Final Report for AI based SE

Seah Kim – 20184228 Jeonggwan Lee – 20184400 GHI – 2018xxxx
Nick Heppert – 20186505

December 17, 2018

Chapter 1

Introduction

The amount of information is exponentially increasing, a robust and delicate index structure also be needed. SQL, which is a language to manage Database Management System(DBMS) is widely used as tool which arranges effectively indices and relationship between tables, the area of application is very large, such as Electronic Health Record(EHR), customer management, and distribution management. Software developers can get necessary information through query statement in SQL. However, they need to verify their own query statement and check whether unexpected bugs exist. Otherwise, we can't believe the results of query and beyond lose reliability of SQL. After verifying, we can judge whether this query statement is semantic or not. If the query statement is simple, developers can make test cases manually considering coverage targets. However, if query statement is complex, this manual making is very inefficient and hard to deal with.

To handle this problem, EvoSQL made test case generation tool covering whole coverage targets for SQL queries using genetic algorithm(GA), it shows astonishingly covering performance, 98.6%. Their contributions are, first suggesting the definition of test case generation problem for SQL and a "physical query plan", which is a series of step to solve in query. Through the physical query plan, they defined a fitness function and implement crossover and mutation as general GA. They presented seeding strategies, which the model has seeding pool to generate new individual, to fit the characteristic of SQL which normally covers string and integer(days). They compared three methods (Random search, Biased Random Search (supported by Seeding strategy), GA), and show how GA can cover coverage targets of various queries.

GA is much better than other two methods. But we thought any other improvements than GA, and wondered why they didn't use Multi-Objective Optimization(MOO) methods. What we assumed the weakness of GA is that they didn't arrange the order of coverage targets to solve so that we should wait for solving easy coverage target if it is behind difficult coverage targets, and coverage targets don?t share their semantic discovery to others because of limitation of single target strategy.

To apply MOO, we expected that our model can be guided by similar solution from other coverage targets. Therefore, we first applied NSGA-II as basic MOO method, changed crowding distance into "sort fronts by covered target" and added combine operator as minor technique to satisfy the coverage target easily.

Our contribution is an implementation of MOO approach of test case generation for SQL, analysis and comparison of results and the answer to suitability of MOO for test case generation for SQL. we successfully implemented the MOO approach, but our conclusion is it is not applicable to test case generation for SQL, because of too much time cost compared to the performance.

We organized contents as follows. Section 2 describes genetic algorithm in EvoSQL as baseline, and the representation of GA setting for SQL test case generation. Section 3 presents our representation of MOO setting, our modified model based on NSGA-II. Section 4 we evaluate our model and analyze failure of it. We conclude the paper in section.

Chapter 2

Problem Formulation

Chapter 3

Methodology

It is noticeable from the result of EvoSQL that GA approach outperforms biased search in solving complex queries, whereas biased search excels in simple ones. Although the GA implementation has an initialization step that is similar to what happens in the biased search, the GA spends time calculating the fitnesses and applying the search operators at every iteration of evolution. All these steps do not happen in the biased search. Therefore, we expected that reformulating the single GA approach into using multi-objective strategies could reduce the inefficiency of the original search strategy. By using a multi-objective approach, a population from each iteration would be able to share its semantic discovery and save the usage of unnecessary time budget.

3.1 Extending Fitness Calculation

Each coverage target from a single query is set as an objective to be optimized. For every coverage target, fitnesses of a solution are being calculated at the same time. We newly defined `class FixtureMOO`, which has a member variable of the list containing the fitnesses of targets(`fitness_moo`). We utilized the same fitness function with EvoSQL for effective comparison to single objective approach. Once a `FixtureMOO` object is created, it calculates the fitness of the query on a test.

```
public class FixtureMOO extends Fixture{
[...]

private List<FixutreFitness> fitness_moo = new ArrayList<FixtureFitness>();
[...]

public int calculate_fitness_moo(List <String> paths_to_test,
Map <String, TableSchema> tableSchemas){
[...]

}
}
```

Listing 3.1: FixtureMOO.java

3.2 Implementing NSGA-II

With multiple objectives in a problem indicates that there is a set of optimal solutions instead of a single optimal solution. Mostly, these solutions are Pareto-optimal and hard to say that one solution to be better

Coverage Targets	1-2	3-4	5-6	7-8	9-10	11-15	16-20	21+	Total
	656	382	408	346	114	107	51	71	2135

Table 3.1: Number of Queries by coverage targets

than the other. Among several classical multi-objective evolutionary algorithms, we choose Non-dominated Sorting Genetic Algorithm II (NSGA-II) for our implementation since it is simple and straightforward.

3.2.1 Non-Dominating Sort with Preference Criterion

70 percent of queries that EvoSQL handles have more than three coverage targets.(Table.1) It is shown that NSGA-II has been successful in solving the optimization problem with multiple objectives. However, it is not effective for solving problems with *many* objectives.In case of many objectives, most of the individuals are non-dominated by each other and it makes hard setting up a Pareto front.

To overcome this limitation, we applied preference sorting which Panichella *et al.* proposed., Panichella *et al.* suggested a way of considering both the non-dominance relation and the preference criterion. It imposes an order of preference among non-dominated solutions. In short, preference sorting adds individuals which achieved the lowest in each objective to the first non-dominating front. In our case, a set of tables which covers the maximum number of target among other solutions will be located on the first pareto front.

3.2.2 Sorting by covered targets

As a next step, nondominated solutions are sorted by the number of covered targets. Originally in NSGA-II, comparing the crowdedness among nondominated solutions was introduced. This procedure ensures diversity of solutions. However, the current implementation of EvoSQL does not provide an explicit calculation of the fitness as the paper said. Only the comparison of fitnesses between two individual is possible. Therefore, it is not feasible to get a numeric value for setting up the boundary points. This problem is still left as an open Github issue.[1]

[1] https://github.com/SERG-Delft/evosql/issues/41

Chapter 4

Evaluation

After successfully implementing our MOOP extension we started to conduct various experiments. In the following we will give an overview of our experimental setup, followed by the results and analyzing them.

4.1 Setup

We used the previous serialized queries (see **??**) from the three different projects that were provided with the Github repository[2] by the authors of the original EvoSQL paper. As previously explained in **??** for each query we extract multiple coverage targets. In fig. 4.1 we report the distribution of all coverage targets from all three projects.

As seen in the data distribution a lot of our queries have three coverage targets which need to be covered in order to say the query is solved. The main reason why so many queries have three targets is because if we have single condition in our query, for instance:

```
SELECT user_id AS 'userId' FROM autofollow WHERE entity_type = 'Account'
```

The three possible targets are:

```
SELECT "user_id" AS "userId" FROM "autofollow" WHERE NOT ("entity_type" = 'Account')
SELECT "user_id" AS "userId" FROM "autofollow" WHERE ("entity_type" = 'Account')
SELECT "user_id" AS "userId" FROM "autofollow" WHERE ("entity_type" IS NULL)
```

In our experiment we measured the average time it takes to solve a query with a given amount of targets. To ensure we don't run into unsolvable targets we limit the maximum execution time to 30 minutes.

In the case of the original implementation we sum up the amount of time needed to solve each target individually. This sum then forms the total execution time for the single-objective optimization. When measuring the multi-objective optimization we only need to measure the over all execution time.

4.2 Results

To our personal surprise the time to fully cover a query increased for our multi-objective optimization implementation with increasing amount of targets to cover, even hitting time outs when having a lot of coverage targets (objectives).

[2] https://github.com/SERG-Delft/evosql

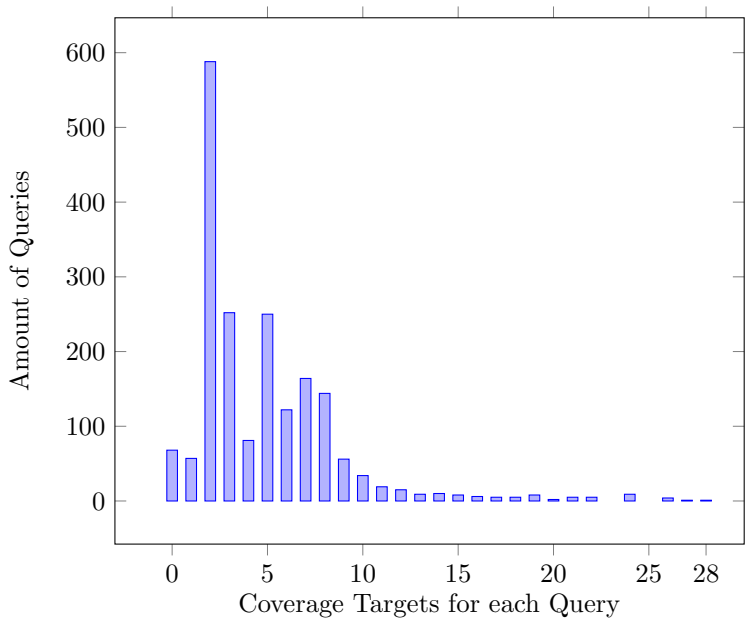

Figure 4.1: Distribution of coverage targets (i.e. objectives) in our whole test query dataset. The were also queries with more than 28 coverage targets but they were truncated as they could be considered as not important.

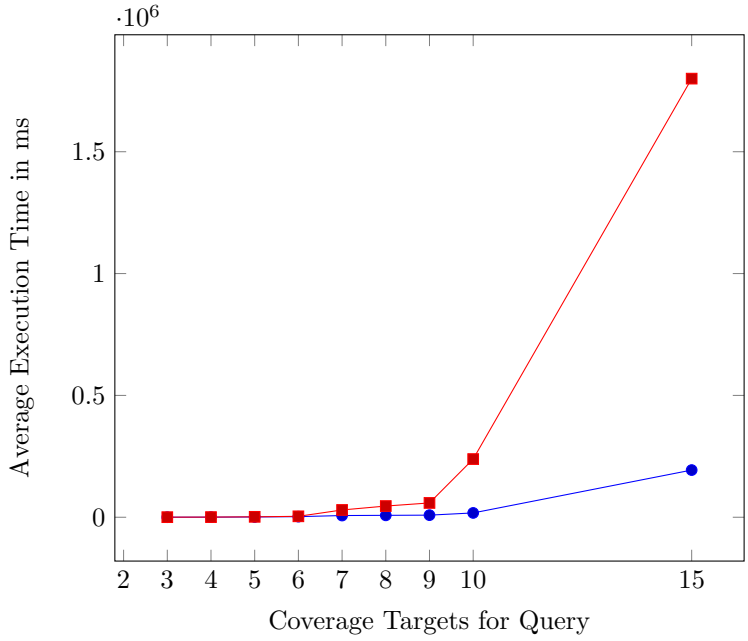

Figure 4.2: Average execution time for each covered target. For the last data point (15 coverage targets, our MOO was able to only cover 3 targets in the given time frame of 30 mins.)

Table 4.1: Ratio of execution time between original single-objective optimization and our implemented multi-objective optimization (if > 1: multi-objective optimization is slower)

Coverage Targets	3	4	5	6	7	8	9	10
Ratio (MOO/GA)	2.259	1.587	1.501	1.581	4.265	5.860	6.816	13.463

4.3 Failure Analysis

If we look at table 4.1 we can see that the actual ratio between the two different execution time is not staying constant. This lead to the conclusion on our side that with increasing amount of coverage targets our multi-objective algorithm performs worse and worse.

To further investigate why this happens we took a look at the execution time of each component. Here we could narrow it down that our fitness calculation becomes worse with increasing amount of coverage targets. Initially, this makes sense as we have to calculate the fitness `amount of coverage targets`-times for each individual. But as we have execute the optimization algorithm only once in the case of multi-objective optimization and not "amount of coverage targets" times it should even out in the end.
Next, we found out that the actual time for calculating the fitness for one coverage target increases as well with having more coverage targets. When looking at this aspect, the only thing that changes with increasing amount of coverage targets is the maximum amount of rows we can have in an individual.
We have to scale the maximum amount of rows based on the amount of coverage targets as our individuals need to have enough "capacity" to potentially cover all targets. We do so by setting the capacity to

```
max_rows = max_rows_per_target * amount_coverage_targets
```

We adapted the given `max_rows_per_target` from the single objective optimization and set it to 4.

4.3.1 Why Does the Capacity Matters?

We will now analyze why the fitness calculation becomes more costly when the amount of rows increase per table.

Theoretical Analysis

Assuming we have a simple example with two tables, if we now execute a query on these two tables with a form like `WHERE table1.field == "Foo" AND table2.field == "Bar"` we have to calculate the minimum fitness for each possible assignments of `table1.field` as well as `table2.field`.
Abstractly this could be described in the following way:

```
min_fitness = Null
foreach row1 in table1:
    foreach row2 in table2:
        fitness = query.get_distance(row1, row2)
        if fitness < min_fitness:
            min_fitness
return min_fitness
```

So in this hypothetical case, where no optimizations are applied we the total amount of query plan analyses (in `get_distance`) can be calculated by

$$c(T) = \prod_{t \in T} |t| \tag{4.1}$$

where T is the set of all tables and the norm $|t|$ is the amount of rows in a table.

For our convenience we set the maximum amount of rows m_r equal for all tables. Therefore we can construct our worst case scenario/an upper bound for eq. (4.1) by assuming all tables are filled completely:

$$c(T) = {m_r}^{|T|} \tag{4.2}$$

where $|T|$ is the total number of tables.
If we now look at a constructed example with

- 27 coverage targets,
- 6 tables
- 4 max rows per target

In the single objective case the total amount of comparisons is

$$\underbrace{27}_{\text{Coverage Targets/Single Optimization Executions}} \cdot \underbrace{4^6}_{\text{Equation (4.2)}} \cdot \mathit{generations} \cdot \mathit{individuals} \tag{4.3}$$

while in the multi-objective case we construct the amount of comparisons the following way

$$\underbrace{1}_{\text{Optimization Execution}} \cdot \underbrace{(27 \cdot 4)^6}_{\text{Equation (4.2)}} \cdot \mathit{generations} \cdot \mathit{individuals}. \tag{4.4}$$

There in this simple example the ratio of comparisons is

$$\frac{\text{Comparisons MOO}}{\text{Comparisons SO}} = \frac{(27 \cdot 4)^6}{27 \cdot 4^6} = 14,348,907 \tag{4.5}$$

Of course this example is constructed and does not reflect all the cases we have in our distribution (see fig. 4.1). But even if we lower the amount of coverage targets to 3, this ratio still would be 243.

Empircal Study

In order to check our assumptions we recorded the actual amount of comparisons we did when calculating the fitness. We identified the following piece of code[1] to be crucial for our study:

Listing 4.1: Actual implementation of the previous presented loop

```
for (ComparisonRow c : iterStore.getRows()) {
        try {
                currentDistance = c.getDistance();
        } catch (OperationNotSupportedException e) {
                log.error(e);
                currentDistance = Double.MAX_VALUE;
        }

        [...]
}
```

Here we use the size of `iterStore.getRows()` in order to determine the total amount of executed fitness plans. Compared to the previous nested `foreach`-loops in lis. 4.1 all executed fitness plans are stored in one iterable list `iterStore`.
If we assume our instrumentation doesn't do any optimization and we have full rows the size of the list would be calculated according to eq. (4.1).
We recorded the size of this list for two different queries. The properties and results can be found in table 4.2. Here we can see that luckily our previously calculated ratio of $14,348,907$ (See eq. (4.5)) was not fully reached.

[1] See for more context

Table 4.2: Empirical comparison of the executed query plans

Query	Tables	Targets	Max Rows	Single-Objective		Multi-Objective		Factor/Magnitude	
				Average	Max	Average	Max	Average	Max
Simple	3	11	4	4.82	21.00	1273.99	9246.00	264.21	440.29
Complex	6	27	4	14.17	460.00	16619.06	863391.00	1173.07	1876.94

4.3.2 Consequences

In our analysis in section 4.3.1 we showed why the maximum row size matters in our specific optimization problem. As we didn't expect that the impact of increasing the maximum row size is that big we couldn't foresee that we would run into such problems.

Nonetheless, to fasten up this issue we tested multiple things. Unfortunately all without any significant improvement. Please note that the instrumented database application was already operating in "in-memory" memory mode.

First, we tested increasing the cache size of the database, hoping we can have faster access.

Second, we reduced the amount of "back up" writes to the disk.

Unfortunately there was no further time to investigate the instrumentation and how to possibly improve it. As the instrumentation was added by the authors of the original paper with small maximum row sizes in mind it is no surprise that it isn't optimized for relatively large maximum sizes.

Chapter 5

Conclusion

The goal of our paper was an implementation of MOO version of EvoSQL, analysis and comparison of results and the answer to suitability of MOO for test case generation for SQL. In conclusion, we successfully implemented the MOO approach, but it is not applicable to, because the execution of fitness function is too much compared to GA so that the time is out of scope. our model expanded rows to be proportional to the number of objectives, it takes lots of cost. Adding combine operator also increased execution time since the number of rows increased. If we would have more time on this project, we suggested future directions.

First, since our MOO approach with existing fitness function was not suitable, we need to define another numeric fitness function to be used for MOO. Second, when we find a solution for a particular coverage target, we have kept it and it is a waste of resource. Therefore, if we remove that solved target from the amount of objectives, it reduces the amount of executions. Third, our basic expectation was, there might be unnecessary time budget to solve infeasible coverage targets if the users didn't eliminate infeasible coverage targets. EvoSQL also removed those infeasible targets manually. We assumed that MOO can handle them without manual elimination, since it simultaneously covers multiple coverage targets and detect them if it analyzes the tendency of non-decreasing fitness function. But, we expected it is hard to judge that a specific coverage target is difficult to solve or infeasible case. Therefore, we need a robust tendency analysis for each coverage target.