Automatic Query Generation

Bruno Castro da Silva and Philip Thomas

Technical Report UM-CS-2010-074

Computer Science Department
University of Massachusetts Amherst
{bsilva, pthomas}@cs.umass.edu

May 12, 2010

Abstract

In this paper we present novel techniques for Automatic Query Generation (AQG). Query languages like Query by Example (QBE) allow programmers to create queries without the having to learn more complex languages like SQL. However, they still function in the same manner, with the programmers' input being translated directly into a form of relational calculus or relational algebra. In this work, we take the idea of querying by examples to its extreme, in which the user provides example tuples that should be returned and example tuples that should not be returned. Though this may seem similar to the recent Query by Output (QBO) paradigm, they are fundamentally different. The QBO system assumes that the user already has access to a query that performs the desired lookup, while AQG makes no such assumptions. The AQG techniques presented in this paper search the space of Select-From-Where (SFW) SQL queries to find the simplest one that best matches the example tuples provided. Thus, the difficulty of searching the space of relational algebra or calculus queries for the desired query is relegated to the AQG engine rather than the user. The benefit of this novel query language is that it is extremely simple, which allows unskilled workers to generate SQL queries.

1 Introduction

Query languages such as Query-by-Example (QBE) [4] allow unskilled workers to write basic queries. However, contrary to its name, QBE is actually a query language much like SQL, in which the user must know the structure of the relations and directly specifies what he or she wants the query to do. For example, consider a query that returns all people over a certain age. A QBE query will still explicitly encode this contraint. In this work, we use automatic query generation (AQG) to allow the user to create queries purely from examples. That is, the user provides *desired* tuples as well as a list of $undesirable$ tuples. We then search for the simplest SQL query that generates as many of the desired tuples as possible without including the undesirable tuples.

Thus, AQG has some commonality with Query by Output (QBO) [3], where, for a database D, the user provides $Q(D)$, the output of some query Q, and the QBO algorithm produces a set of Instance-Equivalent Queries (IEQs), or approximately equivalent queries, $Q_1, Q_2, ...Q_n$. When using QBO, the user must already have access to a query Q that accomplishes the desired result, whereas AQG is a method for constructing this initial Q. This difference will be further discussed in the Prior Work Section (3).

The primary benefit of AQG is that the user needs minimal knowledge of databases and of the specific relational schema being used. For example, the query to select people over a certain age would look the same regardless of the structure of the relational table backend. So, if an elementary school student wished to find everyone in her grade, she could provide a list of a few students she knows are in her grade, and a few that are not. The AQG engine would then generate the query for her. Additionaly, the AQG can be used as a sort of search engine. For example, a student may enter a list of his friends and people that he doesn't like. Ideally, the AQG would return a list of other people that he might like. Another perk of the AQG is that the query can be returned to the user for analysis. Thus, a skilled user could enter a list of people and the AQG would return a query that determines what separates that group from others, such as common interests.

In Section 2 we describe the problem that AQG tackles and in Section 3 we discuss its relation to prior work. In Section 4 we describe our methods, followed by the results in Section 5. We then discuss these results and conclude in Sections 6 and 7 respectively.

For the remainder of this paper, we will consider the example relational schema:

Person: (PID, Name, Age, JID, Gender)
Job: (JID, Title, Salary, Hours)
Manager: (MID, SID),

where MID and SID in the **Manager** relation correspond to PIDs in the **Person** relation.

2 Problem Statement

Provided a set of $desired$ and a set of $undesirable$ tuples, we wish to find the corresponding SQL query that the user had in mind. This can be formalized by assuming the user wishes to find $Q(D)$, the output of query Q on database D, but does not know the structure of D, and is unable to generate Q. However, the worker is able to generate a small set of desired tuples $desirable \subseteq Q(D)$ and undesired tuples $undesirable \subseteq \neg Q(D)$, where $|desired| << |Q(D)|$ and $|undesirable| << |\neg Q(D)|$. The goal is to construct a Q' such that $Q'(D)$ contains the tuples in $desired$ but not the tuples in $undesirable$. In order to do this, we will assume that the desired query is a tradeoff between simplicity and correctness. That is, if a very simple query exists that produces a nearly correct result, and a very complex query produces the exact tuples specified, the simple query will be preferred. However, this exact tradeoff will be left as a tunable parameter.

Consider the example **Person** relation

PID	Name	Age	JID	Gender
1	Alice	25	1	Female
2	Bob	30	2	Male
3	Charlie	24	1	Male
4	Debra	17	8	Female
5	Eve	15	7	Female
6	Frank	18	2	Male
7	Gretta	27	3	Female
8	Henry	34	2	Male
9	Ilene	12	5	Female
10	John	32	3	Male

A user may wish to find the names of all the people older than 23. A possible query to find this could be the $desired$ table:

$Desired$
Bob
Gretta
Alice

and the $undesirable$ table:

$Undesirable$
Debra
Frank
Eve

The simplest SQL query to generate the tuples in the $desired$ table without generating any of the tuples in the $undesirable$ table is:

```
SELECT Name
FROM Person
WHERE Person.Age > 23
```

This is therefore the desired query, though other more complex queries exist that would still return the correct tuples, such as:

```
SELECT Name
FROM Person
WHERE Person.Name = ”Bob”
    OR Person.Name = ”Gretta”
    OR Person.Name = ”Alice”
```

The goal of this paper is to implement a AQG engine and determine how complicated a query can be such that the target query can be reliably retrieved, when provided with small $desirable$ and $undesirable$ sets. In the following section, we discuss prior work, and clarify how previous systems differ from the AQG engine described in Section 5.

3 Prior Work

As mentioned in the introduction, Query by Example (QBE) systems [4] are the most similar to the AQG engine presented herein. The goal of QBE was to allow unskilled workers to create database queries without having to learn a more complicated language like SQL. To an extent, QBE achieved this by creating a simple interface for constructing queries. However, QBE still requires a user to understand the structure and relationships between tables in the database. In fact, when designing a QBE query, a user is directly filling in fields within a corresponding SQL query. The primary flaw with QBE is its foundation in relational calculus. The AQG engine removes these requirements and simplifies query writing beyond the level of a QBE statement.

Additionally, QBE is incapable of representing more complicated queries. For example, an unskilled worker would not be able to create a QBE query, on the relational scheme proposed in Section 1, which returns the names of all people who have higher salaries than their managers. Though this query is not within the space of possible QBE queries, it is within the scope of possible AQG queries. However, this is a more complex query, as it requires all three tables be joined, which results in a large search space of SQL queries, making difficult for the AQG to find.

Though QBE was initially proposed in 1975, there have been few improvements upon the original concept. Most additional research focusses on extensions to searching for images by example [2] and generating query by example methods for creating relational schemas rather than querying them [5].

Though superficially similary, Tran, Chan, and Parthasarathy's Query by Output (QBO) techniques [3], solve a different problem from AQG. QBO targets skilled workers for companies with hundreds or thousands of tables, while we target very unskilled workers working with small relational schemas. QBO can be used to provide further insight into complicated queries, and to generate similar queries to one that is provided (e.g. for security reasons [3]). QBO is not intended, nor suitable for use by unskilled workers. The algorithm from as well as all examples and potential uses listed in [3] require that a worker either already know the desired query Q, or its $Q(D)$ on the target database D. This is the fundamental difference between QBO and AQG – AQG is intended for situations in which a user does not know how to produce Q, but wishes to obtain $Q(D)$. Thus, QBO assumes the user already has the answer to the question AQG answers.

This difference is exemplified by the differences in the input provided to QBO versus AQG. A QBO system can be provided with either Q or $Q(D)$, from which it provides a set of identical or similar queries $Q_1, Q_2, ..., Q_n$. The AQG engine is provided with $desired \subseteq Q(D)$ and $undesirable \subseteq \neg Q(D)$, where usually $|desired| << |Q(D)|$, where $| \cdot |$ denotes the number of tuples in a relation. The AQG engine then produces $\hat{Q}$, an approximation of the unknown Q.

Though different, QBO does provide some interesting results that carry over to AQG, primarily that the task of finding a query that exactly matches the $desired$ and $undesirable$ tables is **NP**-complete.

4 Methods

All trials were performed on a Sony VAIO VGN-FW running Microsoft Windows 7 with 4GB RAM, and an Intel Centrino 2 processor clocked at 2.534 GHz. Our software was implemented in Python using the SQLite3 database engine. Indices were constructed over all columns of all relations, resulting in execution time improvements of up to an order on some queries. Notice that AQG does not require that such indices exist. However, since the goal of this paper is only to test whether or not we can effectively search the space of queries, without necessarily worrying about how optimized the search engine is, the use of many indices is reasonable.

4.1 Heuristic Function

We use a genetic algorithm [1] to search the space of possible Select-From-Where (SFW) SQL queries for the simplest one that best satisfies the constraints. In order to use a genetic algorithm, we define a heuristic evaluation function $h(R)$, which maps a relation R returned by a query Q to the real numbers. This heuristic will be used to evaluate the fitness of the relation returned by an SQL query. The heuristic must reward returning tuples from the $desired$ table and not returning tuples from the $undesirable$ tables. Additionally, it should punish queries for failing to return tuples in the $desired$ list and for returning tuples from the $undesirable$ list.

Illegal SQL queries (such as ones that reference tables in the WHERE clause that are not included in the FROM clause) must always receive a fitness of negative infinity. Additionally, the heuristic must be able to handle tables with the incorrect number of columns. To handle this, if the solution has too few columns, it is given a fitness of negative infinity. If it has too many columns, and the $desired$ table has n columns, then only the first n columns of R are considered.

In order to simplify the problem and circumvent the problem that data types of columns may not align, we forced all columns to be integers. For strings, we kept a separate look-up table relating the integers to the corresponding strings (names, gender, and job titles). Thus, internally the AQG engine need only handle integers.

Finally, the heuristic should also include a term that punishes complexity. In order to implement this, we subtract a constant W from the fitness for every clause in the query. That is, we subtract W for every term in the SELECT statement, every term in the FROM statement, and every AND, OR, or NOT in the WHERE statement. Thus, W is a tunable parameter that sets the tradeoff between the accuracy of the query and its complexity.

The resulting heuristic, h, can be explicitly defined as a function over the returned relation, R, from an SFW SQL query, Q:

$$h(R,Q) = I_{legal}(R) + I_{columns}(R) + 2 * \frac{p(R)r(R)}{p(R)+r(R)} + W * c(Q) \qquad (1)$$

where I_{legal} is an indicator function taking value 0 if R is legal, and negative infinity otherwise, $I_{columns}(R)$ is an indicator function taking value 0 if R has n or more columns, W is a tunable parameter setting the tradeoff between accuracy and complexity, $c(Q)$ is the complexity of the query Q, defined as the sum of the number of clauses in the SELECT, FROM, and WHERE statements, $r(R)$ denotes the recall:

$$r(R) = \frac{TP}{TP+FP}, \qquad (2)$$

and $p(r)$ denotes the precision:

$$p(r) = \frac{TP}{TP+FN}, \qquad (3)$$

where TP is the number of true-positives: tuples in R that are also in $desired$, TN is the number of true negatives: tuples not in R that are in $undesired$, FP is the number of false-positives: tuples in R that are in $undesired$, and FN is th number of false-negatives: the number of tuples not in R that are in $desired$. The middle term of the heuristic defined in Equation 1 is called the F-measure, and represents the harmonic mean of precision and recall. With $W = 0$, the maximum possible value produced by this heuristic is always one, and the minimum zero.

In order to avoid misleading fitness results, the tuples R must be filtered to only include those in the union of $desired$ and $undesired$ prior to the computation of TP, TN, FP, and FN. Initial implementations without this filter failed to converge to reasonable queries for relatively simple tasks. Without the filter, the definitions of these statistics break down because there are tuples that are in R but neither $desired$ nor $undesired$.

4.2 Genetic Algorithm

In order to simplify the genetic algorithm, the entire database is first modified so that all columns have type integer, the values of which can be mapped back to the original strings or other datatypes. This modification is transparent to the user, though it allows the AQG to ignore all type-based constraints.

Aggregation operators, query unions, nested queries, and all complex operators are left for future work, as this paper is a proof of concept showing the limitations of a simple AQG implementation. The only operators allowed in the WHERE clause are AND, OR, $>$, $<$, $\leq$, $\geq$, $=$, and $<>$. Notice that the NOT operator is not needed because it can be achieved by taking the oposite operator (e.g. $>$ becomes $\leq$) within the Boolean clause.

The genetic algorithm implemented uses a population of $p = 1000$, and allows for both mutations and crossovers. Children for the subsequent generation are selected by directly copying over the queries with fitness in the top $k_1 = 5\%$ (using the fitness function defined by h from Equation 1). The next $k_2 = 20\%$ of the new population are generated by mutating members of the top $k_1 = 5\%$, and the remaining members (75%) are generated randomly. During a mutation, each of the possible mutations is performed with a fixed probability and several different types of mutations can be applied over the same query at the same time.

The mutation function takes on SQL query and generates a mutated version by randomly selecting between the following list of possible mutations.

Flip Operator: A random operator ($>$, $<$, $\leq$, $\geq$, $=$, or $<>$) in the WHERE clause is flipped to a different operator.

Swap Operands: The left hand side and right hand side of a boolean statement in the WHERE clause are flipped. For example, $(a \leq b)$ would become $(b \leq a)$.

Flip Operand: The value of the left hand side or right hand side of a boolean statement in the WHERE clause is changed to a random new value. This new value could be numerical, or any column from a table in the FROM clause. The columns are chosen uniformly over the available tables. Numerical values are chosen depending on tunable parameters. Numerical values are capped to be between 0 and 250, with a higher weight on the values 0 and 1. These constraints are domain specific, though they could be precomputed as a function of all tuples in the database.

Increment Operand: A numerical operand is incremented or decremented by either 1 or 10.

New Constraint: A new constraint is randomly generated and added to the WHERE clause. It is appeneded with either AND or OR, chosen randomly. The condition can be a test between the value of two colums, or a test between a column and a numerical value.

Flip Column: One of the columns in the SELECT clause is replaced with a different column.

Delete Column: One column from the SELECT clause is deleted from the SELECT clause.

Add Column: A column is added to the SELECT clause, chosen uniformly from the tables in the FROM clause.

Swap Columns: Swaps the order of two columns in the SELECT clause.

Flip Table: Replaces one of the relations in the FROM clause with a different relation and remove all references to it in the SELECT and WHERE clauses. Duplicate tables are allowed, with different unique identifiers, so that self-joins are possible.

Add Table: Adds a new table, with a unique identifier, to the FROM clause.

Delete Table: Deletes a random table from the FROM clause and removes all references to it in the SELECT and WHERE clauses.

Delete Constraint: Deletes a constraints from the WHERE clause.

Flip Connector: Flips an AND to an OR or vice versa in the WHERE clause.

The crossover operator takes two SQL queries and combines them to create a new query. Given two queries q_1:

SELECT S
FROM F
WHERE W

and q_2:

SELECT S'
FROM F'
WHERE W'

we generate a new query q_3:

SELECT S, S'
FROM F, F'
WHERE W AND W'

For queries with n desired columns where q_1 already returns n or more columns, the fitness of q_3 will be identical to that of q_1 because the latter columns will be ignored by the heuristic. However, future mutations of q_3 can now relate the WHERE clauses of q_1 and q_2 in novel ways. Preliminary tests showed that including crossovers did not improve performance, so the following results do not include crossovers. The reason, we believe, is that crossovers usually create new queries with very high complexity. Since the fitness of a query depends on both its correctness and its complexity, the GA quickly notices that it can improve upon its fitness by decreasing its complexity; i.e., by removing terms from its clauses. After such deletions are performed, however, most of the resulting queries tend to look much like the ones that were originally used on the crossover. In other words, crossovers tend to generate queries that are not qualitatively better than the ones that already existed, and when it does, they tend to be so complex that they are quickly removed from the population.

In order to improve performance, we allow the a programmer (skilled user) to determine general biases that are expected to improve performance. Such biases include the various probabilities of different mutations and operators. Another example of a bias is in the definition of the Flip Operand operator described above – the higher probability of constants 1 and 0 is a bias that we inserted into the AQG.

In order to further improve the performance of the genetic algorithm, we also required that all joins over tables be done via foreign keys. In our relational schema, this means that we can only compare PIDs, JIDs, MIDs, and SIDs to each other. All other comparisons must be between two of the same column (e.g. Salary and Salary) or a column and a numeric value (e.g. Salary at 10). This avoids exceedingly uncommon comparisons, such as between Salary and Age. Though these comparisons may be included in obscure queries, the benefit of including them is outweighted by their increase in the size of the search space for all other queries.

4.3 Queries

To test the genetic algorithm's performance as an AQG engine, we created an instance of the relational schema presented in Section 1, containing 36 tuples in the **Job** relation, 750 tuples in the **Manager** relation, and 1,000 tuples in the **Person** relation. Units for all values, other than keys, were selected to be within the range 0 to 250, though some columns, such as Job.Hours, do not cover this entire domain.

We then developed a list of queries to test the AQG. These queries are separated into three classes: *easy* queries only require one table and one clause in the WHERE statement, *medium* queries require one table and two clauses in the WHERE statement, and *hard* queries require the join of two tables and two statements in the WHERE clause. Notice that queries often ask for PIDs rather than names. This is because we have a small set of names and a large set of people, so there are likely two people with the same name who are older than and younger than 30, respectively. This quirk is a result of the synthetic database created for this problem, as an actual database would have greater diversity in names.

The $desired$ (resp. $undesirable$) table was a randomly selected subset of the tuples returned by the target query (resp. not returned by the target query). The sizes of $desired$ and $undesirable$ are specified for each query.

Easy:
1. Select the PID of all people older than 30. Desired SQL query:
SELECT $p.PID$
FROM Person p
WHERE $p.Age > 30$
$|Desired| = 10$ and $|Undesirable| = 10$

2. Select the titles of all jobs with salaries greater than 80. Desired SQL query:
SELECT $j.Title$
FROM Job j
WHERE $j.Salary > 80$
$|Desired| = 10$ and $|Undesirable| = 10$

Medium:
3. Select the PID of all people who are male and older than 30 Desired SQL query:
SELECT $p.PID$
FROM Person p
WHERE $p.gender = \text{'}male'$ AND $p.Age > 30$
$|Desired| = 20$ and $|Undesirable| = 20$

Hard:
4. Select the PIDs of all people who work more than 80 hours. Desired SQL query:
SELECT $p.PID$
FROM Person p, Job j
WHERE $(j.Hours > 80)$ AND $(p.JID = j.JID)$
$|Desired| = 20$ and $|Undesirable| = 20$

5 Results

In this section, we present the results of running the AQG on the target queries defined in the previous section. We will report various results for each query, in order.

Query 1: *Select the PID of all people older than 30.* (Easy)
First, we ran the AQG on sets of 10 $desired$ and 10 $undesirable$ tuples, with $W = 0$, and with the initial population populated by random queries containing one clause in each of the SELECT, FROM, and WHERE statements. Below we present the results of 10 executions, each lasting 20 generations.

```
SELECT Person.PID
FROM Person
WHERE Person.Age > 30

SELECT Person.PID
FROM Person
WHERE Person.Age > 34

SELECT Person.PID
FROM Person
WHERE Person.Age > 31

SELECT Person.PID
FROM Person
WHERE Person.Age > 34

SELECT Person.PID
FROM Person
WHERE 44 <= Person.Age

SELECT Person.PID
FROM Person
WHERE Person.Age >= 48

SELECT Person.PID
FROM Person
WHERE Person.Age >= 45

SELECT Person.PID
FROM Person
WHERE Person.Age > 30

SELECT Person.PID
FROM Person
WHERE Person.Age >= 30

SELECT Person.PID
FROM Person
```

WHERE Person.Age $>=$ 40

In all cases, the final fitness was 1.0, which was typically reached within 10 iterations. Throughout all queries, we observed the trend that variance in constants increases as the size of $desired$ and $undesirable$ decreases. This occurs because a smaller sample of points has a smaller probability of strictly enforcing the desired constraint. For example, if we wish to find people older than 30, but only provide positive examples of people older than 40, the AQG has no way of knowing whether the target query is people older than 30 or people older than 40. Because our selection of $desired$ and $undesirable$ tuples is random, the probability of tight bounds on constants increases as the number of $desired$ and $undesirable$ tuples increases. However, if the tuples are provided by a user, a tight bound can be implemented using fewer tuples.

When this test was repeated with the random initial queries having up to three clauses in the WHERE statement and two columns in the FROM statement, but still $W = 0$, the AQG finds a solution that is more complicated than necessary, but produces the desired result, such as the following query generated by the AQG (fitness = 0.947368421053):

SELECT Person.PID
FROM Person, Job
WHERE Job.JID = Person.JID AND 37 $<$ Person.Age

If W is set to a value larger than zero, the AQG will again find the minimal query with one column in the FROM statement and one clause in the WHERE statement. Additionally, when using 5 $desired$ and 5 $undesirable$ tuples, the AQG finds the desired query with higher variance in the age constant, though it always achieves perfect fitness.

Query 2: *Select the JID of all jobs with salary greater than 80.* (Easy)
As with Query 1, we present the result of 10 runs with $W = 0$, and with random initial queries containing one clause in the SELECT, FROM, and WHERE statements. Again, we run for 20 generations. All results achieved a fitness of 1.0, and most converged within 10 generations. Performance with different settings of W resulted in similar results to those described for Query 1.

SELECT Job.JID
FROM Job
WHERE 89 $<$ Job.Salary

SELECT Job.JID
FROM Job
WHERE Job.Salary $>=$ 75

SELECT Job.Title
FROM Job
WHERE 79 $<$ Job.Salary

SELECT Job.Title

FROM Job
WHERE Job.Salary $>$ 77

SELECT Job.Title
FROM Job
WHERE 98 $<=$ Job.Salary

SELECT Job.JID
FROM Job
WHERE Job.Salary $>$ 76

SELECT Job.JID
FROM Job
WHERE Job.Salary $>=$ 73

SELECT Job.JID
FROM Job
WHERE Job.Salary $>=$ 79

SELECT Job.JID
FROM Job
WHERE 68 $<$ Job.Salary

SELECT Job.Title
FROM Job
WHERE Job.Salary $>=$ 89

Query 3: *Select the PID of males older than 30.* (Medium).
Below we present the results of 10 consecutive AQG executions, with $W = 0$, 50 generations, and queries starting with 1 SELECT clause, 1 FROM clause, and 2 WHERE clauses. Executions starting with up to 2 FROM clauses and 3 WHERE clauses result in the correct, though more complex, solution being found after more generations. When using 3 WHERE clauses and two FROM clauses with $W > 0$, the AQG fails to converge, though if the initial population is set to the results from a run with $W = 0$, a subsequent run with $W > 0$ results in the least complex query, as desired. We suspect that the reason for the divergence is that fixing $W > 0$ tries to enforce queries with minimal complexity even during the initial phase of the algorithm when the space of solutions should be explored more aggressively. Thus, by using $W > 0$ since the beginning of the algorithm, several queries that could form the basis for correct solutions might not be found.

Because the fitness is not always 1, the fitness of each query generated is also provided below. Recall that all values were changed to integers, so $male = 0$ and $female = 1$.

Fitness: 1.0
SELECT Person.PID
FROM Person
WHERE Person.Age $>=$ 31 AND 0 $>=$ Person.Gender

```
Fitness: 1.0
SELECT Person.PID
FROM Person
WHERE Person.Gender <= 0 AND Person.Age > 30

Fitness: 0.975609756098
SELECT Person.PID
FROM Person
WHERE 156 >= Person.Name AND 34 <= Person.Age

Fitness: 1.0
SELECT Person.PID
FROM Person
WHERE 1 <> Person.Gender AND Person.Gender < 45

Fitness: 0.975609756098
SELECT Person.PID
FROM Person
WHERE Person.Age >= 25 AND 1 <> Person.Gender

Fitness: 1.0
SELECT Person.PID
FROM Person
WHERE 0 >= Person.Gender AND Person.Age >= 36

Fitness: 1.0
SELECT Person.PID
FROM Person
WHERE 0 = Person.Gender AND Person.Age >= 31

Fitness: 0.975609756098
SELECT Person.PID
FROM Person
WHERE Person.Name < 118 AND 26 <= Person.Age

Fitness: 0.952380952381
SELECT Person.PID
FROM Person
WHERE Person.Name < 130 AND Person.Name <> 51

Fitness: 0.93023255814
SELECT Person.PID
FROM Person
WHERE Person.Name < 114 AND Person.Age > 11
```

When run for more than 50 generations, the queries tend to converge to those with perfect fitness (1.0). We report the results after only 50 generations to showcase the AQG's gradient ascending properties. That is, the target query is not found randomly. Rather, queries such as those reported above slowly evolve until they represent the target query.

The weakness of the constraints in $desired$ and $undesirable$ when using random subsets of the target tuples is evident in the first query returned above, in which it returns all people older than 31, regardless of gender, but still achieves a fitness of 1.0 (perfect classification of the $desired$ and $undesirable$ tuples). Some of the queries presented above show how unexpected solutions can also often be returned. In this case, the AQG took advantage of the unintentional structure of the mapping from names to integers. Specifically, all male names mapped to the integers from 1 to 123, while 124 to 221 are female names. Thus, the constraint that the person's name is less than than 114 is an alternate representation of the constraint that we desired PIDs of people who are male.

The second to last result is particularly interesting, in that the query returned has very high fitness, but doesn't initially seem related to the target query. However, the same name-mapping argument means that the first clause actually constrains the tuples returned to be all males, with a few exceptions (those with certain names are not returned). This query can have high fitness in cases where the $desired$ set contains mostly males, and the $undesirable$ set contains mostly females and males with the names that are filtered out. Such a result is indicative of a poor selection of the $desired$ and $undesirable$ sets, as they do not well represent the age constraint.

Query 4: *Select the PIDs of all people who work more than 80 hours.* (Hard)
Again, we present the results of 10 consecutive executions of the AQG, with $W = 0$, starting with one SELECT clause, two FROM clauses, and two WHERE constraints. Results beginning with different numbers of clauses require additional time to converge. We present the results after 50 generations, though the join of the Person and Job table is usually discovered within 5 generations. All trials resulted in a query with fitness 1.0.

```
SELECT Person.PID
FROM Job, Person
WHERE Job.JID = Person.JID AND 68 < Job.Hours

SELECT Person.PID
FROM Job, Person
WHERE Job.Hours > 80 AND Person.JID = Job.JID

SELECT Person.PID
FROM Job, Person
WHERE Person.JID = Job.JID AND Job.Hours >= 71

SELECT Person.PID
FROM Job, Person
WHERE Job.Hours >= 82 AND Person.JID = Job.JID

SELECT Person.PID
FROM Job, Person
WHERE Job.Hours > 79 AND Job.JID = Person.JID

SELECT Person.PID
FROM Person, Job
```

```
WHERE Job.Hours > 79 AND Person.JID = Job.JID

SELECT Person.PID
FROM Job, Person
WHERE Job.JID = Person.JID AND Job.Hours >= 81

SELECT Person.PID
FROM Job, Person
WHERE Job.Hours > 67 AND Person.JID = Job.JID

SELECT Person.PID
FROM Person, Job
WHERE 78 < Job.Hours AND Job.JID = Person.JID

SELECT Person.PID
FROM Person, Job
WHERE 72 < Job.Hours AND Job.JID = Person.JID
```

The queries returned all contain the join of the **Person** and **Job** tables, and tend to represent the constraint on hours-worked at least as well as the constraints in the first and second (easy) queries.

6 Discussion

The results are encouraging, suggesting that AQG is feasible even for queries that require joins. We observed that the AQG often achieved perfect fitness, yet failed to exactly predict the desired query. This occurs because there are multiple queries that achieve perfect fitness for the given subset of $desired$ and $undesirable$ tuples. We stated that we desire the simplest such query (i.e. with the fewest clauses), however there are often many queries with perfect fitness and minimal clauses. An example of this was the first query, in which the $desired$ and $undesirable$ tuples often failed to specify exactly where the age boundary should be, with many queries returning only people older than 35, rather than the ideal query of all people older than 30. Thus, if a user provides $desired$ and $undesirable$ tuples, he or she should provide tuples that tightly constrain the desired results. So, if one desired people older than 30, positive examples of people who are 31, and negative examples of people who are 30 should be provided.

All queries presented in the previous section resulted from trials with $W = 0$. We noted that queries requiring more than one clause in the SELECT, FROM, and WHERE statements are not found when $W > 0$. This is because the genetic algorithm can rapidly improve fitness by removing clauses, until the entire population that carries from generation to generation contains queries that are not complex enough to represent the target query. Thus, we propose that the AQG first be executed with $W = 0$, and the resulting query used to seed the population of a second run with $W > 1$. Trials of this method on the third (medium) query in the previous section resulted in the target query being found without extraneous clauses in the WHERE statement that are always true or always false.

Though these results are encouraging, we did observe that AQG encounters serious limitations when trying to solve more complicated queries. Specifically, consider a query that requires three tables in the FROM clause, such as selecting all people who are older than their managers. When the AQG is executed on such a query, it quickly generates random queries that take the cross product of the **person** table (or any other large table) with itself three times. This results in $1,000,000,000$ tuples being returned, which results in exorbitant execution times. Future implementations should include techniques for terminating queries that take too long to execute. Unfortunately, SQLite3 does not allow for this. Thus, fitness computations of such queries kill the performance of the AQG, causing it to hang for extended periods of time. An example of such a query is provided below:

```
SELECT p1.PID
FROM Person p1, Person p2, Person p3
WHERE p1.PID >= 0
```

7 Conclusion

We have shown that AQG is capable of generating desired queries when provided with only a small number of $desired$ and $undesirable$ tuples. Thus, an unskilled user can formulate SQL queries without any knowledge of relational algebra, nor the structure of data. In our review of the literature we have found that, though some methods are superficially similar, this work is the first of its kind.

We conclude that AQG should be the focus of additional research. Specifically, methods should be created to allow for the generation of queries including multiple joins. Part of achieving this will likely involve the AQG allowing for the termination of queries that run for extended periods of time and the inclusion of query execution time in the heuristic.

References

[1] S. Russell and P. Norwig. *Artificial Intelligence: A Modern Approach.* Pearson Education, second edition, 2003.

[2] S. Santini and R. Jain. Beyond query by example. In *Proceedings of the sixth ACM international conference on multimedia*, pages 345–250, 1998.

[3] Q. Tran, C. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the 35th SIGMOD international conference on Management of data*, 2009.

[4] M. Zloof. Query by example. In *AFIPS National Computer Conference*, pages 431–438, 1975.

[5] M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *Proceedings of the 1st International Conference on Very Large Data Bases*, pages 1–24, 1975.

8 Addendum: Improved Results

In this section, we present results that were generated after the creation of this initial report. Rather than rework the contents of the writeup to include these improvements, we provide them as an addendum.

The conclusion stated that future work should focus on methods for extending AQG to larger queries, which would likely require the termination of queries that run for extended periods of time. In this addendum, we did just this, and managed to generate queries including up to four clauses in the FROM statement, and five clauses in the WHERE statement. We suspect that our current method would take excessive amounts of time for more clauses in the FROM statement, though additional WHERE clauses would not significantly affect performance.

We put a catch in place to terminate queries that run for more than a second, and give them fitness of negative infinity. We then ran the AQG on the query: Select the PIDs of everyone who manages someone younger than 20. An example query to compute this is:

```
SELECT p1.PID
FROM Person p1, Person p2, Manager m
WHERE p1.PID = m.MID
AND p2.PID = m.SID
AND p2.Age < 20
```

Notice that this query has three clauses in the FROM statement and three clauses in the WHERE statement. The following queries were all generated using the same setup as the previous qeuries from this report, with 10 $desired$ and 10 $undesirable$ tuples. In cases where the fitness reported below is not between 0 and 1, a different fitness metric was used:

$$f(R, Q) = TP + TN - FN - FP + W * c(Q) \tag{4}$$

Each of the following queries was produced after a different number of generations, so the number of generations is also reported. Each query below is from a different run - they are not consecutive queries from a single run.

Generation 44
Fitness: 60 (optimal fitness: 60)
Query:

```
SELECT m.MID
FROM Person p, Job j, Manager m
WHERE p.PID = m.SID
AND p.Age <= 22
AND p.JID = j.JID
```

Generation 35
Fitness: 60 (optimal fitness: 60)
Query:

```
SELECT m.MID
```

```
FROM Person p, Job j, Manager m
WHERE p.PID = m.SID
AND p.Age <= 32
AND p.JID = j.JID

Generation 51
Fitness: 60 (optimal fitness: 60 )
Query:
SELECT m.MID
FROM Person p, Manager m, Job j
WHERE p.PID = m.SID
AND 26 >= p.Age
AND p.JID = j.JID

Fitness: 0.952380952381
Query:
SELECT m.MID
FROM Person p, Job j, Manager m
WHERE j.JID = p.JID
AND j.Salary > 10
AND m.SID = p.PID

Fitness: 0.952380952381
Query:
SELECT m.MID
FROM Person p, Job j, Manager m
WHERE 10 < j.Salary
AND p.JID = j.JID
AND p.PID = m.SID

Fitness: 0.967741935484
Query:
SELECT m.MID
FROM Person p, Job j, Manager m
WHERE 24 > p.Age
AND m.SID = p.PID
AND p.JID = j.JID
```

These are all very succesfull runs, accurately predicting the target query. These queries either generate the target query directly, or they use the trick that employees who have a Salary $>$ 10 is a very accurate prediction for employees with Age $<$ 20. Over the entire database, this is true approximately 95% of the time. Thus, the constraint 10 $<$ Salary is effectively equivalent to Age $<$ 20.

Next, we ran an even more difficult query, requiring four clauses in the FROM statement, and five in the WHERE statement. This query select people who work more than 10 hours a week and manage someone younger than 20:

```
SELECT p1.PID
FROM Person p1, Person p2, Job j, Manager m
WHERE p1.PID = m.MID
```

```
AND p2.PID = m.SID
AND p2.JID = j.JID
AND p1.Hours > 10
AND p2.Age < 20
```

Below we list three queries generated by the AQG when run on this desired query with 10 $desirable$ and 10 $undesirable$ tuples.

```
Fitness: 0.739
SELECT m.MID
FROM Job j1, Manager m, Job j2, Person p1,
WHERE j2.Title <= 0
AND j1.JID = j2.JID
AND 27 <= p1.Age
AND j1.Title <> 186
AND p1.PID = m.SID
```

```
Fitness: 0.956
SELECT m.MID
FROM Job j1, Person p1, Manager m, person p2
WHERE 153 <> j1.Title
AND j1.JID = p1.JID
AND m.MID = p1.PID
AND p2.PID = m.SID
AND 23 <= p2.Age
```

```
Fitness: 1.0
SELECT p2.PID
FROM Manager m, Job j1, Person p1, Person p2
WHERE j1.JID = p2.JID
AND 152 <> j1.Title
AND m.MID = p2.PID
AND 31 < p1.Age
AND p1.PID = m.SID
```

Again, these are excellent results. The first one implements just part of the requirement (selecting managers who manage someone younger than 20), and therefore receives only mediocre fitness. The second two get the training set either perfectly correct or almost perfectly correct. They use the trick that managers who work more than ten hours tend to be those whose job title is not 153. This works because the set of jobs with title other than 153 happens to equal the set of jobs that require more than 10 hours of work per week, with only a four tuple difference. Thus, the two constraints are interchangeable for practical purposes, because the random selection of a JID, filtering by title $<>$ 156, will be exactly the same as filtering by hours $>$ 10 in $36/40 = 90\%$ of the time.

In these additional results, we have shown that the AQG system is capable of generating fairly complicated queries when provided with only a small sample of $desirable$ and $undesirable$ tuples. We believe that the performance would be greatly enhanced by additional runtime and computational power. AQG is certainly worthy of future research to determine its limits.