2005

AQUAGP: Approximate QUery Answering Using Genetic Programming

Jason Brandon Peltzer

AQUAGP:
Approximate QUery Answering
Using
Genetic Programming

by

Jason Brandon Peltzer

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Supervised by

Dr. Ankur M. Teredesai
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York
December 2005

Approved By:

Ankur M. Teredesai

Dr. Ankur M. Teredesai
Primary Adviser

Peter G. Anderson

Dr. Peter G. Anderson
Reader

Rajendra K. Raj

Dr. Rajendra K. Raj
Observer

Thesis/Dissertation Author Permission Statement

Title of thesis or dissertation: AQUAGP: Approximate QUery Answering using Genetic Programming

Name of author: Jason Brandon Peltzer
Degree: Masters of Science
Program: Computer Science
College: GCCIS

I understand that I must submit a print copy of my thesis or dissertation to the RIT Archives, per current RIT guidelines for the completion of my degree. I hereby grant to the Rochester Institute of Technology and its agents the non-exclusive license to archive and make accessible my thesis or dissertation in whole or in part in all forms of media in perpetuity. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Print Reproduction Permission Granted:

I, Jason Peltzer, hereby **grant permission** to the Rochester Institute Technology to reproduce my print thesis or dissertation in whole or in part. Any reproduction will not be for commercial use or profit.

Signature of Author: Jason Peltzer Date: _____

Print Reproduction Permission Denied:

I, _____, hereby **deny permission** to the RIT Library of the Rochester Institute of Technology to reproduce my print thesis or dissertation in whole or in part.

Signature of Author: _____ Date: _____

Dedication

I dedicate this thesis in memory of my late grandmother Dorothy Koretsky, who believed in me when nobody else did, including myself. She is a never ending source of encouragement for me.

Acknowledgments

I would like to thank my Advisor Dr. Ankur M. Teredesai, and the rest of my committee for all of their aid and advisement during our research. I would also like thank the following students who aided in my efforts and research: William Leiserson, Andrew Schenck, and Garrett Reinard.

Abstract

Speed, cost, and accuracy are crucial performance parameters while evaluating the quality of a query using any Database Management System (DBMS). For some queries it may be possible to approximate the answer using an approximate query answering algorithm or tool. Also, for certain queries, it may not be critical to determine the *perfect/exact* results so long as the following conditions are true: (a) a high percentage of the relevant data is retrieved correctly, (b) irrelevant or extra data is minimized, and (c) an approximate answer (if available) results in a significant savings in terms of the overall query cost and retrieval time. In this paper we describe a novel approach for approximate query answering using the Genetic Programming (GP) paradigms. We develop an evolutionary computing based query space exploration framework. Given an input query and the database schema, our framework uses tree-based GP to automatically generate and evaluate approximate query candidates. We highlight and discuss different avenues we explored. We evaluate the success of our experiments based on the speed, the cost, and the accuracy of the results retrieved by the re-formulated (GP generated) queries and present the results on a variety of query types for TPC-benchmark and PKDD-benchmark datasets.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

1.1 Introduction

Query processors in most modern database management systems include some form of query approximation or query optimization component(s). There are several scenarios where an exact answer may not be required and the end-user may prefer a fast approximate answer. For example, say the exact answer to the query, *Find the number of employees with salary greater than $50,000,* is 48 employees, and it takes 6 hours to run the query. On the other hand, an approximate query answering system may return the answer as 50 employees in 6 seconds, thereby saving significant time and effort if the original query was intended to gain a general idea about the underlying data. The approximate query answering system can also provide appropriate confidence bounds to advise the user of the validity of the approximate answer as compared to the exact answer. i.e. 50 (+/- 5 employees) as in the previous example.

1.1.1 Problem Formulation

The approximate query answering problem we are trying to solve (using a Genetic Programming based implementation) can be defined as follows:

Definition Approximate Query: Find a query Q_x such that:

$$C_{Q_i} \geq C_{Q_i^o} + S_i \tag{1.1}$$

where C_{Q_i} is the aggregate cost of executing the user query, S_i is the cost of finding Q_x, and $C_{Q_i^o}$ is the aggregate cost of executing an alternate query that is more optimal compared to the original query, such that:

$$\tau_{Q_i} \approx \tau_{Q_i^o} \tag{1.2}$$

where τ_{Q_i} is the tuple-set-based accuracy of the query Q_i.

Hence, the idea is to search the query space using a heuristic search procedure with the search cost S_i to find the most optimal query that provides a good approximate answer. This search procedure is a multi-objective optimization problem where the goal is to minimize the query processing costs and maximize the accuracy of the query results. In this paper we describe a Genetic-Programming-based search procedure to search for approximate queries. Note that the cost S_i is a one-time cost whereas the C_x costs are on a per query basis. Hence once the optimal query is found (off line), it can be repetitively used with significant cost savings.

As shown in Figure 1.1 the traditional query processor typically accepts an input query Q_i and passes it to the parser to generate alternative query plans. The query plan cost estimator provides the cost of each plan to the optimal plan selector. The optimal plan selector then chooses the query plan with the least cost estimate for execution as specified by Q_i^o.

The lower portion of the figure describes our implementation. We begin with the original query and the least-cost-estimate query plan as decided by the underlying DBMS (MS-SQL server in our case). We also provide the database schema as an input to the GP framework. We first randomly generate a population of alternate query plans. Then, we use the query plan cost estimator provided by the native database engine to estimate the cost of these generated query plans. We select the most efficient plans from this initial population based on a threshold for aggregate costs and continue the evolutionary process. The threshold value is based on the estimated cost of executing the least-cost query plan as found by the database engine $C_{Q_i^o}$. Using this cost, we can compute an aggregate cost for the alternate query plans that GP proposes. Note that we are trying to achieve a query with

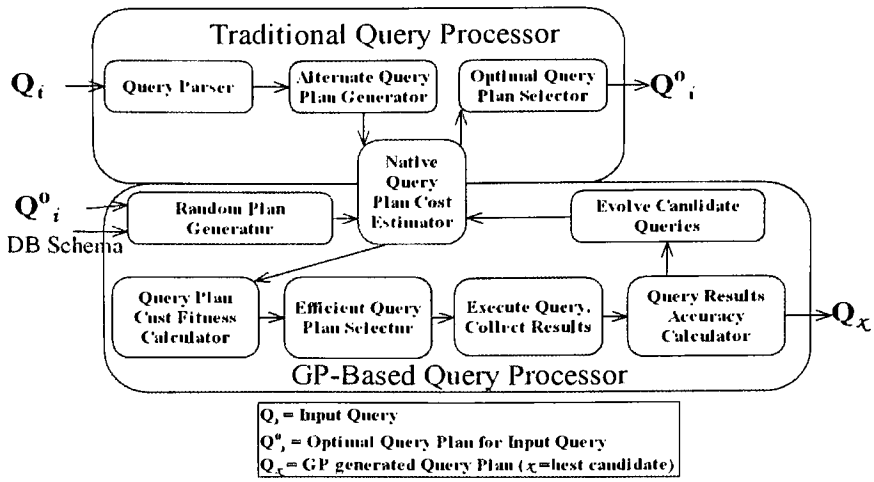


Figure 1.1: Overall architecture of the proposed GP based Approximate Query Answering System. The upper portion of the diagram describes the traditional query processor, while the lower portion describes components of the proposed GP based system.

a cost not only lower than the input query, but also lower than the best estimate that the native DBMS suggested. The assumption is that as the estimated cost of an alternate query goes down, the the execution time becomes faster for that query. Hence, any query whose aggregate cost is greater than that of the input query's estimated cost does not get evaluated by the GP. Using this technique, we only evaluate query plans that are less expensive than the original input query.

It should be noted that one can leverage the existing frameworks such as the AQUA framework [1] to estimate the (in)accuracy of candidate queries to obtain an understanding of the level of approximation using the data summaries. Hence, the motivation for the name AQUAGP. The best candidates are selected for further evolution according to the proposed GP fitness computation algorithm we developed. This process iterates for a pre-specified number of iterations or until a specific termination criteria is met. Typical termination criteria may be a pre-specified accuracy threshold, or any other such fitness measure.

1.2 Background and Related Work

In this section, we will discuss two main areas of research in the field of query approximation and optimization:

1. Algorithmic approaches

2. Formal Systems for query optimization.

We will then have a brief discussion of our contributions and motivation for doing such work in the field of query approximation and optimization.

1.2.1 Previous Heuristic and Non-Heuristic Algorithms

We briefly review the most significant efforts in the fields of query optimization and query approximation in this section. The major effort for optimization deals with cost analysis of queries based on different attributes of the query. Query optimizers have to efficiently iterate through different parse trees for each query to decide what the most optimal query plan could be. Traditional query optimizers tend not to enumerate all possibilities for query plans due to computational constraints. The query optimizer prunes the plan space while enumerating plans in order to cut down on the computational requirements. Such pruning significantly reduces the amount of space and time required to optimize the queries that may contain a lot of Joins. Generally this pruning is done using dynamic programming (DP) algorithms. Hellerstein proposed one such popular DP based approach [6]. Earlier efforts include the work by Graefe, *et al.,* where they survey many techniques for executing complex queries over large data sets [5]. They suggest that a complex query is a query that requires a number of query processing algorithms to work together. They also define a large database as a database that may use as little as megabytes to store the data, and possibly as much as terabytes to store the data. Some aspects deemed very important in Graefe's work are relevant in any query optimization process and include: (a) cost evaluation algorithms and their execution costs, (b) sorting vs. hashing, (c) parallelism, (d)

resource allocation, and (e) scheduling issues in complex queries, etc. An important aspect of query optimization that is also very important is an efficient index selection algorithm. Using referential integrity constraints to optimize queries can be very useful, and we make use of this in our work. In the work done by Weddell, *et al.* [16], they propose a class based index selection algorithm, and they use index selection to determine optimal index use for partial match queries for a simple semantic data model. They are able to find an optimal set of the smallest possible indices that guarantee each query can be efficiently evaluated, and as well they discuss a procedure for refining the set of index choices to result in an overall lower cost. In the work by Yoo, *et al.* [17], they propose an optimization method that is independent of cost models using semi-joins. Their solution proposes an optimization to dynamic programming solutions using search algorithms; their algorithm of choice is the A* algorithm. The A* algorithm is a graph-based searching algorithm and they propose a branch and bound algorithm that does not re-evaluate already eliminated subsets of information. They keep generality in their algorithm by allowing the cost functions involved in doing A* to be any cost functions for a specified system. Steinbrunn, *et al.* [12], outline other algorithms such as deterministic algorithms, dynamic programming solutions, minimum selectivity, the Krishnamurty-Boral-Zanialo (KBZ) algorithm, the AB algorithm, randomized algorithms, iterative improvement, simulated annealing, two phase optimization, toured simulated annealing, and random sampling as relevant heuristic optimizations for the join ordering problem. They also explain the use of genetic algorithms (GA) as a query optimization technique.

In most areas of computing there are always drawbacks to using one approach over another; one approach may have an extremely large time complexity while another approach may have a large space complexity. It is unfortunate, however, that evolutionary algorithms tend to have a very high cost of computation. As well, pure heuristic algorithms tend to have a lower quality of solutions that are found [19]. In the work by Zhang, *et al.* [19], they propose the use of hybrid algorithms which combine the qualities of pure heuristic algorithms with evolutionary algorithms. Their work is centered on global processing plans

for data warehouses to reduce the total query and maintenance costs on data warehouses.

1.2.2 Formal Systems for Query Optimization

Query processors typically perform several optimizations and approximations on-the-fly as queries are presented in a pervasive manner without specific direction from the end-user. Few such systems currently exist; one AutoAdmin from Microsoft, and the Aqua project from Bell Laboratories. AutoAdmin is a self-tuning and self-administering system for the MS SQL Server DBMS. Using an index tuning wizard, a database administrator can optimize a database relatively easily and inexpensively [9] [`http://research.microsoft.com/dmx/autoadmin/default.asp`]. Aqua is designed for approximate query answering that can be used with an SQL-Compliant DBMS. Aqua precomputes statistical summaries of data, generally in the form of histograms. Aqua also provides error/confidence bounds on the answer returned [1, 2]. In many cases query optimization is an interactive process where the analyst has effective domain knowledge of the underlying relations, indices and constraints to speed up or approximate the query by hand. All this is typically based on the query plan for the original query. The database system, in itself, proposes efficient ways of restructuring the query plan based on statistical analyses of the underlying database using histograms and other data summarization techniques [10]. Logic-based approaches and semantic transformations were initially proposed to solve this problem around twenty years ago [8]. The approach described in the work by Sun, *et al.* [14], proposes semantic integrity constraints for databases. In the work by Yu, *et al.* [18], they propose an optimization technique using knowledge acquisition from a database. They are able to perform their optimizations using dynamically semantic integrity constraints based on different database states. As noted in their work, if the query processor can dynamically identify the types of constraints and store them, then it may be able to answer subsequent queries of the same type more efficiently by inference, without physically touching the database. Work by Shekar, *et al.* [11], describes a data driven approach to query-transformation rule learning. Their approach differs from most, including the work

that we describe later on in this paper, in that their system is data driven, rather than query driven. Their approach is appropriate for a system that has large numbers of differing queries being issued to the database, as they can have rules for the data based on different attributes within a database. Many query-driven approaches may incur larger costs when many new queries are presented to the database, whereas their approach is able to handle this with a lower cost at the time of query execution.

As the level of statistical analysis techniques improve, efficient user interaction provides even greater insight and value to the overall optimization process. Let's take a case in point. The Microsoft SQL Server 2005's query optimizer takes a query and creates a parse tree for it. From the parse tree of the query, statistical information, such as (a) **estimated IO Time**, (b) **estimated CPU Time**, (c) **estimated rows resulting from a Join**, (d) **estimated total subtree cost**, etc., is computed and translated into various types of query costs. The main idea is to make use of the statistics that the DBMS provides, and use them to approximate/optimize queries, no matter the implementation.

These statistics then result in an aggregate query cost estimate. The entire process of generating a parse tree, computing statistics, and estimating query costs results in the generation of a query plan for the original input query. The query processor will then (typically) try and come up with several different permutations of the parse tree, and assess the different individual costs associated with each parse tree. For example, using relational algebra, different operator trees can implement the same algebraic expressions [3]. Equation 1.3 suggests how two join orders result in equivalent tuple sets being returned but might have different query costs associated with them, based on the size and order of the intermediate joins:

$$((A \bowtie B) \bowtie C) \cong ((B \bowtie C) \bowtie A) \tag{1.3}$$

In the work by Lee et. al. [9], they propose a graph-based approach to query optimization. They use a graph-theoretic model to represent all types of operations. The $\bowtie$ operation is used most frequently, and it has a high cost. Each $\bowtie$ is represented as a vertex in the

graph. A directed edge between two $\bowtie$ operations then indicates the ordering of the operations. This representation can show all possible permutations of the query graph. They use spanning trees to determine lower-cost query plans. Another attempt to find the optimal nesting order for N-Relational joins is shown in the work by Ibaraki, *et al.* [7]. They propose a method that speeds up the computation of multi-relational joins using a nested loops program, and their contribution embeds indices into the data structures holding relations, as to avoid unnecessary page fetches. As the problem of computing the optimal nesting order for N-Relational joins is generally NP-complete, they too resort to using different heuristics to compute the nesting order. There is still a large problem of computing the proper nesting order for queries with many joins. As the number of joins in a query gets increasingly large, it takes the query optimizer more time to compute the best plan, so the query optimizer must take certain shortcuts and will often drop down its optimization level, may use less time, and may still obtain a poor execution plan [15]. In the work by Tao, *et al.* [15], they propose a method to solve the optimization degradation that can be caused by the query optimization process. Using a set of heuristics, they are able to optimize large star-schema queries.

1.2.3 Contributions and Motivation

The central focus of our work presented in this paper is to develop a query processor that presents the DBMS with a *re-formulated* query using GP such that it is fundamentally as equivalent as possible to the original input query such that it is relatively efficient in terms of speed, cost, and accuracy compared to the input query. The resulting query may differ in overall structure from the input query, and the output is a new query plan that returns the same, or approximately the same, results as the input query.

The proposed implementation also differs from most prior evolutionary frameworks. Prior attempts at using evolutionary computation techniques for approximate query answering focused on a single relation or the specific join operations. The GP implementation used in [13] attempts to optimize the different $\bowtie$ algorithms used by an input query.

The operator tree is made up of specific $\bowtie$ operations and the relations being joined, using standard TREE crossover operations to recombine parents to develop children queries. Our implementation is a multi-relational implementation designed to search the schema space of a database for referential ties to the input query's select attributes. In other words, since the input query's select attributes tend to be distributed throughout the database as primary and foreign keys, the same values can occur in other tables under the same, or different, attribute names. In some cases optimization can be accomplished by building a new query that uses the attributes of some other table in the schema, as long as those new attributes can be formally mapped back to the original input query's attributes through key references. Thus we can look deeper into the data and take advantage of certain aspects of the relationships between database entities. Furthermore, using the primary and foreign key relationships of the schema to find and join data allows us to take advantage of the built-in efficiency of using the indices placed on those keys by the DBMS.

To restate the above objectives informally, our GP framework searches different subsets of the schema iteratively, one candidate query at a time, calculating different values such as cost, time, and (in)accuracy. All of these calculations are valuable to the query processor while performing approximations and/or optimizations. This applies to both our GP-based algorithm and to other standard DBMS algorithms. The job of our GP is to exploit all of these interrelationships in such a way that they produce proper optimizations which return a very accurate (as high as 100%) approximation of the correct answer in significantly less time and with a significantly less cost. It is important to note that the actual cost in terms of time for the GP search can be a significant factor. However, if the queries that are submitted to a database are repetitive, then the cost savings over the long term substantially offset this search cost. This is especially useful if there are queries that will be requested often. So, this cost could be a one time cost, as the original queries would only need to be analyzed once. The need to rerun the GP may exist if the data being queried changes significantly or if the data summaries indicate a significant drift in the underlying data statistics. One can explore the idea of evaluating a pool of GP-generated queries over time to study the

approximation characteristics to avoid multiple GP runs.

Chapter 2

Implementation Details

2.1 The AQUAGP Framework

The overall architecture of the proposed framework was described in Figure 1.1. The user-defined query Q_i and the database schema are the available inputs to this system. In traditional databases the input query follows a certain path to execution. The job of the query optimizer is to produce an alternate query, Q_i^o, based on the inherent properties (types) of the query, the metadata available to the optimizer, and the database schema. Our goal was to demonstrate that it is possible to extend the query optimizer further by including a GP-based query optimization routine that lowers the query cost significantly without resulting in inaccurate results.

The simplest approach to developing a GP-based system would be to take the original query Q_i as the input query, evolve several combinations of the query based on this input and the underlying schema, and evaluate each candidate query for its query cost to find the most optimal query plan. There are several problems with this approach. The first and the foremost problem is "semantic mapping". Since the query space is very large for such a problem, every possible query posed to the underlying database can be a candidate query. In fact, one can see why it will be impossible to establish the internal logical consistency of what an optimal query might mean in this context, since we will have to adopt principles of reasoning so complex that their internal consistency will be as open to interpretation as the optimal query itself (somewhat like Gödel's incompleteness proof). On the other hand,

developing a heuristic system that is constrained by semantic considerations and follows a logical query plan when searching for the optimal query within the constrained set of parameters might actually result in an alternative query plan that has effective cost less than the original query plan. It is due to this reason that we use the original input query and the query Q_i^o – the optimal version of some input query, as calculated by a native DBMS system (MS SQL Server 2005 in our case) – as seed queries. The size of the population of candidate queries is a parameter of the system denoted by N, and the candidate queries are denoted by $Q_1, Q_2, ..., Q_{N-1}, Q_N$.

2.1.1 Framework Components

The first component of our GP framework is an automatic query generator. Automatic query generation is a complicated task requiring a significant effort in constraint processing. In order to be able to efficiently generate queries, a database needs to be represented in terms of its metadata. In other words the database's schema needs to be examined to understand the complex relationships between entities. The generation of these initial candidates results in our GP's initial population. The implementation details for the query processing elements needed by the AQUAGP framework are described here:

2.1.2 Representing the Database Schema

Databases typically contain not only a large amount of data but also a substantial amount of metadata (structural as well as semantic) about the database itself. A database can have any number of different tables. These tables can each have any number of columns, also known as attributes. Each attribute, in turn, has its own metadata such as:

a. The relation that the attribute belongs to

b. The attribute's name

c. The data type of the attribute

d. Whether or not it is a primary or foreign key (if it is foreign key, what relation and attribute does it reference.)

e. for numbers the min and max value will be stored

f. for dates the min and max year will be stored

g. and for string types distinct string values in the database can be found easily as well.

Storage of all this metadata by the DBMS is very important. Having information about the structure of the objects in the database makes query generation much easier. Traditionally, a hash table is considered an appropriate data structure for holding such metadata. The table name may be used as the key to the hash, and a list of all of the attributes corresponding to that table can be used as the value. Now it becomes easier to find attributes with the same data type. It is also easier to randomly generate constant values for use in comparisons against the attribute. These constants can be generated based on the data type and min and max values of the attribute, as specified by that attribute's metadata. For our implementation, the database schema gets read at run time from an *XML* file and resides in the main memory for the duration of the framework's operation.

2.1.3 Automatic Query (Tree) Generation

Next we describe the process of automatic query parse tree generation by leveraging the schema's metadata. These new parse trees represent individual queries in the population of our GP. There are several possible query types to categorize queries. In our case we categorize queries based on their plan complexity.

Simple Queries

Query generation in our GP is a multistage process. The first goal is to be able to generate simple queries, and complicate the process incrementally. The framework would support a structure of queries found in most modern query languages:

```
SELECT      { a list of attributes }
FROM        { some list of tables  }
WHERE       { certain conditions exist }
```

Some constraints and assumptions we made are:

- the attribute list is a list of $\{table\_name\}.\{attribute\_name\}$ pairs
- what this implies is that any $\{table\_name\}$ that occurs in the SELECT section of the query must appear within the FROM section of the query.
- any condition that is created in the WHERE section of the query must be a table that was queried in the FROM section of the query.

If these rules are followed, a series of simple queries can be created. A sample *simple query* generated by our GP can be:

```
SELECT DISTINCT a_account_id
FROM   account, trans
```

While this may seem trivial at first, this lays down a strong framework for more complex queries.

Join Queries

The Join operation is a very important operation for a variety of reasons. The Join operation limits the number of rows that are scanned after two tables have been joined. If two tables are combined without a Join, a Cartesian product is computed over the two tables with a resulting set of information of size $M * N$, where M and N are the number of rows in each of the two tables. The Join can select out different features of a table and limit the number of rows returned to significantly less than $M * N$ depending on the $\bowtie$ order and the $\bowtie$ condition.

Because of the importance of the Join operation, it must be used correctly. It is very uncommon that two or more tables are joined together based on attributes other than primary or foreign keys. So Joins should only be done on tables that can be properly joined together on primary keys and their corresponding foreign key attributes, or vice versa. An example of how joins were implemented is shown below:

```
SELECT a_account_id
FROM   account,
       trans,
       loan
WHERE  a_account_id = t_account_id
AND    a_account_id = l_account_id
AND    t_account_id = l_account_id
....
```

In the above example, three tables are used in determining the data returned by this query. The Joins wire together the tables based on the primary and foreign key attributes. In this example, $account.a\_account\_id$ is the primary key. The trans and loan tables are joined with account based on their corresponding foreign key attributes, $t\_account\_id$ and $l\_account\_id$, respectively. Also, it can be seen from the above examples that queries can become quite complex, or they can be extremely simple. The complexity of the resulting query tree is a function of the randomness of the generation process.

The Where Clause

An important part of the query generation process is being able to specify the selection conditions to reflect the type of information desired. Suppose the person issuing the query desires to obtain the ids of all employees whose $salary > \$50,000$ or some other specific search criterion as expressed by a range or projection query. Using a WHERE clause allows the query issuer to specify the search criteria necessary to selectively obtain only the desired data.

Heuristically generating a **valid** search criteria for a query is not trivial. We implemented a few operations that can be performed to compare attribute data. These operations included range finding operations using BETWEEN clauses, substring matching operations using LIKE clauses, and logical operations for comparing values using predicate clauses. In some cases, we may want to narrow down the information retrieved. For example, all the people who made a transaction between such and such a date (i.e. BETWEEN 1991 10 01 AND 1991 12 31), or all people that had a certain account balance (i.e. balance $\leq$ 2000), or all people whose city contains a certain substring (i.e. my city LIKE '%city%'). As mentioned above, predicate clauses can be used individually to compare a records attribute values to some constant (i.e. 5000) or can be used in conjunction with another predicate clause to create a range finding operation similar to BETWEEN (salary $\geq$ 50000 and salary $\leq$ 70000). In our framework, the predicate clauses can use any of the following logical operators: =,<>,! =,>,<,! >,! <,>=,<=,! >,! <.

Note that we did not use/develop a native query analyzer similar to a commercial database system. Hence, our GP does not implement the entire collection of possible features available for the WHERE clauses; this is a constraint of the current system that we intend to address in the future. We attempted to identify the most important clauses for a proof-of-concept implementation. The addition of a feature to our GP's WHERE clauses required radiating the changes throughout our entire implementation of random query generation, crossover, and mutation process since the validation checks percolate through all three operations.

Nested Queries

The existing framework made implementing nested queries a somewhat trivial task. By extending the existing generate query function to take a list of query attributes, a nested query could be generated by calling the same generate query function with a different set of attributes. Therefore, randomly many levels of nesting can occur through recursion. However, in our implementation, the depth of the recursively built nested query trees is

limited to a specific value, which is a parameter of the framework. It is our belief that allowing arbitrarily deep nested queries leads to the same problems as bloating in GP: high complexity and exhaustion of memory space; in addition it can cause DBMS constraint problems, all while not providing for much of an approximation or optimization benefit over an implementation that has a pre-set limit on nesting levels for queries.

2.1.4 A Probabilistic Evaluation of GP's Correctness (and Capacity) to Produce Valid Approximate Queries

Before going any further, it is important to note the different types of queries that AQUAGP can currently handle reasonably well, and the rationale behind this. Let's assume that using this GP framework, there is a certain probability that any input query will converge to an approximate answer. Can we characterize such a probability function to estimate how or when the proposed system is likely to perform well? In this section we briefly develop a theoretical framework that can act as a step in this direction. We have the following definitions:

Q_i, the input query

Q_i^o, the optimal form of Q_i given by the DBMS

$Q_i^o.r$, the set of results returned by Q_i^o

DSA, **D**omain of the **S**elect **A**ttributes, In other words, all possible values for the select attributes.

$Contiguousness(Q_i^o.r, DSA)$, the contiguousness or sequentiality of the data in $Q_i^o.r$ throughout DSA.

PSA, **P**robability of **S**uccessful **A**pproximation

Using the above definitions, we attempt to formally define **PSA** in Equation 2.1:

$$PSA = \frac{|Q_i^o.r|}{|DSA|} \times Contiguousness(Q_i^o.r, DSA) \quad (2.1)$$

Without a formal proof, it is difficult to ascertain the correctness of Equation 2.1 is perfect. However, what it does say is that the probability of successful approximation of our framework depends on the ratio between the number of desired records for $Q_i^o.r$ and the number of possible records for the select attributes (DSA), as well as the contiguousness of the desired records throughout the set of all possible records. An exact approximation of the contiguousness is another problem altogether, but we can see that these two parameters are not entirely independent of one and other. More over, the probability of successful approximation is a function of the contiguousness of the data.

The ratio $\frac{|Q_i^o.r|}{|DSA|}$ is important, because it defines the precision needed for a decent approximation of the results. If the ratio is high (i.e. close to 1), then the precision is low and thus easier to attain. If the ratio is low (i.e. close to 0), then the precision is high and thus more difficult to attain. For example, if there is a relation with 1,000,000 records, and the attribute being selected is the primary key of the relation in question, the DSA is 1,000,000 records. Now suppose that $|Q_i^o.r|$ is only 1,000 records; the ratio is 1/1000, making the precision quite high. On the other hand, if $|Q_i^o.r|$ is 750,000 records, the ratio is high (75/100), making the precision fairly low. Our implementation works especially well for queries requiring a low precision. In general, it is easier to define a query that returns a larger subset of the data, than it would be to define a query that returns a small "precise" subset of data.

The ratio described above combines the correlation between the projection attributes(s) and the select attribute(s) to influence the contiguousness of $Q_i^o.r$ in the DSA. The contiguousness of a subset of records represents the spatial dispersion of the records throughout the DSA. To go back to the example from above, with the DSA being 1,000,000 records, if $|Q_i^o.r|$ is 1,000 records, they can either be contiguous (sequential) or non-contiguous (non-sequential) to some degree. If the desired records are the first 1,000 records or the last 1,000 records, or even a range of 1,000 records located somewhere in the middle of a data set,

then the desired data is considered contiguous and should be easy for our framework to approximate. On the other hand, if the 1,000 records are dispersed throughout the 1,000,000 DSA records, they are considered non-contiguous (non-sequential) and thus harder for our framework to approximate. It is important to note that while the ratio is an easily calculated field, the contiguousness of a given queries result set is not easily calculated or even understood without the use of advanced clustering algorithms, or other data mining related algorithms. However, that is out of the scope of our framework, but is certainly worth mentioning. This is troublesome in that the contiguousness of the data returned plays a primary role in determining whether this framework can approximate a query.

We will now discuss the importance of the correlation between the projection attributes(s) and the select attributes(s). The correlation depends on how contiguous the values of the select attribute(s) are when the projection attributes(s) don't represent the same values (i.e. they are not related by keys, perhaps the select attribute is the value of a unique id, but the projection is based on salary, and the two may have nothing to do with each other). Simply put, these sorts of queries produce non-contiguous data; it could be like having to pick out 1,000 specific needles from a pile of 1,000,000 needles in a haystack quickly and efficiently. An example of this can be seen in Figure 2.1. Essentially, there is no correlation between partsupp.ps_partkey and part.p_retailprice. It is not to say that our framework could not find an approximate answer to this query; what we are saying is that the probability of being able to do so is quite low, making it more difficult to approximate this kind of query. However, when the select attributes are closely tied to the projection attributes (i.e. they are the same attribute of primary and foreign key references to each other) they tend to produce subsets of contiguous data. These types of queries tend to have a high probability of convergence. An example of a query that our framework works better with can be found in Figure 2.2.

From our experimentation, we have concluded that the contiguousness of the data is more important than the ratio of the result sets. Even small subsets of records can be approximated if those records are relatively contiguous. Conversely, some larger subsets of

```
SELECT DISTINCT partsupp.ps_partkey
FROM part, supplier, partsupp
WHERE part.p_partkey = partsupp.ps_partkey
AND supplier.s_suppkey = partsupp.ps_suppkey
AND part.p_retailprice ≥ 950
AND part.p_retailprice ≤ 1000
```

Figure 2.1: A query that would have a low probability of convergence due to the (non)correlation of select and project attributes.

```
SELECT DISTINCT supplier.s_suppkey
FROM lineitem, supplier, nation
WHERE lineitem.l_suppkey = supplier.s_suppkey
AND supplier.s_nationkey = nation.n_nationkey
AND supplier.s_suppkey > 3000
AND supplier.s_suppkey < 7000
```

Figure 2.2: A Query with three joins and a large projection of the data based on a query attribute. This is a suitable query for approximation by AQUAGP.

records can be difficult to approximate if the data is non-contiguously distributed throughout the entire DSA.

Figure 2.2 is a good candidate for approximation using AQUAGP, because it involves large $\bowtie$s, supposing that the lineitem table has number of rows $\geq$ 6,000,000, with the other two tables having significantly less rows of info proportionally. Apart from the joins, the projection of the data is on the query attribute supplier.s_suppkey, and is looking for a range of sequential data. These two factors make Select, Project, and Join queries good candidates for approximation using AQUAGP. Figure 2.3 shows a possible approximation generated using AQUAGP.

```
SELECT DISTINCT partsupp.ps_suppkey
FROM partsupp, part
WHERE partsupp.ps_suppkey = part.p_partkey
AND partsupp.ps_suppkey BETWEEN 2975 AND 6800
```

Figure 2.3: A Query found by AQUAGP to approximate the answer to the query found in 2.2, with significantly less cost and an approximate result.

Query	time	number of results
query1	30 seconds	3999
query2	5 seconds	3826
query1 $\bigcap$ query2	n/a	3800
query1 - query2	n/a	199
query2 - query1	n/a	26

Table 2.1: statistics to verify that the query in figure 2.2 is much less optimal than the query shown in 2.3, and that the query in 2.2 can be approximated using the query in 2.3.

We note here that the query found by AQUAGP computes a much smaller Join on the partsupp table, which has a number of rows $\geq$ 800,000 and the part table which has a number of rows $\geq$ 10,000. It also has the same kind of projection, but instead of a two-phase projection it uses the BETWEEN operator.

Based on accuracy, cost, and time spent to execute the query, we can see that the query in Figure 2.2 is significantly less optimal than the query in Figure 2.3. The statistics for these two queries are shown in Table 2.1.

The data in Table 2.1 indicates that the intersection is rather large, 3800 out of 3999 possible results. It also shows that the extra results are minimized to 26, and that it is missing 199 results. This query could be more finely tuned, if the expected accuracy of the system is raised. However, even with the accuracy setting as is, we are still returning approximately 95% of the desired results, without a large number of extra results.

Aggregate Queries as Bad Choices for GP.

Aggregate Queries, such as queries that may select an average (avg), a sum, or a count of a column will not perform well using GP. These queries, generally produce very specific data that is normally non-contiguous in nature. As an example, we take queries from the TPC-D benchmark used in [2]. Figure 2.4 shows a query that uses aggregates in the select clause, and uses non-contiguous data as the projection:

The query in Figure 2.4 computes the average price of products delivered by suppliers

```
SELECT avg(l_extendedprice)
FROM customer, order, lineitem, supplier, nation, region
WHERE c_custkey = o_custkey
AND o_orderykey = l_orderkey
AND l_suppkey = s_suppkey
AND c_nationkey = s_nationkey
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = [region]
AND o_orderdate ≥ [startdate]
AND o_orderdate < [enddate]
```

Figure 2.4: A query that will not approximate well using AQUAGP.

in a nation to customers who are in the same nation. This query also has three variables, [region], [startdate], and [enddate]. The [region] variable restricts which regions the suppliers can be from. As well, [startdate] and [enddate] restrict the focus of the query to a certain time interval. As mentioned in [2] computing approximate aggregates on multi-way joins is a very hard problem. It is also a very hard problem for GP. Based on our discussion in section 2.1.4 it should become quite clear why our GP model will not have a high probability of being able to compute the approximation for such a query. First of all, the select attribute is avg(l_extenedprice), which is not primary or foreign key and does not appear as an attribute in another relation. So our GP query generator would not be able to exploit the attribute from other tables. However, it can always take the l_extendedprice from the lineitem table, but the big problem faced is that the lineitem table may be quite large, and the GP tries to exploit relationships amongst different tables that share attributes. The other major problem is the non-contiguousness of the data that can be returned. For each region, this query returns an avg(l_extendedprice), on a specific time interval. There is a non-correlation between the data that is being projected and the data that is being selected. This is a major problem in the GP implementation of a query processor. Figure 2.5 shows another aggregate query, a much simpler one that would still be troublesome for the GP, as found in [2] and the TPC-D benchmark data set.

Again, the problem that is faced in Figure 2.5 is the non-correlation between the select

```
SELECT avg(l_quantity)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey
AND o_orderkey = F
```

Figure 2.5: A simpler aggregate query that would still pose a problem to AQUAGP.

attributes and the projection attributes. Not only do the queries in Figures 2.4 and 2.5 use aggregates, but they produce non-contiguous data. We believe that queries like these two are not impossible to approximate using GP, and AQUAGP may even be able to find an answer. We are able to recompute large joins to find a more optimal join scheme for a query. But non-contiguous data still poses a problem for GP and aggregates make dealing with non-contiguous data even more difficult. Possible solutions to this problem are to use data summarization techniques such as Join Synopses, as does the AQUA framework described in [2]. Another alternative is to use two-dimensional histograms for data summarization as described in [10]. Two-dimensional histograms seem to be a good alternative approach to dealing with independent rows in a database. The construction of a two-dimensional histogram can give insight to non-correlated data that may be overlooked by the GP approach proposed here.

2.2 GP-based Candidate Query Generation: Parameters and Issues

2.2.1 Fitness Evaluation

Fitness evaluation of any query Q_x is based on a few different pieces of information about the query. The following statistics about a given query Q must be derived:

1. The execution time of the query (in seconds).
2. The estimated cost of the query, which is based on:
   (a) The estimated CPU time.

(b) The estimated IO time.

(c) The estimated number of rows a $\bowtie$ (join) will cause.

(d) The estimated total subtree cost.

3. The resulting set of information that the query returns when executed.

From the metadata, query cost estimates, and other such information provided by the DBMS, we developed a function to evaluate the GP fitness such that the value returned is a normalized value on the interval $(0, 1]$ where a fitness of 1 indicates the best possible individual in a given population and 0 is the worst. During query evolution, two quantitative criteria can be minimized as discussed earlier. These entities, time and cost, are denoted as T_{Q_x} and C_{Q_x} respectively. A third qualitative measure is a query's accuracy in comparison to the optimal query, denoted as τ_{Q_x}.

Calculating the accuracy of all generated candidate queries is computationally infeasible. In fact, even for a reduced set of queries, a large portion of the randomly generated queries take an extremely long time to execute. Thus, the initial set of queries, whose accuracy the GP will examine, must be pruned. To facilitate pruning, a "costFitness" value is introduced. costFitness can be used to weed out the poorly formed queries that would take too long to execute. The costFitness formula is comprised of four components as follows:

$$ioFit(Q_x) = (Q_i^o.io/(Q_x.io + Q_i^o.io)) \tag{2.2}$$

$$cpuFit(Q_x) = (Q_i^o.cpu/(Q_x.cpu + Q_i^o.cpu)) \tag{2.3}$$

$$rowFit(Q_x) = (Q_i^o.row/(Q_x.row + Q_i^o.row)) \tag{2.4}$$

$$streeFit(Q_x) = (Q_i^o.stree/(Q_x.stree + Q_i^o.stree)) \tag{2.5}$$

$$costFitness(Q_x) = \tag{2.6}$$

$$((ioFit(Q_x) \times W_1) + (cpuFit(Q_x) \times W_2) \tag{2.7}$$

$$+ (rowFit(Q_x) \times W_3) + (streeFit(Q_x) \times W_4))/100 \tag{2.8}$$

where W_i = the weight of that particular fitness in the costFitness calculation, and $W_1 + \ldots + W_4 = 100$.

The costFitness value of any query lies in the interval $(0, 1]$. It is an estimate of how fast the query would take to complete, if it were actually executed. The framework has a hard-coded costFitness threshold, ρ ($\rho = 0.7$ in our implementation) which is used to prune queries whose $costFitness < \rho$.

Given a pruned population of test queries, accuracy is calculated by examining the results returned ($q_x.r$) from both the current and optimal queries in the following manner:

$$inaccuracy(Q_x) = |Q_i^o.r \bigcup Q_x.r| - |Q_i^o.r \bigcap Q_x.r| \tag{2.9}$$

$$accuracyQ_x = \frac{|Q_i^o.r|}{|Q_i^o.r| + inaccuracy(Q_x)} \tag{2.10}$$

The initial query result set, $Q_i^o.r$, is the standard by which accuracy is measured. Inaccuracy of the test query, Q_x, can be defined, concretely, as the set of tuples which are returned by either query but do not appear in the intersection. Clearly, the lower the value of $inaccuracy(Q_x)$, the more similar the result sets. The results of measuring inaccuracy are normalized on the interval, $(0, 1]$, in the $accuracy(Q_x)$ function.

A similar procedure is defined for determining the actual execution time of the query. Each query's execution time is measured and tabulated, and an efficiency function is used to derive a normalized value over the interval, $(0, 1]$:

$$efficiency(Q_x) = \frac{Q_i^o.t}{Q_i^o.t + Q_x.t} \tag{2.11}$$

The accuracy and efficiency of a particular query are formalized in the following equation:

$$Fitness(Q_x) = accuracy(Q_x) \times efficiency(Q_x) \tag{2.12}$$

Any query whose fitness exceeds 0.5 is considered "better than" the initial query, ${Q^0}_i$, as $Fitness(Q_i^o) = 0.5$. The strength of this fitness equation is that it favors neither criterion at

the expense of the other. Queries which show perfect performance in one area, and abysmal performance in the other, will not be ranked higher than the initial query.

2.2.2 Query Tree Crossover

Next, we describe the GP crossover operation as implemented for generating approximate query plans. We assume all queries within this system have the following structure:

```
SELECT    <s_list>
FROM      <f_list>
WHERE     <w_list>
```

Any $< *\_list >$ is a list of valid strings that can be placed in that part of the query to generate valid SQL syntax and semantics. During SELECT list crossover, one of the biggest obstacles we had to overcome was making sure that attributes of a certain data type were compared only with other attributes of the same data type that corresponded to the same data in the schema. Although everything cannot be restricted, it is necessary to impose restrictions on how pieces are compared and how much comparing actually gets done. From two parent queries, the crossover operation does its work in three main steps:

1. Iterate for the number of SELECT attributes (each parent will have the same number of SELECT attributes) and at each iteration randomly take the corresponding attribute from one of the two parents. Create a new $< f\_list >$ and fill it with all of the required tables corresponding to the new set of SELECT attributes in the new $< s\_list >$.

2. Combine the $< w\_list >$'s from both parent queries and randomly select a subset of that list to create a new $< w\_list >$ for the offspring. If not already added, append any necessary parent tables to the $< f\_list >$ to accommodate the WHERE clause.

3. Construct the new child query from the newly generated $< s\_list >$, $< f\_list >$, and $< w\_list >$.

This crossover process results in evolving semantically equivalent (albeit in the limited context of this implementation) candidate queries.

2.2.3 Query Tree Mutation

We developed the GP mutations to focus exclusively on the WHERE clause portion of the queries. The rationale is that the WHERE clause is more than likely the key determining factor in how accurate a randomly generated query will be. Each mutation operation consists of the following steps:

1. Separate the WHERE clauses into the individual clause types (i.e. LIKE, BETWEEN, predicate). Ignore wiring (join defining) and nested clauses.

2. Randomly generate new values for the LIKE, BETWEEN, and predicate clauses of the query. This must be done with consideration of the data type and range of the attributes being mutated. Both measures are available via metadata from the schema.

3. Randomly insert or remove NOT operators to negate the current functionality of each clause.

4. Randomly select AND and OR operators to be placed between the clauses. We favored AND's in this step to decrease the chances of creating overly complex, and therefore inefficient, queries.

Chapter 3

Experiments and Results

3.1 Experiments and Results

The implementation of our framework is written using C#. The database we used was Microsoft SQL Server 2005 Beta 2 June CTP, which is a commercial Database Management System consisting of native query parsing and cost estimators. We used Visual Studio 2005 Beta 2 as the development platform on an IBM PC with an Intel Pentium 4 processor and 1GB RAM. The two datasets we used include: (a) The PKDD Cup Multi-relational transaction dataset and (b) the TPC-H benchmark dataset containing business transaction data.

Recall that the idea is to efficiently evolve new queries, evaluate their fitness, and determine if they meet the accuracy criterion as approximate queries for the original queries. In this section, we discuss the statistical analysis of the results we collected. For simplicity of explanation, we divide the discussion of results in two sections, a section containing results from the generational GP implementation and the other containing results from the steady-state GP implementation. For both of the GP strategies, we decided to evolve queries varying the GP parameters for the following types of query categories: (i) 2-table join without projection, (ii) 2-table join with projection, (iii) 3-table join without projection, and (iv) 3-table join with projection. The complexity of various queries broadly depends on the join condition and on the constraints if projections can/need be achieved pre- or post-join. Queries over both the datasets mentioned above were computed for all four of the above

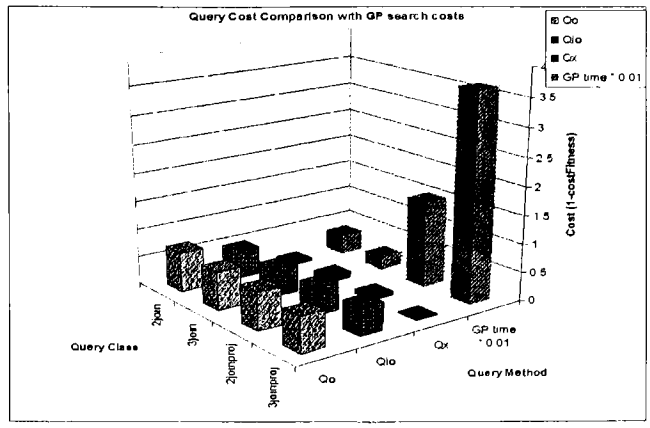


Figure 3.1: Graph of Cost Reduction Comparison for Differing Query Types. This graph shows a comparison between the cost of executing the original input query, Q_i, the cost of executing the optimal form of that query, Q_i^o, and the cost of executing our GP's best candidate query, Q_x. The costs compared were all calculated for queries Q_x which return results with an accuracy of 80% or greater. Results are shown for queries of the following types: 2-table join without projection, 2-table join with projection, 3-table join without projection and 3-table join with projection.

categories, and results were evaluated. Figure 3.1 describes the comparison between the optimal query, as proposed by the query plan analyzer built into the MS SQL Server 2005; and the cost of executing the GP-generated, best-fit candidate query. The accuracy threshold was set to 80% for this plot. As can be seen in Figure 3.1 and Figure 3.2, the queries that the GP finds are higher in cost fitness than the input query. The higher the cost fitness, the lower the actual cost, so the GP is finding a much faster query with an approximate answer. Figure 3.2 shows the accuracy of Q_x. Another thing to note in figure 3.1 is the significant amount of time needed to perform the GP. However, as we mentioned earlier this is a one-time cost. Once the database engine has performed these optimaztions, it can store the values of Q_x and use this query plan as an option the next time the query gets requested. It would not be optimal to run the GP every time a query is submitted to the database. GP is not assured to converge, as well it is not assured to find an optimal answer either. So, we can see that the first time the query is submitted to the database engine, it may take some time to process and come up with a good approximation, but the queries

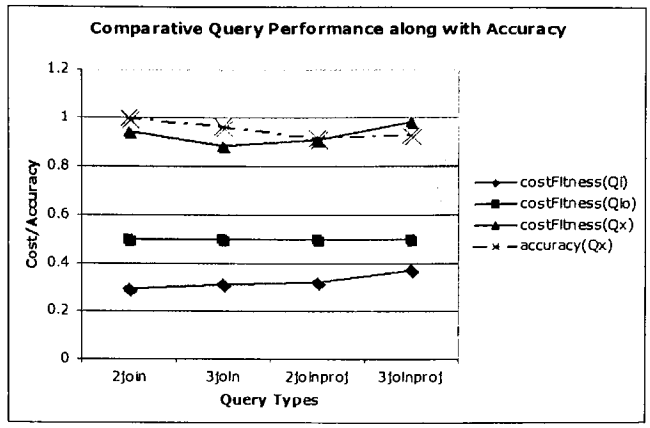


Figure 3.2: Graph of comparative query performance along with the accuracy of the queries returned by the GP. The comparison of queries is the same as in figure 3.1

found by the GP are substantially faster than the input query.

3.1.1 Algorithms

We employed two GP algorithms in our studies; the following discussion will describe the two algorithms we used:

1. Generational GP algorithm

2. Steady-State GP algorithm

The information in table 3.1.1 shows the default parameters used in the experiments.

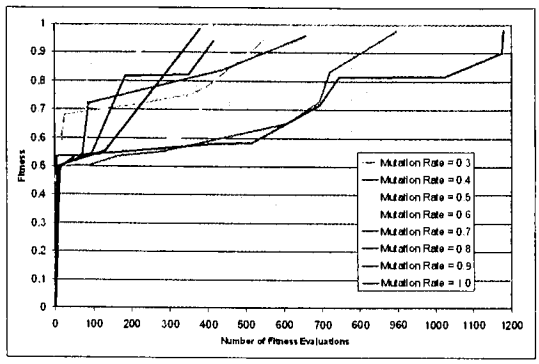
Generational GP Implementation

In this discussion we will talk about the generational GP implementation. Table 3.1.1 contains the GP parameters set as defaults in all test runs for the generational GP implementation. For generational GP mid (0.5) to high (0.99) rates of mutation lead to faster convergence compared to lower mutation rates. As shown in Figure 3.3(a) the increase in fitness over number of evaluations indicates progress towards convergence. It can be noticed that average fitness levels such as 0.5 were quickly achieved irrespective of the

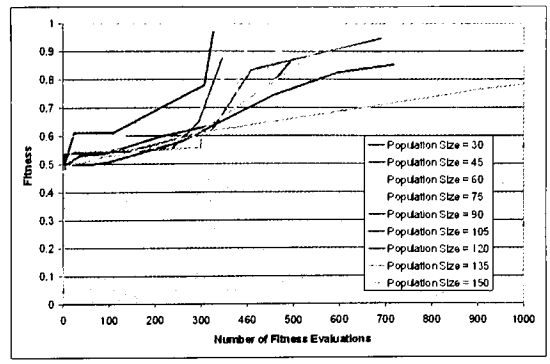
Generational GP Parameters	Value
Population Size	100
Generations	20
Selection Size	10
Percentage of Copies	20%
Percentage of Crossovers	80%
Desired Accuracy	.85
Mutation Rate	50%
W_1, W_2, W_3, W_4	25
Steady-State GP Parameters	**Value**
Population Size	100
Loops	20
Selection Size	5
Desired Accuracy	.85
Mutation Rate	50%
W_1, W_2, W_3, W_4	25

Table 3.1: Default parameter values for Generational and Steady-State GP implementations.

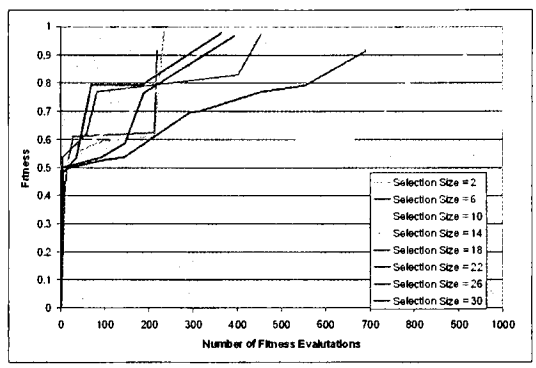
mutation rates. Since fitness is a measure proportional to accuracy of the tuples returned, our understanding is that the generational GP-based approximate query answering system finds an average-fit query within the initial population most of the time, and it takes some effort later on to fine tune the candidate queries towards the most optimal query. This fact is also apparent in Figure 3.3(b), which shows the variation of fitness as a parameter of the population size. Large population sizes tend to be slower to converge, but this might be a dataset-specific occurrence and could not be conclusively proven. Figure 3.3(c) demonstrates the effects on fitness of modifying the selection size and suggests that keeping too few or too many parents in the newer population leads to sub-optimal local minima.



(a) mutation



(b) population



(c) selection

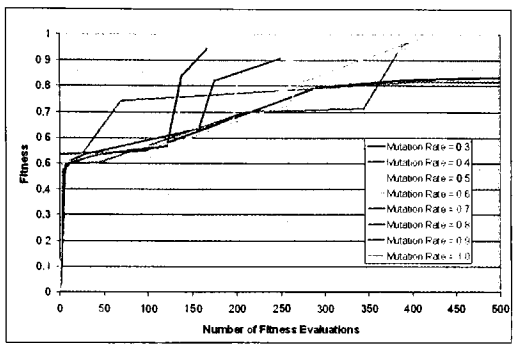
Figure 3.3: The effects of different parameters on the generational algorithm

3.1.2 Steady-State GP Implementation

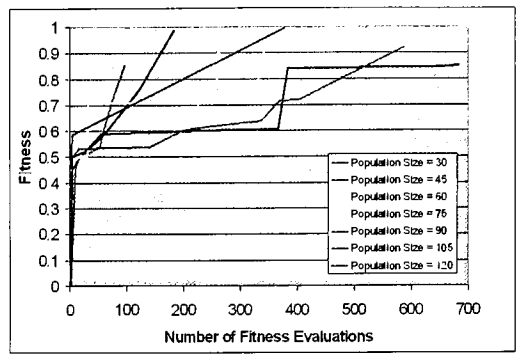
In this discussion we will talk about the steady-state GP implementation. Table 3.1.1 contains the GP parameters set as defaults in all test runs for the steady-state GP implementation. As per the generational algorithm mid (0.5) to high (0.99) rates of mutation lead to faster convergence as compared to lower mutation rates. We can see a steady progression of the best fit individual in our population using mid to high mutation rates. We observed

the need for higher mutation rates to search for the appropriate constants and logical operators that generate a well formed query. In all cases, we see in Figure 3.4(a) that an average individual, generally with fitness of 0.5 within the initial population. However, these individuals' fitnesses indicate that they are efficient enough to execute, but not accurate enough to be considered fit for our purposes. It is the mutation operator that leads to better individuals in later iterations of the steady-state GP. Figure 3.4(b) shows that convergence varies with population size. Finally, Figure 3.4(c) demonstrates the use of different selection sizes on the populations. When the selection/tournament size is too low (2), convergence may not occur in the specified number of iterations, and too high a selection size (>18) leads to drastic overfitting of non-optimal minima. Figure 3.4(c) suggests that selection size (6-10) leads to fast convergence with a higher accuracy.

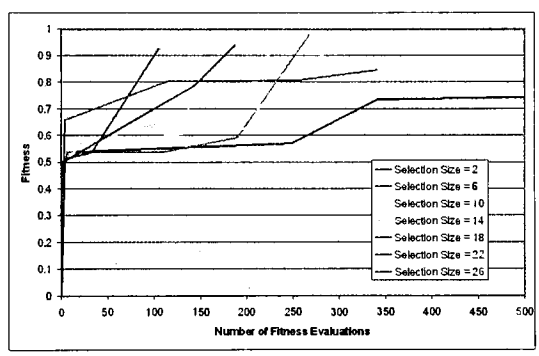
Overall, we believe that it is safe to say that neither algorithm, Steady-State or Generational, was any better than the other; there are drawbacks and advantages to both algorithms.



(a) Mutation



(b) Population Size



(c) Selection

Figure 3.4: The effects of different parameters on the steady-state algorithm

Chapter 4

Final discussion

4.1 Summary and Future Work

In this paper we described a novel approach for approximate query answering using Genetic Programming. We evaluated different GP techniques for generating approximate queries and then developed a cost model to determine how fit a given candidate query is compared to the original query in terms of the results returned. The framework described in this paper handles a variety of queries involving relational joins, multi-joins, projections, and select project joins.

Approximate Query Answering is becoming more and more important in many fields, including sensor networks [4]. In this paper, we proposed a new approach to address this complex challenge. We encountered several issues such as non-contiguous data being returned from a query based on the non-correlation between select and project attributes. Another major issue that we faced is computing complex aggregates over large joins. These two problems are interdependent and currently open for future research, as we highlight in the paper. Data summarization techniques can be used to deal with issues such as join synopses, etc. The need to rerun the GP may exist if the data being queried changes significantly or if the data summaries indicate a significant drift in the underlying data statistics. One can explore the idea of evaluating a pool of GP-generated queries over time to study the approximation characteristics, to avoid multiple GP runs. The current framework can be

extended to include an enhanced tree-generation module and a semantic constraint verification module that will allow greater flexibility with respect to the queries that the framework can handle. Other models of GP, such as linear GP, might be suitable techniques for query generation to compare to tree-based GP for searching through the query space. Based on the speed, cost, and accuracy we derived methods to show that an evolved query can result in an accurate approximation of an input query. To conclude, we demonstrated that tree-based GP can be successfully utilized to advance the state-of-the-art approximate query processing when used in conjunction with a query cost estimation framework.

Bibliography

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 574–576, 1999.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 275–286, 1999.

[3] S. Chaudhuri. An overview of query optimization in relational systems. *Symposium on Principles of Database Systems Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, 1998.

[4] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. *The VLDB Journal*, 2004.

[5] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.

[6] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. Database Syst.*, 23(2):113–157, 1998.

[7] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[8] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[9] C. Lee, C.-S. Shih, and Y.-H. Chen. Optimizing large join queries using a graph-based approach. *Knowledge and Data Engineering, IEEE Transactions on*, 13(2):298–315, 2001.

[10] H. T. A. Pham and K. C. Sevcik. Structure choices for two-dimensional histogram construction. *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 13–27, 2004.

[11] S. Shekar, B. Hamidzadeh, A. Kohli, and M. Coyle. Learning transformation rules for semantic query optimization: adata-driven approach. *Knowledge and Data Engineering, IEEE Transactions on*, 5(6):950–964, 1993.

[12] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

[13] M. Stillger and M. Spiliopoulou. Genetic programming in database query optimization. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 388–393, Stanford University, CA, USA, 28–31 1996. MIT Press.

[14] W. Sun and C. Yu. Semantic query optimization for tree and chain queries. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):136–151, 1994.

[15] Y. Tao, Q. Zhu, C. Zuzarte, and W. Lau. Optimizing large star-schema queries with snowflakes via heuristic-based query rewriting. *IBM Centre for Advanced Studies Conference Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 279–293, 2003.

[16] G. Weddell. Selection of indexes to memory-resident entities for semantic datamodels. *Knowledge and Data Engineering, IEEE Transactions on*, 1(2):274–284, 1989.

[17] H. Yoo and S. Lafortune. An intelligent search method for query optimization by

semijoins. *Knowledge and Data Engineering, IEEE Transactions on*, 1(2):226–237, 1989.

[18] C. Yu and W. Sun. Automatic knowledge acquisition and maintenance for semantic queryoptimization. *Knowledge and Data Engineering, IEEE Transactions on*, 1(3):362–375, 1989.

[19] C. Zhang, X. Yao, and J. Yang. An evolutionary approach to materialized views selection in a datawarehouse environment. *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, 31(3):282–294, 2001.