Many-objective Test Database Generation for SQL

Zhilei Ren[1], Shaozheng Dong[1], Xiaochen Li[2], Zongzheng Chi[1], and He Jiang[1]

[1]School of Software, Dalian University of Technology
[2] University of Luxembourg
zren@dlut.edu.cn dsz201493078@mail.dlut.edu.cn
xiaochen.li@uni.lu czz.dut@163.com jianghe@dlut.edu.cn

Abstract. Generating test database for SQL queries is an important but challenging task in software engineering. Existing approaches have modeled the task as a single-objective optimization problem. However, due to the improper handling of the relationship between different targets, the existing approaches face strong limitations, which we summarize as the inter-objective barrier and the test database bloating barrier. In this study, we propose a two-stage approach **MoeSQL**, which features the combination of many-objective evolutionary algorithm and decomposition based test database reduction. The effectiveness of **MoeSQL** lie in the ability to handle multiple targets simultaneously, and a local search to avoid the test database from bloating. Experiments over 1888 SQL queries demonstrate that, **MoeSQL** is able to achieve high coverage comparable to the state-of-the-art algorithm **EvoSQL**, and obtain more compact solutions, only 59.47% of those obtained by **EvoSQL**, measured by the overall number of data rows.

Keywords: Test Database Generation, Search Based Software Engineering, Many-objective Optimization.

1 Introduction

Recent years have witnessed the emergence and the rapid development of evolutionary computation based test case generation research [1, 2]. Especially, due to the importance in database-centric applications, test database generation for SQL queries has gained great research interest [3, 4]. The idea is to construct test databases, in pursuit of certain coverage criteria, such as to exercise all branches (also known as targets, see Section 2 for details) that can be executed in the SQL query. Due to the intrinsic complexity of SQL features, e.g, JOINs, predicates, and subqueries, test database generation for SQL queries can be difficult and time-consuming.

In the existing studies, this problem has been modeled as an optimization problem. Various approaches such as constraint solving and genetic algorithm have been employed to solve the problem [3, 4, 5]. Among these approaches, **EvoSQL** [3], a search-based algorithm, achieves the state-of-the-art results. **EvoSQL** features the support for the SQL standard, and has been evaluated over a set of real-world SQL queries.

However, despite the promising results accomplished, we could observe significant limitations in the existing studies. For example, **EvoSQL** models the test database generation problem as a single-objective problem, by designing an objective function that aggregates the coverage over all the branches. Consequently, such problem solving mechanism may face great challenges, which are summarized as follows.

(1) Inter-objective relationship barrier: taking **EvoSQL** as an example, to achieve satisfactory coverage, the underlying genetic algorithm has to be executed for multiple times, to cover each branch in a sequential way. Hence, a solution from one pass of evolution could not take all the branches into account. Also, the solutions within one evolution process could not help improve the other independent runs of evolution [6].

(2) Test database bloating barrier: **EvoSQL** achieves the branch coverage by merging the test databases obtained by the multiple executions of the genetic algorithm. The final test database may suffer from scalability issues [7], due to the improper handling of the relationship between different targets. Although **EvoSQL** adopts a post-process for reduction, chances are that the reduced test databases are still of large size.

To overcome these challenges, we propose a two-stage algorithm **MoeSQL** (**M**any-**o**bjective **e**volutionary algorithm for **SQL**) in search of better test data. More specifically, to tackle the inter-objective relationship barrier, in the first stage, we adopt a many-objective evolutionary algorithm to avoid redundant computation. The many-objective algorithm features a corner solution based sorting mechanism, with which we are able to cover multiple targets in a single evolution process.

To tackle the test database bloating barrier, we further leverage the solutions obtained from the first stage. We decompose the original problem into a series of subproblems, and employ a local search operator to achieve better solutions. Due to the reduction of the search space, it is easier to obtain more compact test database.

By combining the two stages, we develop an integrated framework **MoeSQL**. To evaluate **MoeSQL**, we consider real-world datasets for experiments, with 1888 SQL queries [3]. Extensive experiments demonstrate that with the many-objective evolutionary algorithm, **MoeSQL** is able to obtain high target coverage of 99.80%, which is comparable to the state-of-the-art approach **EvoSQL**. Meanwhile, with the reduction stage, **MoeSQL** obtains much more compact test databases, only 59.47% of those provided by **EvoSQL**, measured by the overall number of data rows for all the instances.

The main contributions of this paper are as follows:

(1) A many-objective search method is proposed for test database generation of SQL queries. To the best of our knowledge, this is the first study that solves this problem with a many-objective approach.

(2) We propose a novel decomposition based local search algorithm to address the test database bloating issue in SQL test database generation.

(3) We implement a prototype of **MoeSQL**. The prototype system and the experiment data are available at https://github.com/TheSecondLoop/MoeSQL.

(4) We conduct extensive experiments to demonstrate the effectiveness of **MoeSQL** compared with the state-of-the-art algorithm.

The rest of the paper is organized as follows. Section 2 describes the background of test database generation for SQL queries with a motivating example. Section 3 introduces the proposed approach. The empirical study is presented in Section 4. Finally, the conclusion and future work are given in Section 5.

2 Background and Motivating Example

2.1 Coverage Criteria

For the test database generation task, we intend to populate a set of databases based on certain coverage criteria. Considering the following SQL query *S* as an example:

```
SELECT * FROM
    Ta JOIN Tb ON Ta.p = Tb.q          -- step 1
    WHERE (Ta.a = 1) OR (Ta.b = 2);    -- step 2
```

In the query *S*, both columns **a** and **b** are non-nullable. To thoroughly test *S*, we adopt the SQL full predicate coverage criteria [8], which is inspired by the modified condition decision coverage [9] in software testing studies. The underlying idea is that given a SQL query, all the possible conditions which contribute to the query should be tested. For example, if we combine the modified conditions of the predicates in the **WHERE** clause of *S* with two predicates, we obtain six queries, generated by the SQL analysis tool SQLFpc [8]. More specifically, the predicates "Ta.a = 1" and "Ta.b = 2" correspond to targets 1-3 and 4-6, respectively:

```
(1) SELECT * FROM Ta JOIN Tb ON Ta.p = Tb.q WHERE (Ta.a = 0) AND NOT (Tb.b = 2);
(2) SELECT * FROM Ta JOIN Tb ON Ta.p = Tb.q WHERE (Ta.a = 1) AND NOT (Ta.b = 2);
(3) SELECT * FROM Ta JOIN Tb ON Ta.p = Tb.q WHERE (Ta.a = 2) AND NOT (Ta.b = 2);
(4) SELECT * FROM Ta JOIN Tb ON Ta.p = Tb.q WHERE NOT (Ta.a = 1) AND (Ta.b = 1);
(5) SELECT * FROM Ta JOIN Tb ON Ta.p = Tb.q WHERE NOT (Ta.a = 1) AND (Ta.b = 2);
(6) SELECT * FROM Ta JOIN Tb ON Ta.p = Tb.q WHERE NOT (Ta.a = 1) AND (Ta.b = 3);
```

With these targets, the next goal is to construct a set of test databases, so that each of the six queries, when applied on the test databases, retrieves non-empty result. If such goal is accomplished, it is claimed that the test databases have achieved complete coverage on the SQL query under test.

2.2 Test Database Generation

In this study, we focus on search-based test database generation. In these approaches, a common technique is to encode the test databases as candidate solutions, and model the objective function based on certain coverage criteria. For example, **EvoSQL** uses the concept of physical query plan [10] to divide each target into several execution steps. The objective function of the test database is determined according to its performance on each execution step. More specifically, the search problem is defined as follows:

Problem 2.1: (single-objective model) Let $R = \{r_1, \dots, r_k\}$ be the set of coverage targets of the SQL query under test. Find a set of test databases $D = \{t_1, \dots, t_k\}$ to cover all the coverage targets in R, i.e., one that minimizes the following objective function:

$$\min F(D, R) = \sum_{i=1}^{k} step\_level(t_i, r_i) + step\_distance(t_i, L), \quad (1)$$

where $step\_level(t_i, r_i)$ denotes the number of steps that are not executed, and $step\_distance(t_i, L)$ is the distance of t_i in satisfying the first unsatisfied step L.

To explain the objective function, consider the distance of target 2 (**SELECT** * **FROM** Ta **JOIN** Tb **ON** Ta.p = Tb.q **WHERE** (Ta.a = 1) **AND NOT** (Ta.b = 2)) and **db 1** in Fig. 1(a). In the physical query plan of S, target 2 can be divided into two steps: the first step considers the predicate in the **FROM** clause, and then the predicate in the **WHERE** clause (see the comments in S). The predicate in the **FROM** clause could be satisfied by **db 1**. In **db 1,** an empty result is returned when the predicate in the **WHERE** clause is examined. Hence, there are no unexecuted steps, i.e., $step\_level(t_i, r_i) = 0$. Meanwhile, in **db 1,** the predicate "Ta.a = 1" in the **WHERE** clause is not satisfied. According to the predicate, we choose the closest value 0 in column **a** of **db 1**. Then, the step distance is calculated as $step\_distance(t_i, L) = |0 - 1| = 1$ [11]. In this way, we can calculate the distance between the test database and the coverage target. Further details about the objective function evaluation could be found in reference [3].

	Ta			Tb
	Ta.p	Ta.a	Ta.b	Tb.q
db 1	1	0	1	1
db 2	1	1	1	1
db 3	1	2	1	1
db 4	1	0	1	1
db 5	1	0	2	1
db 6	1	0	3	1

(a) Test databases obtained by **EvoSQL**

	Ta			Tb
	Ta.p	Ta.a	Ta.b	Tb.q
db 7	1	0	1	1
	1	1	1	
	1	2	2	
	1	2	3	

(b) A more compact test database

Fig. 1. Example of solutions for query S

Obviously, the objective function is essentially an aggregate form of a multi-objective problem. Typically, existing approaches such as **EvoSQL** optimize each term of the summation in Eq. 1 with respect to each target, in a sequential way. The number of test databases equals to the number of coverage targets. For example, for query S, **EvoSQL** executes the underlying genetic algorithm six times, and generates six test databases, each with one row for **Ta** and **Tb**, respectively. However, the single-objective model may face obvious challenges:

(1) Inter-objective relationship barrier: In the SQL query S, targets 1-3 share the same predicate "Ta.b = 2". During the evolution towards target 1, the solutions obtained during the search procedure may also partially satisfy some predicates of targets 2-3. Although **EvoSQL** uses the population of the previous pass of evolution as the initial population for the next pass, the performance of this approach may depend on the invocation sequence of the underlying genetic algorithm. Consequently, single-objective approach cannot deal with the inter-objective relationship properly.

(2) Test database bloating barrier: In Fig. 1(b), we present a more compact solution (**db 7** with five data rows) that satisfies all the targets of the query S. Compared with the results of **EvoSQL**, **db 7** has the same coverage but less data rows. Interestingly, although **EvoSQL** is equipped with a reduction operator, the results in Fig. 1(a) could not be further simplified.

3 Our Approach

In order to tackle the two challenges of the existing algorithms, we propose our two-stage algorithm **MoeSQL**. In the first stage, the algorithm takes the coverage target generated by SQLFpc as the input, and obtains multiple databases to cover different targets. These databases serve as an intermediate solution to the problem. In the second stage, we use these solutions to divide the problem into sub-problems, and solve the induced problems to achieve a more compact solution.

3.1 Many-objective Test Database Generation

To generate test database with many-objective algorithms, we first modify the problem definition in Section 2 as follows.

Problem 3.1: (many-objective model) Let $R = \{r_1, \dots, r_k\}$ be the set of coverage targets of the SQL query under test. Find a test database t to cover as many coverage targets in R as possible, and keep the test database compact, i.e., minimize the following $k + 1$ objectives:

$$\min F'(t, R) = (d(t, r_1), d(t, r_2), \dots, d(t, r_k), size(t))^T, \quad (2)$$

where $d(t, r_i) = step\_level(t, r_i) + step\_distance(t, L)$ denotes the distance between the test database t and the coverage targets r_i as in Eq. 1. The extra objective $size(t)$ represents the number of data rows in the test database t. The superscript T represents transpose of vector.

The pseudo code of **TestDatabaseGen** is presented in Algo. 1. The workflow is similar with most existing many-objective algorithms. In Lines 1-3, a set of solutions are initialized. More specifically, each solution is encoded as a set of tables, each of which corresponds to a schema involved in the targets. We extract the constant values in the targets, and assign the constant values to the fields in initial solutions with probability p_s. Otherwise, the value for the field is initialized by a random value with probability $(1 - p_s)$ [12].

Then, in the main loop (Lines 4-15), the evolution process consists of the evaluation, sorting, selection, and reproduction procedures. For the evaluation procedure, we apply Eq. 2 over each solution, to calculate the objective values. In particular, we adopt a dynamic objective strategy [13], i.e., if there exists any new target that can be covered by a solution, we keep the solution and remove the target from the objective evaluation. With this strategy, we are able to deal with a relatively large number of objectives. For the sorting and the selection procedures, we consider the many-objective sorting mechanism used in MOSA [6, 14], a well-known many-objective algorithm in the search-

based software engineering community. The sorting mechanism features the multi-level prioritization of the solutions. Within the sorting procedure, the population is categorized the into levels. For the first level, we consider the best solutions (corner solutions) with respect to each objective. Then, the next level comprises the non-dominated solutions for the rest solutions. This process continues, until all the solutions are iterated. With this mechanism, the search could be guided towards covering more targets. During the selection, the elitism strategy is considered, i.e., only when one level is selected, we consider the solutions in the next level. In the same level, the tournament selection [15] is applied, so that both intensification and diversification are considered.

Algorithm 1: **TestDatabaseGen**
Input: coverage set R, population size pop_num, seeding probability p_s,
crossover probability p_c, mutation probability p_m
Output: a set of test database D
1 $seed \leftarrow seed\_extract(R)$
2 $archive \leftarrow \{\}$
3 $population \leftarrow initialization(pop\_num, seed, p_s)$
4 while stopping criterion not met
5 $evaluation(population, R)$
6 if there exists r_i covered by $population_j$
7 $archive \leftarrow archive \cup population_j$
8 $R \leftarrow R \setminus \{r_i\}$
9 end if
10 $population \leftarrow many\_objective\_sort(population)$
11 $population \leftarrow selection(population, pop\_num)$
12 $offspring \leftarrow crossover(population, p_c)$
13 $offspring \leftarrow mutation(offspring, p_m, seed)$
14 $population \leftarrow population \cup offspring$
15 end while
20 return $archive$

As for the reproduction operators, we directly adopt the crossover and the mutation operators of **EvoSQL** for simplicity, and no special modifications regarding many-objective algorithms are made in these operators. However, in our preliminary experiment, we find these operators are effective in general. When the stopping criterion is met, the evolution terminates. Finally, the archived solutions are regarded as the set of test databases.

To summarize, we compare **TestDatabaseGen** with the genetic algorithm used in **EvoSQL**. The approach proposed in this study features the following characteristics:

(1) Many-objective model: unlike the existing approaches in which test database generation is modeled as a single-objective problem, **TestDatabaseGen** adopts a many-objective sorting mechanism, so that the solutions in the population could take all the targets into consideration during the selection. Furthermore, in contrast to **EvoSQL** in which the objective values have to be calculated for all the targets separately, **TestDatabaseGen** could handle all the targets in a single evaluation. Hence, redundant computation could be prevented to some extents.

(2) Dynamic objective strategy: instead of applying static objective function along the evolution process, **TestDatabaseGen** dynamically removes targets that have been

covered. With this strategy, the number of targets decreases along the evolution process, and the search could be focused on the uncovered targets. Consequently, the algorithm scales up well to a relatively large number of targets.

3.2 Sub-problem Decomposition based Reduction

In the second stage, we focus on the test database bloating barrier. To reduce the size of the test database obtained by **TestDatabaseGen**, we develop a decomposition based local search strategy.

The idea is intuitive, i.e., when a candidate database covers one or more targets, it means that there are a series of data rows in the database that can satisfy the predicates in the SQL queries. However, it is possible that not all the data rows are contributive to the coverage. In other words, only a part of the data rows leads to the satisfaction of the predicates. Hence, we need to filter out the values with no contribution, and generate more compact test databases. To realize the reduction effect, we consider the following problem:

Problem 3.2: Let $D = \{t_1, \dots, t_m\}$ be a set of test databases. For each database t_i, $f(t_i) = \{r_{i1}, \dots, r_{in}\} \subseteq R$ represents the targets covered by t_i. Find a subset of databases $T' = \{t'_1, \dots, t'_c\}$ that minimizes the following function:

$$\min \sum_{i=1}^{c} size(t'_i), \quad s.t. \ \bigcup_{i=1}^{c} f(t'_i) = \bigcup_{i=1}^{m} f(t_i), \tag{3}$$

where $size(t'_i)$ indicates the number of data rows in the test database t'_i.

Unfortunately, with the increase of the targets, the number of data rows in the database T will increase accordingly, which leads to the search space explosion problem [16]. Therefore, we propose a decomposition strategy to transform the original problem into a set of sub-problems. Given two databases t_1 and t_2, we can construct a sub-problem, in search of a database with more compact size in a small neighborhood. More specifically, the sub-problem is defined as follows.

Problem 3.3: Let $D = \{t_1, t_2\}$ be a set of two test databases. For each database t_i, $f(t_i) = \{r_{i1}, \dots, r_{in}\} \subseteq R$ represents the targets covered by t_i. Find a new database t' that minimizes the following function:

$$\min size(t'), \quad s.t. \ f(t') = f(t_1) \cup f(t_2) \tag{4}$$

In this way, we can find the solution of the original problem by solving the sub-problem for each pair of test databases.

The main workflow of the second stage is presented in the pseudo code of Algo. 2 **TestDatabaseReduction**. In the main loop, we set all the solutions in the population as unreached, to indicate whether the solution should be involved in the generation of the next sub-problem. In Lines 3-4, we select two individuals in the population to construct the sub-problem. Then, the **LocalSearch** operator is applied, to obtain a solution to the induced sub-problem. In Lines 6-9, we verify the solution obtained by the local search operator. If a more compact solution is achieved, the two individuals under examination

will be replaced with the reduced solution returned by **LocalSearch**. Otherwise, we continue investigating other pairs of individuals that have not been investigated, until all the individuals have been reached.

In particular, our method adopts a local search operator to solve the induced sub-problem. As presented in Algo. 3, a hill climbing approach is considered.

Algorithm 2: **TestDatabaseReduction**
Input: databases D, coverage set R
Output: Best solution
1 $population \leftarrow D$
2 $unreached \leftarrow \{\langle t_i, t_j\rangle |\ t_i, t_j \in D, i < j\}$
3 while $unreached$ not empty
4 select a database pair $\langle t_1, t_2\rangle$ from $unreached$
5 $t' \leftarrow$ **LocalSearch**(t_1, t_2, R)
6 if $size(t') < size(t_1) + size(t_2)$
7 $population \leftarrow population \setminus \{t_1, t_2\} \cup t'$
8 $unreached \leftarrow unreached \cup \{\langle t_i, t'\rangle | t_i \in population\}$
9 $unreached \leftarrow unreached \cap \{\langle t_i, t_j\rangle |\ t_i, t_j \in population, i < j\}$
10 else
11 $unreached \leftarrow unreached \setminus \{\langle t_1, t_2\rangle\}$
12 end if
13 end while
14 return $population$

Algorithm 3: **LocalSearch**
Input: database t_1, database t_2, coverage set R
Output: reduced database
1 $t^* \leftarrow merge(t_1, t_2)$
2 while $size(t) \geq size(t_1) + size(t_2)$ and t^* changed in while
3 $t \leftarrow t^*$
4 for each data row r of t
5 if $evaluation(\{t \setminus \{r\}\}, R)$ not deteriorated
6 $t \leftarrow t \setminus \{r\}$
7 end if
8 end for
9 for each data row r of t
10 if $evaluation(\{t^* \setminus \{r\}\}, R)$ not deteriorated
11 $t^* \leftarrow t^* \setminus \{r\}$
12 break for
13 end if
14 end for
15 end while
16 return t

In Algo. 3, a first-improvement local search is realized. More specifically, we construct an incumbent database by merging the two input databases (Line 1). Then, we iteratively examine each data row of the incumbent database (Lines 2-15). If we observe that, the deletion of a data row does not deteriorate the coverage metric, we simply delete this data row to generate a new database (Line 5-7). Otherwise, we recover the

deletion, and make a perturbation accordingly (Lines 9-14). Then, we restart the investigation from the perturbed database. The traversal continues, until all the data rows have been iterated. By embedding the local search operator in Algo. 2, we are able to accomplish the reduction of the test databases.

As a brief summary, in this section, we present the **TestDatabaseReduction** stage. The reduction algorithm features a hill climbing based local search operator to explore the possibility of minimizing the test databases obtained by the first stage. In the next section, we would conduct extensive experiments to evaluate the proposed approach.

4 Experimental Results

4.1 Research Questions

In this section, we investigate the performance of **MoeSQL**. Our experiment focuses on the following three Research Questions (RQs).

RQ1: How does **MoeSQL** perform in terms of coverage metrics?

RQ2: How does **MoeSQL** perform in terms of the runtime and the size metrics?

RQ3: How does **MoeSQL** performs over different instances?

In these RQs, **RQ1** is used to verify the feasibility of **MoeSQL**. **RQ2** is adopted to examine whether our algorithm tackles the existing challenges properly. **RQ3** intends to investigate the trade-off between runtime and size metrics achieved by **MoeSQL**.

To evaluate **MoeSQL**, we adopt **EvoSQL**, the state-of-the-art algorithm as the baseline of our experiments. Besides, we also propose a variant algorithm (denoted as **MoeSQLv**) as a comparative approach. In this variant, **MoeSQL** will terminate after the first stage, without further consideration of the scalability issue. In this way, we can investigate the contribution of both stages.

Table 1. Statistics of the benchmark instances

Feature \ #targets	0	1-2	3-4	5-6	7-8	9-10	11-15	16-20	21+
Predicates	57	1278	424	54	27	10	14	21	3
JOINs	1831	41	3	1	11	1	-	-	-
Subqueries	1851	37	-	-	-	-	-	-	-
Functions	1735	149	2	2	-	-	-	-	-
Columns	59	1271	413	85	16	13	14	7	10
Targets	-	645	337	370	310	95	55	27	49

In the experiments, the parameter settings follow those of **EvoSQL**. More specifically, we set the population size pop_num to 50. Seeding probability p_s is set to 0.5. Crossover probability p_c is set to 0.75. Due to the various operations in mutation operator, the mutation probability p_m is a set of numbers. The mutation probability for inserting, deleting, and duplicating operation is set to $1/3$, the row change mutation probability is set to 1, and the NULL mutation probability is set to 0.1. Our experiments run under a PC with an Intel Core i5 2.3 GHz CPU, 16 GB memory, and Windows 10. All algorithms are implemented in Java 1.8. Our experiments use three datasets provided

by **EvoSQL**. Over the instances, we execute each algorithm five times. There are 1888 SQL queries and 10338 coverage targets in total. The statistics of these SQL queries are shown in Table 1. Because SQLFpc may generate some targets that cannot be covered theoretically, we manually examine and delete these targets to ensure that the rest targets could be covered, given sufficient runtime.

4.2 Experimental Results

Investigation of RQ1. We first present the coverage statistics of the comparative approaches in Table 2. In the table, the first column indicates the number of targets. Columns 2-3 represent the instance coverage (number of fully covered instances). Columns 4-5 are the target coverage (number of covered targets). The coverage of **MoeSQLv** is the same as **MoeSQL**, because the second stage of **MoeSQL** does not alter the coverage metric. From the table, the following phenomena could be observed:

(1) **MoeSQL** achieves high coverage over all the instances. Similar as **EvoSQL**, **MoeSQL** can cover all targets over instances with less than 10 coverage targets. With the increase of the number of targets, the performance of both algorithms decreases.

Table 2. The instance coverage and the target coverage of each algorithm

#targets	Instance Coverage		Target Coverage	
	EvoSQL	**MoeSQL/MoeSQLv**	**EvoSQL**	**MoeSQL/MoeSQLv**
1-2	**645 / 645**	**645 / 645**	**1232 / 1232**	**1232 / 1232**
3-4	**337 / 337**	**337 / 337**	**1095 / 1095**	**1095 / 1095**
5-6	**370 / 370**	**370 / 370**	**1970 / 1970**	**1970 / 1970**
7-8	**310 / 310**	**310 / 310**	**2314 / 2314**	**2314 / 2314**
9-10	**95 / 95**	**95 / 95**	**892 / 892**	**892 / 892**
11-15	**53 / 55**	**53 / 55**	679 / 699	**686 / 699**
16-20	**26 / 27**	25 / 27	473 / 485	**481 / 485**
20+	42 / 49	**46 / 49**	1633 / 1651	**1647 / 1651**

(2) In terms of the target coverage, **MoeSQL** performs slightly better than **EvoSQL**. Over all the instances, **MoeSQL** is able to cover 99.80% of targets. Meanwhile, the target coverage by **EvoSQL** is 99.52%.

(3) In terms of instance coverage, the results of **EvoSQL** and **MoeSQL** are very close. However, the performance of the two algorithms is not the same. **EvoSQL** performs better over instances with more than 16 but less than 20 coverage targets. Meanwhile, **MoeSQL** has a higher coverage in instances with more than 20 coverage targets.
Answer to RQ1: MoeSQL can completely cover 99.63% of all instance, which is comparable to the state-of-the-art approach.

Investigation of RQ2. In this RQ, we are interested in the efficiency of **MoeSQL**. We calculate the runtime and the size of test database (measured by the number of data rows in the test databases). The statistics are presented in Table 3. The table is organized as follows. The first column indicates the number of targets of the queries. Columns 2-4 represent the median runtime statistics in seconds, for **EvoSQL**, **MoeSQLv**, and **MoeSQL**, respectively. Similarly, columns 5-7 are associated with the size statistics,

measured by the average number of data rows in the test database, for the three approaches. From the table, we observe that:

Table 3. The runtime and the test database size statistics of each algorithm

#targets	Runtime (s)			Size (#data rows)		
	EvoSQL	**MoeSQLv**	**MoeSQL**	**EvoSQL**	**MoeSQLv**	**MoeSQL**
1-2	0.03	0.02	**0.02**	2.00	2.00	**2.00**
3-4	0.04	0.02	**0.02**	3.00	3.00	**3.00**
5-6	0.07	0.02	0.04	5.00	5.00	**5.00**
7-8	0.25	0.13	0.20	8.00	7.00	**7.00**
9-10	0.62	0.32	0.61	11.00	10.00	**9.00**
11-15	2.13	1.11	3.87	16.00	13.40	**11.00**
16-20	10.15	7.76	116.30	40.40	33.40	**14.40**
20+	130.32	108.36	1483.15	54.00	40.00	**26.00**

(1) **MoeSQLv** achieves the minimum runtime over all instances. The time of **MoeSQLv** is almost half that of **EvoSQL** in instances with less than 15 targets. And over other instances, the runtime of **MoeSQLv** is also significantly less than **EvoSQL**.

(2) **MoeSQL** performs the best over instances with less than 10 targets. Due to the second stage, the runtime of the whole algorithm is longer than **MoeSQLv**. With the increase of the number of targets, the gap between the two variants also increases.

(3) When considering all the instances, without the second stage, **MoeSQLv** is able to outperform **EvoSQL** by 22.62%, in terms of the size metric of the test database. Moreover, with the reduction mechanism, **MoeSQL** is able to further reduce the test database size by 17.91%. For the instances with more than 20 coverage targets, the overall number of data rows is reduced by up to 68.59%, compared with **EvoSQL**.

Answer to RQ2: the advantage of **MoeSQL** in runtime is more reflected over instances with small number of coverage targets. At the same time, **MoeSQL** can significantly reduce the size of the test database, especially for complex instances. Although the second stage of **MoeSQL** costs extra runtime, the local search operator reduces the size of the database. For most instances, the time consumption is acceptable.

Investigation of RQ3. To answer RQ3, we classify all instances according to the performance of each algorithm. According to the two performance indicators, i.e., runtime and size, we categorize all the instances into the following four types:

Type A: **MoeSQL** outperforms **EvoSQL** in terms of both indicators.

Type B: **MoeSQL** outperforms better than EvoSQL in terms of runtime, and the size of **MoeSQL** is the same as **EvoSQL**.

Type C: The size of **MoeSQL** is better than **EvoSQL**, but the runtime metric of **MoeSQL** is inferior to that of **EvoSQL**.

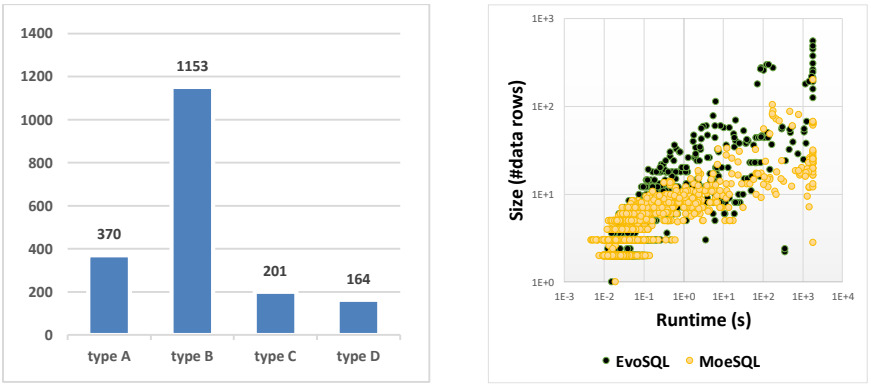
Type D: **MoeSQL** fails to outperform **EvoSQL** in terms of either indicator.

We summarize the number of instances of each type. The statistics of these type are shown in Fig. 2(a). According to the figure, we observe that:

(1) **MoeSQL** is more time efficient than **EvoSQL** over the majority (types A and B, 1523 / 1888) of instances. Over these instances, **MoeSQL** can find test databases of

same or more compact size than **EvoSQL**. In particular, over (370 / 1888) 19.60% of instances, **MoeSQL** outperforms **EvoSQL** for both performance indicators.

(2) Over (201 / 1888) 10.65% of instances, **MoeSQL** consumes more time than **EvoSQL**, but is able to achieve more compact solutions. Only over (164 / 1888) 8.69% of instances, **MoeSQL** is dominated by **EvoSQL**.



(a) Statistics of instance types (b) Runtime-size comparison

Fig. 2. Comparison between **EvoSQL** and **MoeSQL**

To gain more insights, we plot the runtime and the size metrics obtained by **EvoSQL** and **MoeSQL** over all the instances in Fig. 2(b). From the figure, we observe that the points for **MoeSQL** are concentrated in the area closer to the origin, which to some extents demonstrates the ability of **MoeSQL** to balance the runtime and the size.
Answer to RQ3: **MoeSQL** performs better than **EvoSQL** over most instances, and is able to achieve moderate trade-off between the runtime and the size metrics.

5 Conclusion and Future Work

In this paper, we present a novel two-stage algorithm **MoeSQL** to solve the test database generation for SQL queries. The proposed approach features the combination of a many-objective evolutionary algorithm and a local search based reduction mechanism, to tackle the inter-objective barrier and the test database bloating barrier. Experimental results over real-world datasets demonstrate the effectiveness of **MoeSQL**.

Despite the promising results, there is still room for improvement. For example, the local search based reduction is time consuming. To mitigate the limitation, an interesting direction is to consider very large neighborhood search [17] or surrogate based acceleration mechanisms [18]. If feasible, the efficient reduction mechanisms may enable more advanced algorithms, such as reduction during evolution.

Acknowledgement. This work is supported in part by the National Key Research and Development Program of China under grant no. 2018YFB1003900, and the National Natural Science Foundation of China under grant no. 61772107, 61722202.

References

1. Fraser, G., Arcuri, A., McMinn, P.: Test suite generation with memetic algorithms. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, pp. 1437–1444. ACM, New York, (2013)
2. Arcuri, A.: RESTful API Automated Test Case Generation with Evo-Master. ACM Transactions on Software Engineering and Methodology 28(1), 1-37 (2019)
3. Castelein, J., Aniche, M., Soltani, M., Panichella, A., van Deursen, A.: Search-based test data generation for SQL queries. In: Proceedings of the 40th international conference on software engineering, pp. 1220-1230. ACM, Gothenburg, (2018)
4. Suárez-Cabal, M., de la Riva, C., Tuya, J., Blanco, R.: Incremental test data generation for database queries. Automated Software Engineering 24(4), 719-755 (2017)
5. Shah, S., Sudarshan, S., Kajbaje, S., Patidar, S., Gupta, B., Vira, D.: Generating test data for killing SQL mutants: A constraint-based approach. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 1175-1186. IEEE, Hannover (2011)
6. Panichella, A., Kifetew F., Tonella P.: Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. IEEE Transactions on Software Engineering 44(2), 122-158 (2018)
7. Tuya, J., de la Riva, C., Suárez-Cabal, M., Blanco, R.: Coverage-Aware Test Database Reduction. IEEE Transactions on Software Engineering 42(10), 941-959 (2016)
8. Tuya, J., Suárez-Cabal, M., de la Riva, C.: Full predicate coverage for testing SQL database queries. Software Testing, Veriication and Reliability 20(3), 237-288 (2010)
9. Chilenski, J., Miller, S.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal 9(5), 193-200 (1994)
10. Garcia-Molina, H., D Ullman, J., Widom, J.: Database system implementation. Prentice Hall, Upper Saddle River (2000)
11. Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering 16(8), 870-879 (1990)
12. Rojas, J., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. Software Testing, Verification and Reliability 26(5), 366-401 (2016)
13. Rojas, J., Vivanti, M., Arcuri, A., Fraser G.: A detailed investigation of the effectiveness of whole test suite generation. Empirical Software Engineering 22(2), 852-893 (2017)
14. Panichella, A., Kifetew F., Tonella P.: Reformulating Branch Coverage as a Many-Objective Optimization Problem. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1-10. IEEE, Graz (2015)
15. Goldberg, D., Deb, K. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In: Proceedings of the First Workshop on Foundations of Genetic Algorithms, pp. 69-93. Elsevier, Indiana (1991)
16. Ramírez, A., Romero, J., Ventura, S.: A survey of many-objective optimisation in search-based software engineering. Journal of Systems and Software 149, 382-395 (2019)
17. Ghoniem, A., Flamand, T., Haouari, M.: Optimization-Based Very Large-Scale Neighborhood Search for Generalized Assignment Problems with Location/Allocation Considerations. INFORMS Journal on Computing. 28(3), 575-88 (2016)
18. Pan, L., He, C., Tian, Y., Wang, H., Zhang, X., Jin, Y.: A classification-based surrogate-assisted evolutionary algorithm for expensive many-objective optimization. IEEE Transactions on Evolutionary Computation, 23(1), 74-88(2018)