Final Report for AI based SE

ABC – 2018xxxx   Jeonggwan Lee – 20184400   GHI – 2018xxxx

Nick Heppert – 20186505

December 17, 2018

Chapter 1

Introduction

The amount of information is exponentially increasing, the 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 figure out this query statement is semantic or not. If the query statement is simple, developers can make test cases manually considering coverage targets, however 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(MOOP) methods. First, 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 of hard coverage targets, and coverage targets don't share their semantic discovery to others because of limitation of single target strategy. Second, we thought there might be unnecessary time budget to solve to solve infeasible coverage targets if the users didn't eliminate infeasible coverage targets. EvoSQL also removed those infeasible targets manually. We assumed that MOOP 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.

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

The goal of our paper is implementation of MOOP version of EvoSQL, analysis and comparison of results, answer to suitability of MOOP for test case generation for SQL.

Our conclusion is, MOOP of EvoSQL is not applicable to, future work are ...1. add new fitness function 2. find a solution for a coverage target and save it, then remove that target from the pool of objectives

(need to add our contribution, invention, results) 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 MOOP 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 5.

Chapter 2

Problem Formulation

Chapter 3

Methodology

Chapter 4

Evaluation

After successfully implementing our MOOP extension we started to conduct various experiments.

4.1   Setup

We used the previous serialized data (see **??**) from the three different projects that were provided with the Github repository[1] 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 to 30 mins.

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).

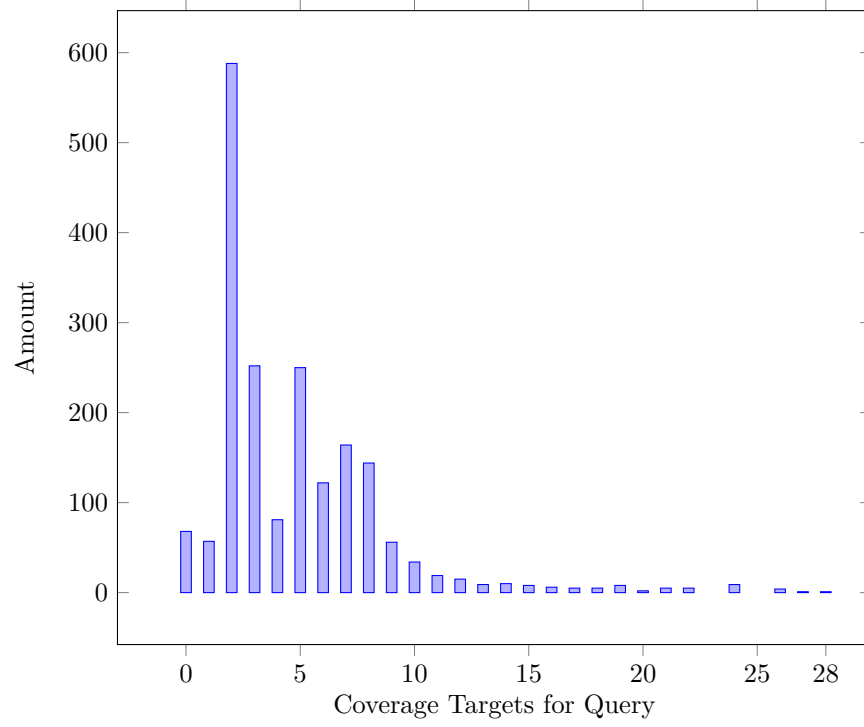---

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

Figure 4.1: Distribution of coverage targets (i.e. Objectives) in our whole test query dataset
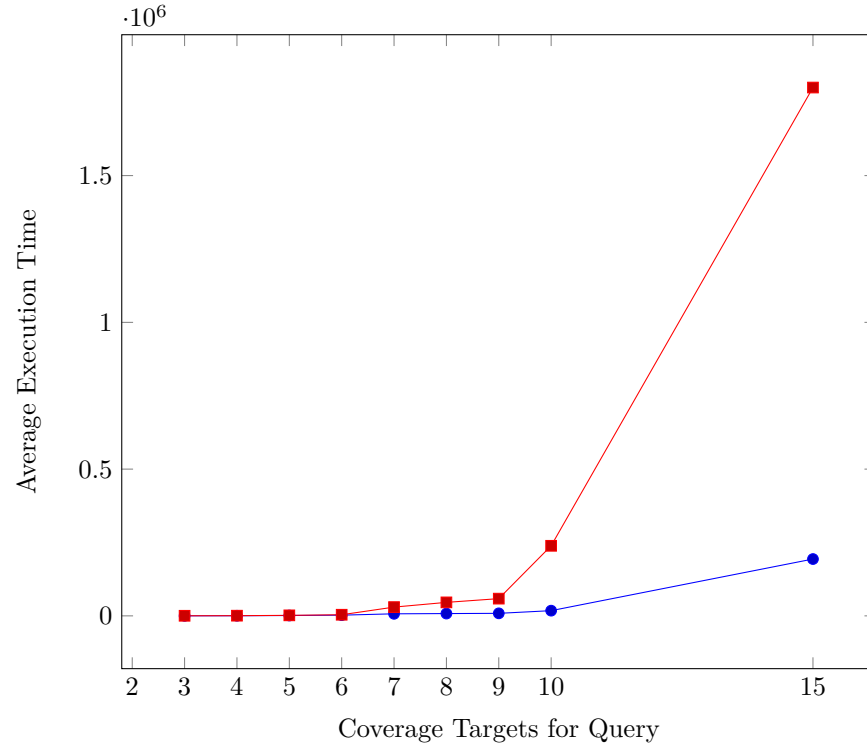

Figure 4.2: Average execution time for each covered target. For the last data point (15 coverage targets, our MOO was able to only cover 3 targets in the given time frame of 30 mins.)

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 section 4.3 we can see that the actual ratio between 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.

To further investigate why this happens we took a look at the execution time of each components. 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 the form of like `WHERE table1.field == "Foo" AND table2.field == "Bar"` we have to calculate the minimum fitness.

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 optimization are applied we could boil the total amount of query plan analyzations (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.

7

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 by assuming all tables are filled completely we have the worst case scenario for eq. (4.1) of

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

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

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

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

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

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 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[2] to be vital for our study:

```
for (ComparisonRow c : iterStore.getRows()) {
        try {
                currentDistance = c.getDistance();
                /*if (currentDistance == 0) {
                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. If we assume our instrumentation does not prior optimization and we have full rows this value would be calculated according to eq. (4.1).

We recorded the size of the row-array for two different queries the properties and results can be found in section 4.3.1. Here we can see that luckily our previously calculated ratio of $14,348,907$ (See eq. (4.5)) was not fully reached.

---

[2] 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