piggybacked onto queries as they are being performed. The profile collectors gather updates to the static parameters describing the database. These updates are eventually incorporated into the query estimators. It is important to note that even with the use of piggybacking, query estimation is not a self-tuning system. System administrators are still required to tune the weight of each database parameter. Therefore, we consider our research complementary to the work on piggybacking.

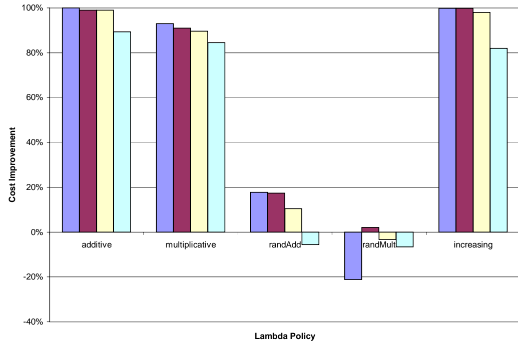
Other related work has been done in the area of Eddies [1], which is the idea of changing the order of sub-query execution on-the-fly, depending on the current delays experienced during query execution. Our work differs from the work on eddies because we do not change our optimization decisions mid-flight, but are instead optimizing our means for estimating costs and deciding between competing query execution options. Several other groups have researched the area of adaptive and mid-execution optimization of queries [3, 5, 7], and we believe that feedback-directed query optimization is complementary and non-overlapping with their research.



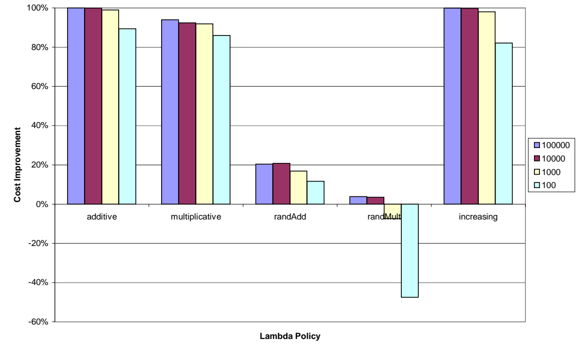
(a) Aggregation



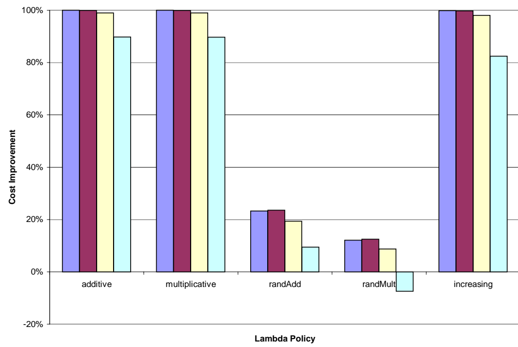
(b) Binary Search



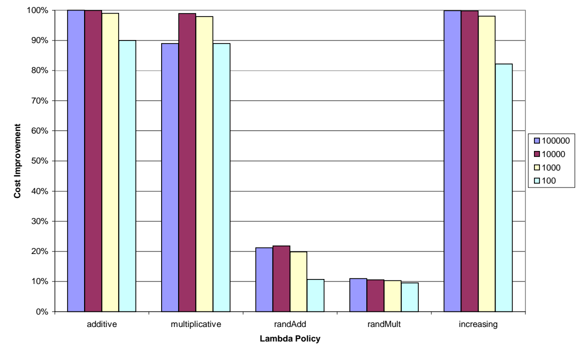
(c) Hash-Join



(d) Linear Search

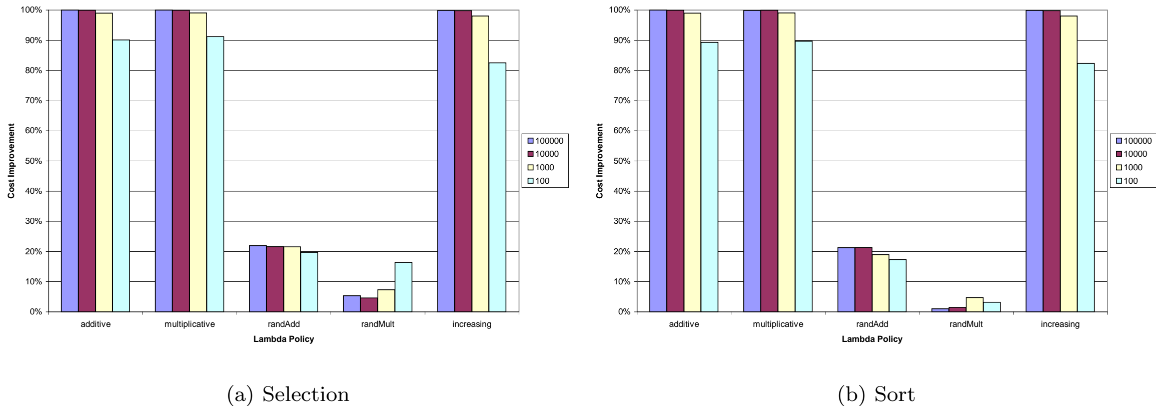


(e) Projection



(f) Select and Compare

Figure 7: Cost estimate improvement of FDQO over Non-FDQO for various overhead trends. Results are separated by query type.



(a) Selection

(b) Sort

Figure 8: Cost estimate improvement for selection and sort. (Continuation of Figure 7.)

Much of our work was motivated by recent advances in programming language compilation. In the compiler world, feedback-directed code optimization has enabled significant performance improvements over statically optimized code. By recording concrete performance achievements of code optimizations during previous program executions, the compiler can make more informed decisions on the optimizations that should be applied during new compilations. More recent advances in dynamic code optimization have shown that compilation can occur in parallel with program execution, while still achieving performance improvements if the compiler focuses on frequently executed portions of code. This focus on *hot execution streams* motivates us to focus on *hot queries* in future work.

7 Conclusions

The push toward self-tuning autonomous database management systems has compelled a great deal of database research in recent years. While the benefits of autonomy are obvious, path leading to such a DBMS is a challenging one, and there is a great deal of work left to be done. In this paper, we present a query processing system that has the potential for automatic tuning using cost profile feedback. Our design was evaluated using a trace-driven simulator. Our results indicate that the use of dynamic feedback during cost estimation can improve the accuracy of the cost estimation significantly, even for short sequences of queries. However, a real implementation that includes the overhead of maintaining profiled cost data for previous queries should be the final test.

8 Future Work

This research opens the doors for a vast amount of follow-up work in this area. Using the myriad of research spawning from feedback-directed compiler optimizations as a guide, we can envision dozens of publications and PhD theses that could stem from this work. Future work on optimal storage and retrieval techniques for our profile data could be explored. Smart techniques for deciding whether to use profile data or estimation for query decisions could be expanded. We could also research the existence of phase changes in the database that should trigger a flush of our profile cache. Now that we have produced an analytical evaluation, the simulation framework could be extended to provide for execution-driven simulation. This would provide a means for collecting overhead information. The research possibilities and implications are endless.

Acknowledgments

We wish to acknowledge the assistance of Mark Day for his invaluable input and advice during the entirety of this project. Professor Day suggested the use of the simulation model for testing the ideas presented within our work. We also wish to acknowledge the input of David Hoa, who focuses on database query optimization at IBM. David was very helpful in providing pointers to current research on the area. Finally, we wish to thank Glenn Holloway for his undying assistance as we fought with LaTeX.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, 2000.

[2] B. Dunkel, Q. Zhu, W. Lau, and S. Chen. Multiple-granularity interleaving for piggyback query processing. In *Proceedings of the 1999 CASCON*, 1999.

[3] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 299–310, 1999.

[4] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys (CSUR)*, 16(2):111–152, 1984.

[5] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 106–117, 1998.

[6] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. The McGraw-Hill Companies, Boston, Massachusetts, USA, 1999.

[7] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 130–141, 1998.

[8] C. T. Yu and C. C. Chang. Distributed query processing. *ACM Computing Surveys (CSUR)*, 16(4):399–433, 1984.

[9] Q. Zhu, B. Dunkel, N. Soparkar, S. Chen, B. Schiefer, and T. Lai. A piggyback method to collect statistics for query optimization in database management systems. In *Proceedings of the 1998 CASCON*, pages 67–82, 1998.

A Source Code Availability

The simulator and trace generator used for our experiments is available open source at our project website: `http://www.eecs.harvard.edu/~cettei/cs265/`

B Simulator Source Code

```
/************************************************************************
*
*  Feedback-Directed Query Optimization
*  Analytical Simulator
*
*  CS 265 Final Project
*  Written by: Kim Hazelwood Cettei
*  Last Update: Fri Dec 13 22:30:42 EST 2002
*
************************************************************************/

#include "query.h"
#include "analyticSim.h"
#include "sharedSim.h"
#include "math.h"

int main(int argc, char *argv[])
{
  int linesRead;

  if (argc != 2) {
    printf("Usage: %s [policy]\n", argv[0]);
    printf("0=NONE 1=ADD 2=MULT 3=RAND_ADD 4=RAND_MULT 5=INCR\n");
    exit(1);
  }
  simPolicy = atoi(argv[1]);

  /* open trace pipe */
  openPipe();

  /* initialize both systems */
  initSystems();

  /* read query trace */
  linesRead = processTraceFile();

  /* analyze results */
  analyzeResults(linesRead);

  pclose(tracePipe);
  return 0;
}

int processTraceFile()
{
  char queryString[20];
  QueryType queryType;
  int costFDQO, costNonFDQO, costActual;
  int lines = 0;
  Query *thisQuery = (Query*)malloc(sizeof(Query));

  while ( fscanf(tracePipe, "%s", queryString) != EOF) {
```

```
    queryType = translateQueryType(queryString);
    assert(queryType != INV);

    readQuery(queryType, thisQuery);
    if (DEBUG_PRINT) printQuery(thisQuery);

    /* simulate the non feedback-directed query optimizer */
    costNonFDQO = simulateNonFDQO(thisQuery);

    /* simulate the non feedback-directed query optimizer */
    costFDQO = simulateFDQO(thisQuery);

    /* simulate the actual cost */
    costActual = simulateActual(thisQuery);

    /* compare the two systems */
    compareCosts(lines, costNonFDQO, costFDQO, costActual);

    /* update FDQO database with actual query cost */
    updateHistory(thisQuery, costActual);

    lines++;
  }
  return lines;
}

void readQuery(QueryType queryType, Query * thisQuery)
{
  int valuesRead;
  int id, lambda;

  /* read rest of query */
  valuesRead = fscanf(tracePipe, "%d %d\n", &id, &lambda);
  assert(valuesRead==2);

  /* assign values to query struct */
  thisQuery->queryType = queryType;
  thisQuery->id = id;
  thisQuery->lambda = lambda;

  switch(queryType) {
    case LSEARCH: {
      LsearchQuery * thisQ = (LsearchQuery*)&(thisQuery->params.lsearch);
      fscanf(tracePipe, "%d", &(thisQ->numBlocks));
      break;
    }
    case BSEARCH: {
      BsearchQuery * thisQ = (BsearchQuery*)&(thisQuery->params.bsearch);
      fscanf(tracePipe, "%d", &(thisQ->numBlocks));
      break;
    }
    case SELECTION: {
```

```
      SelectionQuery * thisQ = (SelectionQuery*)&(thisQuery->params.selection);
      fscanf(tracePipe, "%d", &(thisQ->indexHeight));
      fscanf(tracePipe, "%d", &(thisQ->selectCard));
      fscanf(tracePipe, "%d", &(thisQ->fitInBlock));
      break;
    }
    case SELECTCOMP: {
      SelectCompQuery *thisQ= (SelectCompQuery*)&(thisQuery->params.selectComp);
      fscanf(tracePipe, "%d", &(thisQ->selectCard));
      fscanf(tracePipe, "%d", &(thisQ->numBlocks));
      break;
    }
    case SORT: {
      SortQuery * thisQ = (SortQuery*)&(thisQuery->params.sort);
      fscanf(tracePipe, "%d", &(thisQ->numBlocks));
      fscanf(tracePipe, "%d", &(thisQ->pageFrames));
      break;
    }
    case HASHJOIN: {
      HashJoinQuery * thisQ = (HashJoinQuery*)&(thisQuery->params.hashJoin);
      fscanf(tracePipe, "%d", &(thisQ->numBlocksR));
      fscanf(tracePipe, "%d", &(thisQ->numBlocksS));
      fscanf(tracePipe, "%d", &(thisQ->partitions));
      break;
    }
    case PROJECTION: {
      ProjectionQuery * thisQ=(ProjectionQuery*)&(thisQuery->params.projection);
      fscanf(tracePipe, "%d", &(thisQ->numDistinct));
      break;
    }
    case AGGREGATION: {
      AggregationQuery * thisQ =
          (AggregationQuery*)&(thisQuery->params.aggregation);
      fscanf(tracePipe, "%d", &(thisQ->numBlocks));
      break;
    }
    default:
      assert(false);
  }
  return;
}

void initSystems()
{
  int i, j;
  trender = 1;
  costArray = (QueryCost*)malloc(sizeof(QueryCost)*TRACELENGTH);
  for (i=0; i<MAXTYPES; i++) {
    for (j=0; j<NUMROWS; j++) {
      historyTable[i][j].id = -1;
      historyTable[i][j].cost = -1;
    }
  }
```

```
  return;
}

void openPipe()
{
  tracePipe = popen("tgen_knobs.txt", "r");
  if (tracePipe == NULL) {
    perror("analyticSim: popen:");
    exit(-1);
  }
  fscanf(tracePipe, "%d\n", &TRACELENGTH);
}

QueryType translateQueryType(char * queryString)
{
  if (strcmp(queryString, "lsearch") == 0) return LSEARCH;
  if (strcmp(queryString, "bsearch") == 0) return BSEARCH;
  if (strcmp(queryString, "selection") == 0) return SELECTION;
  if (strcmp(queryString, "selectcomp") == 0) return SELECTCOMP;
  if (strcmp(queryString, "sort") == 0) return SORT;
  if (strcmp(queryString, "hashjoin") == 0) return HASHJOIN;
  if (strcmp(queryString, "projection") == 0) return PROJECTION;
  if (strcmp(queryString, "aggregation") == 0) return AGGREGATION;
  return INV;
}

void compareCosts(int index, int costNonFDQO, int costFDQO, int costActual)
{
  assert(index < TRACELENGTH);
  costArray[index].costFDQO = costFDQO;
  costArray[index].costNonFDQO = costNonFDQO;
  costArray[index].costActual = costActual;
  if (DEBUG_PRINT)
    printf("\t\tCostNon: %d CostAct: %d CostFD: %d\n",
           costNonFDQO, costActual, costFDQO);
  return;
}

void analyzeResults(int linesRead)
{
  int i;
  int aCost, nCost, fCost;
  long totalDiffOld=0, totalDiffNew=0;
  long nDiff, fDiff;
  double aveNDiff, aveFDiff;
  double nMeanSq=0, fMeanSq=0;
  double nVariance, fVariance;
  double nStaDev, fStaDev;
  double improvement1;


  printf("\nRESULTS (based on %d queries)\n", linesRead);
  printPolicy();
```

```
  printf("----------------------------------------\n");

  /* First calculate arithmetic mean */
  for (i=0; i<linesRead; i++) {
    aCost = costArray[i].costActual;
    nCost = costArray[i].costNonFDQO;
    fCost = costArray[i].costFDQO;
    totalDiffOld += (long)abs(aCost - nCost);
    totalDiffNew += (long)abs(aCost - fCost);
  }
  aveNDiff = (double)totalDiffOld / (double)linesRead;
  aveFDiff = (double)totalDiffNew / (double)linesRead;

  /* Then calculate variance, standard deviation */
  for (i=0; i<linesRead; i++) {
    aCost = costArray[i].costActual;
    nCost = costArray[i].costNonFDQO;
    fCost = costArray[i].costFDQO;

    nDiff = (long)abs(aCost - nCost);
    fDiff = (long)abs(aCost - fCost);

    nMeanSq += pow(((double)nDiff - aveNDiff), (double)2);
    fMeanSq += pow(((double)fDiff - aveFDiff), (double)2);
  }
  nVariance = nMeanSq / (linesRead - 1);
  fVariance = fMeanSq / (linesRead - 1);

  nStaDev = sqrt(nVariance);
  fStaDev = sqrt(fVariance);

  improvement1 = ((aveNDiff-aveFDiff) / aveNDiff) * 100;

  printf("\t\tNON-FDQO\t\tFDQO\n");
  printf("aveDiff\t\t%7.2f\t\t\t%7.2f\n", aveNDiff, aveFDiff);
  printf("variance\t%7.2f\t\t\t%7.2f\n", nVariance, fVariance);
  printf("stdev\t\t%7.2f\t\t\t%7.2f\n\n", nStaDev, fStaDev);
  printf("FDQO improvement of %7.2f%%\n\n", improvement1);
  return;
}

void printQuery(Query * thisQuery) {
  switch(thisQuery->queryType) {
    case LSEARCH:
      printf("lsearch %d %d %d", thisQuery->id, thisQuery->lambda,
        thisQuery->params.lsearch.numBlocks);
      break;
    case BSEARCH:
      printf("bsearch %d %d %d", thisQuery->id, thisQuery->lambda,
        thisQuery->params.bsearch.numBlocks);
      break;
    case SELECTION:
      printf("selection %d %d %d %d %d", thisQuery->id, thisQuery->lambda,
```

```
        thisQuery->params.selection.indexHeight,
        thisQuery->params.selection.selectCard,
        thisQuery->params.selection.fitInBlock);
      break;
    case SELECTCOMP:
      printf("selectcomp %d %d %d %d", thisQuery->id, thisQuery->lambda,
        thisQuery->params.selectComp.selectCard,
        thisQuery->params.selectComp.numBlocks);
      break;
    case SORT:
      printf("sort %d %d %d %d", thisQuery->id, thisQuery->lambda,
        thisQuery->params.sort.numBlocks,
        thisQuery->params.sort.pageFrames);
      break;
    case HASHJOIN:
      printf("hashjoin %d %d %d %d %d", thisQuery->id, thisQuery->lambda,
        thisQuery->params.hashJoin.numBlocksR,
        thisQuery->params.hashJoin.numBlocksS,
        thisQuery->params.hashJoin.partitions);
      break;
    case PROJECTION:
      printf("projection %d %d %d", thisQuery->id, thisQuery->lambda,
        thisQuery->params.projection.numDistinct);
      break;
    case AGGREGATION:
      printf("aggregation %d %d %d", thisQuery->id, thisQuery->lambda,
        thisQuery->params.aggregation.numBlocks);
      break;
    default:
      assert(false);
  }
  if (NEWLINES) printf("\n");
}

void printPolicy()
{
  switch(simPolicy) {
    case NOPOLICY:
      printf("No Policy\n");
      break;
    case ADDITIVE:
      printf("Additive Policy\n");
      break;
    case MULTIPLICATIVE:
      printf("Multiplicative Policy\n");
      break;
    case RANDOM_ADD:
      printf("Randomized Add Policy\n");
      break;
    case RANDOM_MULT:
      printf("Randomized Multiply Policy\n");
      break;
    case INCREASING:
```

```
      printf("Increasing Policy\n");
      break;
    case VALCHANGE:
      printf("Random Value Changed Policy\n");
      break;
    default:
      printf("UKNOWN_POLICY\n");
  }
}
```

C Trace Generator Source Code

```
/*************************************************************************
*
*  Feedback-Directed Query Optimization
*  Trace Generation Engine
*
*  CS 265 Final Project
*  Written by: Kim Hazelwood Cettei
*  Last Update: Fri Dec 13 22:30:42 EST 2002
*
*************************************************************************/

#include "query.h"
#include "traceGen.h"

/*
 * Main
 *
 */

int main(int argc, char *argv[])
{
  /* Check input args */
  if (argc != 2) {
    printf("Usage: %s [knobsFile]\n", argv[0]);
    exit(-1);
  }

  /* Read knobs file */
  readKnobs(argv[1]);

  /* Init Query Tables */
  initQueryTables();

  /* Open the pipe */
  openPipe();

  /* Generate trace of queries and send it to the pipe */
  generateTrace();

  return 0;
}
```

```
/*
 * generateTrace
 */
void generateTrace()
{
  int i;
  QueryType qtype;

  for (i=0; i<TRACELENGTH; i++) {
    qtype = generateType();
    generateQuery(qtype);
  }
}

/*
 * generateQuery
 */
void generateQuery(QueryType thisType)
{
  switch (thisType) {
    case LSEARCH:
      generateLsearch();
      break;
    case BSEARCH:
      generateBsearch();
      break;
    case SELECTION:
      generateSelection();
      break;
    case SELECTCOMP:
      generateSelectComp();
      break;
    case SORT:
      generateSort();
      break;
    case HASHJOIN:
      generateHashJoin();
      break;
    case PROJECTION:
      generateProjection();
      break;
    case AGGREGATION:
      generateAggregation();
      break;
    default:
      fprintf(stderr, "Error generating query\n");
  }
}

/*
 * generateType
 */
QueryType generateType()
```

```
{
  double currentOdds = 0;
  double randVal = drand48();

  currentOdds += lsearchOdds;
  if (randVal < currentOdds) return LSEARCH;

  currentOdds += bsearchOdds;
  if (randVal < currentOdds) return BSEARCH;

  currentOdds += selectionOdds;
  if (randVal < currentOdds) return SELECTION;

  currentOdds += selectCompOdds;
  if (randVal < currentOdds) return SELECTCOMP;

  currentOdds += sortOdds;
  if (randVal < currentOdds) return SORT;

  currentOdds += hashJoinOdds;
  if (randVal < currentOdds) return HASHJOIN;

  currentOdds += projectionOdds;
  if (randVal < currentOdds) return PROJECTION;

  else return AGGREGATION;
}

/*
 * generateLsearch
 */
void generateLsearch()
{
  Query thisQuery;
  int queryIndex = (int)(drand48() * 10);
  if (queryIndex >= MAXQUERIES) queryIndex = MAXQUERIES-1;
  thisQuery = queryTables[LSEARCH][queryIndex];

  if (thisQuery.queryType != LSEARCH) {
    fprintf(stderr, "Problem accessing query!\n");
    exit(-1);
  }
  fprintf(simPipe, "lsearch %d %d ", thisQuery.id, thisQuery.lambda);
  fprintf(simPipe, "%d\n", thisQuery.params.lsearch.numBlocks);
}

/*
 * generateBsearch
 */
void generateBsearch()
{
  Query thisQuery;
  int queryIndex = (int)(drand48() * 10);
```

```
  if (queryIndex >= MAXQUERIES) queryIndex = MAXQUERIES-1;
  thisQuery = queryTables[BSEARCH][queryIndex];

  if (thisQuery.queryType != BSEARCH) {
    fprintf(stderr, "Problem accessing query!\n");
    exit(-1);
  }
  fprintf(simPipe, "bsearch %d %d ", thisQuery.id, thisQuery.lambda);
  fprintf(simPipe, "%d\n", thisQuery.params.bsearch.numBlocks);
}

/*
 * generateSelection
 */
void generateSelection()
{
  Query thisQuery;
  int queryIndex = (int)(drand48() * 10);
  if (queryIndex >= MAXQUERIES) queryIndex = MAXQUERIES-1;
  thisQuery = queryTables[SELECTION][queryIndex];

  if (thisQuery.queryType != SELECTION) {
    fprintf(stderr, "Problem accessing query!\n");
    exit(-1);
  }
  fprintf(simPipe, "selection %d %d ", thisQuery.id, thisQuery.lambda);
  fprintf(simPipe, "%d ", thisQuery.params.selection.indexHeight);
  fprintf(simPipe, "%d ", thisQuery.params.selection.selectCard);
  fprintf(simPipe, "%d\n", thisQuery.params.selection.fitInBlock);
}

/*
 * generateSelectComp
 */
void generateSelectComp()
{
  Query thisQuery;
  int queryIndex = (int)(drand48() * 10);
  if (queryIndex >= MAXQUERIES) queryIndex = MAXQUERIES-1;
  thisQuery = queryTables[SELECTCOMP][queryIndex];

  if (thisQuery.queryType != SELECTCOMP) {
    fprintf(stderr, "Problem accessing query!\n");
    exit(-1);
  }
  fprintf(simPipe, "selectcomp %d %d ", thisQuery.id, thisQuery.lambda);
  fprintf(simPipe, "%d ", thisQuery.params.selectComp.selectCard);
  fprintf(simPipe, "%d\n", thisQuery.params.selectComp.numBlocks);
}

/*
 * generateSort
 */
```

```
void generateSort()
{
  Query thisQuery;
  int queryIndex = (int)(drand48() * 10);
  if (queryIndex >= MAXQUERIES) queryIndex = MAXQUERIES-1;
  thisQuery = queryTables[SORT][queryIndex];

  if (thisQuery.queryType != SORT) {
    fprintf(stderr, "Problem accessing query!\n");
    exit(-1);
  }
  fprintf(simPipe, "sort %d %d ", thisQuery.id, thisQuery.lambda);
  fprintf(simPipe, "%d ", thisQuery.params.sort.numBlocks);
  fprintf(simPipe, "%d\n", thisQuery.params.sort.pageFrames);
}

/*
 * generateJoin
 */
void generateHashJoin()
{
  Query thisQuery;
  int queryIndex = (int)(drand48() * 10);
  if (queryIndex >= MAXQUERIES) queryIndex = MAXQUERIES-1;
  thisQuery = queryTables[HASHJOIN][queryIndex];

  if (thisQuery.queryType != HASHJOIN) {
    fprintf(stderr, "Problem accessing query!\n");
    exit(-1);
  }
  fprintf(simPipe, "hashjoin %d %d ", thisQuery.id, thisQuery.lambda);
  fprintf(simPipe, "%d ", thisQuery.params.hashJoin.numBlocksR);
  fprintf(simPipe, "%d ", thisQuery.params.hashJoin.numBlocksS);
  fprintf(simPipe, "%d\n", thisQuery.params.hashJoin.partitions);
}

/*
 * generateProjection
 */
void generateProjection()
{
  Query thisQuery;
  int queryIndex = (int)(drand48() * 10);
  if (queryIndex >= MAXQUERIES) queryIndex = MAXQUERIES-1;
  thisQuery = queryTables[PROJECTION][queryIndex];

  if (thisQuery.queryType != PROJECTION) {
    fprintf(stderr, "Problem accessing query!\n");
    exit(-1);
  }
  fprintf(simPipe, "projection %d %d ", thisQuery.id, thisQuery.lambda);
  fprintf(simPipe, "%d\n", thisQuery.params.projection.numDistinct);
}
```

```
/*
 * generateAggregation
 */
void generateAggregation()
{
  Query thisQuery;
  int queryIndex = (int)(drand48() * 10);
  if (queryIndex >= MAXQUERIES) queryIndex = MAXQUERIES-1;
  thisQuery = queryTables[AGGREGATION][queryIndex];

  if (thisQuery.queryType != AGGREGATION) {
    fprintf(stderr, "Problem accessing query!\n");
    exit(-1);
  }
  fprintf(simPipe, "aggregation %d %d ", thisQuery.id, thisQuery.lambda);
  fprintf(simPipe, "%d\n", thisQuery.params.aggregation.numBlocks);
}

void readKnobs(char * filename)
{
  FILE * knobsFile;
  char thisKnob[20], thisVal[10];

  /* Clear query odds */
  lsearchOdds = 0;
  bsearchOdds = 0;
  selectionOdds = 0;
  selectCompOdds = 0;
  sortOdds = 0;
  hashJoinOdds = 0;
  projectionOdds = 0;
  aggregationOdds = 0;

  knobsFile = fopen(filename, "r");
  if (knobsFile == NULL) {
    fprintf(stderr, "Bad knobs file\n");
    exit(-1);
  }
#if VERBOSE
  fprintf(stderr, "Reading %s...\n", filename);
#endif
  while (fscanf(knobsFile, "%s = %s", thisKnob, thisVal) != EOF) {
    if (strcmp(thisKnob, "TRACE_LENGTH")==0) {
      TRACELENGTH = atoi(thisVal);
    }
    else if (strcmp(thisKnob, "LSEARCH_ODDS")==0) {
      lsearchOdds = atof(thisVal);
    }
    else if (strcmp(thisKnob, "BSEARCH_ODDS")==0) {
      bsearchOdds = atof(thisVal);
    }
    else if (strcmp(thisKnob, "SELECTION_ODDS")==0) {
```

```
      selectionOdds = atof(thisVal);
    }
    else if (strcmp(thisKnob, "SELECTCOMP_ODDS")==0) {
      selectCompOdds = atof(thisVal);
    }
    else if (strcmp(thisKnob, "SORT_ODDS")==0) {
      sortOdds = atof(thisVal);
    }
    else if (strcmp(thisKnob, "HASHJOIN_ODDS")==0) {
      hashJoinOdds = atof(thisVal);
    }
    else if (strcmp(thisKnob, "PROJECTION_ODDS")==0) {
      projectionOdds = atof(thisVal);
    }
    else if (strcmp(thisKnob, "AGGREGATION_ODDS")==0) {
      aggregationOdds = atof(thisVal);
    }
  }
}

void initQueryTables() {
  int i, j;
  int uniqueID = 0;

  for (i=LSEARCH; i<MAXTYPES; i++) {
    for (j=0; j<MAXQUERIES; j++) {
      queryTables[i][j].queryType = i;
      queryTables[i][j].id = uniqueID++;
      queryTables[i][j].lambda = (int)(drand48() * 10);
      switch(i){
        case LSEARCH: {
          LsearchQuery * thisLsearch;
          thisLsearch = (LsearchQuery *)&(queryTables[i][j].params.lsearch);
          thisLsearch->numBlocks =(int)(drand48() * 100000);
          break;
        }
        case BSEARCH: {
          BsearchQuery * thisBsearch;
          thisBsearch = (BsearchQuery *)&(queryTables[i][j].params.bsearch);
          thisBsearch->numBlocks =(int)(drand48() * 100000);
          break;
        }
        case SELECTION: {
          SelectionQuery * thisSelect;
          thisSelect = (SelectionQuery *)&(queryTables[i][j].params.selection);
          thisSelect->indexHeight =(int)(drand48() * 500);
          thisSelect->selectCard =(int)(drand48() * 10000);
          thisSelect->fitInBlock =(int)(drand48() * 25);
          break;
        }
        case SELECTCOMP: {
          SelectCompQuery * thisSelect;
          thisSelect= (SelectCompQuery *)&(queryTables[i][j].params.selectComp);
```

```
          thisSelect->selectCard =(int)(drand48() * 10);
          thisSelect->numBlocks =(int)(drand48() * 100000);
          break;
        }
        case SORT: {
          SortQuery * thisSort;
          thisSort = (SortQuery *)&(queryTables[i][j].params.sort);
          thisSort->numBlocks =(int)(drand48() * 100000);
          thisSort->pageFrames =(int)(drand48() * 25);
          break;
        }
        case HASHJOIN: {
          HashJoinQuery * thisJoin;
          thisJoin = (HashJoinQuery *)&(queryTables[i][j].params.hashJoin);
          thisJoin->numBlocksR =(int)(drand48() * 100000);
          thisJoin->numBlocksS =(int)(drand48() * 100000);
          thisJoin->partitions =(int)(drand48() * 100);
          break;
        }
        case PROJECTION: {
          ProjectionQuery * thisProject;
          thisProject=(ProjectionQuery *)&(queryTables[i][j].params.projection);
          thisProject->numDistinct =(int)(drand48() * 50);
          break;
        }
        case AGGREGATION: {
          AggregationQuery * thisAgg;
          thisAgg = (AggregationQuery *)&(queryTables[i][j].params.aggregation);
          thisAgg->numBlocks =(int)(drand48() * 100000);
          break;
        }
        default:
          assert(false);
      }
    }
  }
}

void openPipe() {

  simPipe = (FILE *)stdout;
  fprintf(simPipe, "%d\n", TRACELENGTH);
}
```

D Cost Calculation Source Code

```
/************************************************************************
*
*  Feedback-Directed Query Optimization
*  Query Cost Calculator
*
*  CS 265 Final Project
*  Written by: Kim Hazelwood Cettei
*  Last Update: Fri Dec 13 22:30:42 EST 2002
```

```
*
*  Description: Calculates the query cost for different query optimizers
*
************************************************************************/

#include "query.h"
#include "sharedSim.h"
#include "math.h"

int staticCostEstimate(Query * thisQuery) {
  int cost, indexHeight, selectCard, fitInBlock, pageFrames, partitions;
  int numBlocks, numBlocksR, numBlocksS;

  switch(thisQuery->queryType) {
    case LSEARCH:
      numBlocks = thisQuery->params.lsearch.numBlocks;
      cost = numBlocks / 2;
      break;
    case BSEARCH:
      numBlocks = thisQuery->params.bsearch.numBlocks;
      cost = (int)log((double)numBlocks);
      break;
    case SELECTION:
      indexHeight = thisQuery->params.selection.indexHeight;
      selectCard = thisQuery->params.selection.selectCard;
      fitInBlock = thisQuery->params.selection.fitInBlock;
      cost = indexHeight + (selectCard / fitInBlock);
      break;
    case SELECTCOMP:
      numBlocks = thisQuery->params.selectComp.numBlocks;
      selectCard = thisQuery->params.selectComp.selectCard;
      cost = selectCard + (numBlocks / 2);
      break;
    case SORT:
      /* cost = nb * (2(log(pf-1)(nb/pf))+ 1) */
      numBlocks = thisQuery->params.sort.numBlocks;
      pageFrames = thisQuery->params.sort.pageFrames;
      cost = (int)log((double)(pageFrames-1));
      cost = cost / log((double)(numBlocks/pageFrames));
      cost = cost * 2 + 1;
      cost = cost * numBlocks;
      break;
    case HASHJOIN:
      numBlocksR = thisQuery->params.hashJoin.numBlocksR;
      numBlocksS = thisQuery->params.hashJoin.numBlocksS;
      partitions = thisQuery->params.hashJoin.partitions;
      cost = (3 * (numBlocksR + numBlocksS)) + (4 * partitions);
      break;
    case PROJECTION:
      cost = thisQuery->params.projection.numDistinct;
      break;
    case AGGREGATION:
      cost = thisQuery->params.aggregation.numBlocks;
```

```
      break;
    default:
      printf("costNonFDQO: Got a bogus query\n");
      exit(1);
  }
  return cost;
}

int simulateNonFDQO(Query * thisQuery) {
  return staticCostEstimate(thisQuery);
}

int simulateFDQO(Query * thisQuery)
{
  int cost;
  int lastCost;

  cost = staticCostEstimate(thisQuery);
  lastCost = searchHistory(thisQuery->queryType, thisQuery->id);

  if (lastCost > 0) return lastCost;
  else return cost;
}

int searchHistory(QueryType type, int id)
{
  int j;
  int cost = -1;

  for (j=0; j<NUMROWS; j++) {
    if (historyTable[type][j].id == id)  {
      cost = historyTable[type][j].cost;
      break;
    }
  }
  return cost;
}

void updateHistory(Query * thisQuery, int cost)
{
  int j;
  int id = thisQuery->id;
  int type = thisQuery->queryType;

  for (j=0; j<NUMROWS; j++) {
    if (historyTable[type][j].id == id) {
      historyTable[type][j].cost = cost;
      break;
    }
    else if (historyTable[type][j].id == -1) {
      historyTable[type][j].id = id;
      historyTable[type][j].cost = cost;
      break;
```

```
    }
  }
}

int simulateActual(Query * thisQuery)
{
  int cost;
  float floatcost;

  cost = staticCostEstimate(thisQuery);

  switch (simPolicy) {
    case ADDITIVE:
      cost += thisQuery->lambda;
      break;
    case MULTIPLICATIVE:
      floatcost = ((float)thisQuery->lambda/10) + 1;
      floatcost = cost * floatcost;
      cost = (int)floatcost;
      break;
    case RANDOM_ADD:
      cost += (int)(drand48() * thisQuery->lambda);
      break;
    case RANDOM_MULT:
      floatcost = (drand48()*thisQuery->lambda/10) + 1;
      floatcost = cost * floatcost;
      cost = (int)floatcost;
      break;
    case INCREASING:
      cost += thisQuery->lambda;
      cost += trender;
      trender++;
      break;
    case VAL_CHANGE:
      changeValue(thisQuery);
      cost = staticCostEstimate(thisQuery);
    default:
      break;
  }
  return cost;
}

void changeValue(Query *thisQuery)
{
  int lambda = thisQuery->lambda;

  switch(thisQuery->queryType) {
    case L_SEARCH:
      thisQuery->params.lsearch.numBlocks += lambda;
      break;
    case B_SEARCH:
      thisQuery->params.bsearch.numBlocks += lambda;
      break;
```

```
      case SELECTION:
        thisQuery->params.selection.selectCard += lambda;
        break;
      case SELECTCOMP:
        thisQuery->params.selectComp.numBlocks += lambda;
        break;
      case SORT:
        thisQuery->params.sort.numBlocks += lambda;
        break;
      case HASHJOIN:
        thisQuery->params.hashJoin.partitions += lambda;
        break;
      case PROJECTION:
        thisQuery->params.projection.numDistinct += lambda;
        break;
      case AGGREGATION:
        thisQuery->params.aggregation.numBlocks += lambda;
        break;
      default:
        break;
  }
}
```

E README File

```
This directory contains all of the components for simulating and analyzing
Feedback-Directed Query Optimization.

The main components consist of a Trace Generator (traceGen.c) and an
Analytical Simulator (analyticSim.c). The trace generator performs the
duty of generating a pseudo-random trace of queries. This query trace
is used to drive the analytical simulator. The two programs communicate
via OS pipes.

The analytical simulator attempts to replicate the decisions made by a
standard query optimizer, as well as our proposed feedback-directed query
optimizer. Resulting costs and cost-estimate errors are compared on the
two systems.


***** FILES CONTAINED IN THIS DIRECTORY *****

analyticSim.[ch]: Contains routines that form the core of the analytical
        simulator.
traceGen.[ch]: Contains routines that form the core of the pseudo-random
        query trace generator.
costCalc.c: Contains routines for estimating the query cost in the three
        compared systems: fdqo, non-fdqo, actual.
query.h: Data structures and globals that are shared by both of the above
        systems.
knobs.txt: A knobs file that controls the pseudo-randomness of the query
        trace generator.
Makefile: Controls the build and execute process
```

***** EXECUTABLES CONTAINED IN THIS DIRECTORY *****

asim   The analytical simulator

tgen   The query trace generator

All files written by Kim Hazelwood Cettei, December 2002.