Touchstone$^+$: Query Aware Database Generation for Match Operators

Hao Li[1], Qingshuai Wang[1], Zirui Hu[1], Xuhua Huang[1], Lyu Ni[1](✉), Rong Zhang[1], Peng Cai[1], Xuan Zhou[1], and Quanqing Xu[2]

[1] East China Normal University, Data Science and Engineering, Shanghai, China
{haoli@stu, qswang@stu, zrhu@stu, xhhuang@stu, lni@dase, rzhang@dase, pcai@dase, xzhou@dase}.ecnu.edu.cn
[2] OceanBase, Ant Group, Hang Zhou, China
xuquanging.xqq@oceanbase.com

Abstract. Query-aware database generator (QAGen) expects to generate an application scenario based on the anonymized query plans as well as the cardinality constraints of all operators. It prefers to have the similar performance to the in-production real performance if applying the generated workload on the generated database. Touchstone is the first work achieving simulating the application with the first 16 queries in TPC-H. However, it is designed based on heuristic rules, and has a weak ability to guarantee the cardinality constraints from match operators, i.e., IN and LIKE, which are important operators for performance optimization. So in this paper, we propose Touchstone$^+$ to solve the problem QAGen involving match operators by modeling constraints from IN and LIKE into a Constraint Programming (CP) problem. After solving the CP problem, it provides an initial data distribution satisfying cardinality constraints from all match operators for the iterative parameter search algorithm of Touchstone. Experiments have verified the effectiveness of our design and we also open code sources [12] for reproducing all results.

1 Introduction

To solve the problem of lacking real-world data for benchmarking, Query-aware Database Generator (QAGen) has been widely studied for performance debugging or query execution optimization [1,2,7,8,10], which is sensitive to queries with various cardinalities. Typically, it extracts the query template of each query plan, the cardinality constraint (the output size) of each operator as well as the schema information from the in-production environment as in Fig. 1a. Query template is generated by anonymizing the values in predicates of the query plan. The anonymized query templates with the cardinality constraints of operators and the schema information as in Fig. 1b act as the inputs to QAGen. It then generates a database and instantiates queries. Note that by running the instantiated queries on the generated database, it should guarantee the cardinality sizes from operators following the requirement of cardinality constraints from the in-production scenario as illustrated in Fig. 1c.

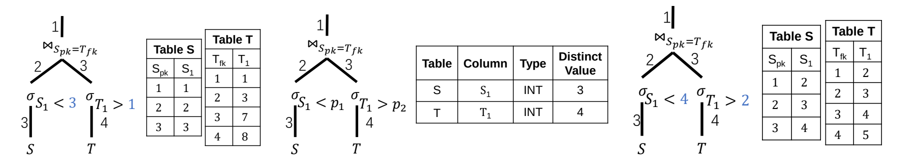


Table S

S_{pk}	S_1
1	1
2	2
3	3

Table T

T_{fk}	T_1
1	1
2	3
3	7
4	8

Table	Column	Type	Distinct Value
S	S_1	INT	3
T	T_1	INT	4

Table S

S_{pk}	S_1
1	2
2	3
3	4

Table T

T_{fk}	T_1
1	2
2	3
3	4
4	5

(a) Original Query and Database (b) Query Template and Schema Information (c) Generated Query and Database

Fig. 1: Process of Query-aware Database Generator

Among current QAGens [1,2,7,8,10], Touchstone [7] is the representative state-of-the-art work. But it is still weak in the support of some commonly used operators, especially the match operator, i.e., IN and LIKE. The IN operator is often named as the multi-value match operator and the LIKE operator is commonly referred to as the pattern-match operator. They may access a large size of data and affect performance in distributed DBMS [16,17], and much research work has been done to optimize these two operators [5,6,19]. Touchstone employs heuristic rules to generate data and instantiate queries. It does not catch the information inside IN and LIKE operators, but only transforms the cardinality constraints with IN operators into multiple = cardinality constraints by dividing the cardinality size equally and the cardinality constraint with LIKE operators into one = cardinality constraint. So its generation fidelity is far from satisfactory for match operators and the relative errors will accumulate as the increasing of match operators.

So in this paper, we provide a solution for QAGen with match operators (QAGM). QAGM is proved to be **NP-Complete**, which challenges finding a data distribution to satisfy all the cardinality constraints. Even if we have the data distribution, we are still challenged by instantiating parameters to satisfy match pattern requirements. The last challenge is to mitigate the computation complexity for a scalable generation solution. To solve these challenges, we design Touchstone$^+$ based on Touchstone and our main contributions are presented as:

1) We model query aware database generation for match operators (QAGM) as a Constraint Programming problem, which can be solved exactly.
2) We introduce a trie-based method to instantiate the parameters for satisfying the pattern requirements from match operators.
3) We propose two optimizations that simplify the complexity of the modeled Constraint Programming problem by trading off acceptable errors.
4) We implement Touchstone$^+$ by integrating these algorithms into Touchstone. Further, we run extensive experiments to verify our designs in Touchstone$^+$.

2 Preliminary

Consider a non-key column A in table R, we define the cardinality of R as its table size, i.e., $|R|$. Further, we donate the cardinality of A as its domain size, i.e., $|R|_A$. The i^{th} cardinality constraint (CC) is defined as the output size of i^{th} match operator, i.e., $\text{CC}_i : |\sigma_P(R)| = c_i$, where P is a predicate with a match operator and c_i is its output size. Specifically, we define P with IN operator as $A\ in\ (p_{i_1}, ..., p_{i_k})$, where $p_{i_j} (j \leq k)$ is its j^{th} parameter of CC_i. Similarly, we define P with LIKE operator as $A\ like\ p_i$, where p_i is its parameter. Throughout the paper, we parameterize all the values in the predicates and all the parameterized values are anonymized to 'p', e.g., $ab\%$ is anonymized as $p\%$.

Example 1. Consider a non-key column t_1 in table T, which contains 5 values, i.e., a, abc, abc, adc, b. Then the cardinalities of table T and column t_1 are $|T| = 5$ and $|T|_{t_1} = 4$. There are five example queries with match operators over t_1 and their corresponding output sizes are presented as follows.

```
select count(*) from T where t1 in (a, abc);       -- 3
select count(*) from T where t1 like a%;           -- 4
select count(*) from T where t1 like ab%;          -- 2
select count(*) from T where t1 like ad%;          -- 1
select count(*) from T where t1 like %c;           -- 3
```

Therefore, we can formulate the CCs for match operators by the equations in Eqn. 1. Among them, the predicate in CC_1 contains IN operator and the predicates in the others contain LIKE operator.

$$\text{CC}_1 : |\sigma_{t_1\ in\ (p_{1_1},\ p_{1_2})}(T)| = 3 \quad \text{CC}_2 : |\sigma_{t_1\ like\ p_2}(T)| = 4 \tag{1}$$
$$\text{CC}_3 : |\sigma_{t_1\ like\ p_3}(T)| = 2 \quad \text{CC}_4 : |\sigma_{t_1\ like\ p_4}(T)| = 1 \quad \text{CC}_5 : |\sigma_{t_1\ like\ p_5}(T)| = 3$$

We now give the problem definition of query aware database generation for match operators (QAGM) in Def. 1.

Definition 1. *Given the cardinalities of table and non-key column, and K cardinality constraints from match operators, i.e.,* IN *and* LIKE, QAGM *aims to generate a synthetic non-key column and instantiate parameters in the match operators, such that all the cardinality constraints are guaranteed if running the instantiated match operators on the synthetic non-key column.*

Example 2. Based on the cardinality constraints in Example 1, we present a feasible solution. Specifically, we can populate column t_1 with fxy, fgz, fgz, fhz, hy, and instantiate parameters in match operators as follows.

$$\text{CC}_1 : |\sigma_{t_1\ in\ (fxy,\ fgz)}(R)|{=}3 \qquad \text{CC}_2 : |\sigma_{t_1\ like\ f\%}(R)|{=}4 \tag{2}$$
$$\text{CC}_3 : |\sigma_{t_1\ like\ fg\%}(R)|{=}2 \quad \text{CC}_4 : |\sigma_{t_1\ like\ fh\%}(R)|{=}1 \quad \text{CC}_5 : |\sigma_{t_1\ like\ \%z}(R)|{=}3$$

According to Definition 1, we divide QAGM into following two sub-problems.

1. **Data Distribution Characterization.** Given the cardinality constraints on a non-key column, it requires characterizing a data distribution to ensure the existence of parameters that satisfy all these cardinality constraints.
2. **Parameter Instantiation.** Given a characterized data distribution and the cardinality constraints on a non-key column, it requires instantiating all the parameters to satisfy the cardinality constraints over this distribution.

Further, we analyze the computational complexity of QAGM. For simplicity, we consider the characterized data distribution of column A as an integer multiset U [3]. U contains items sized $|R|_A$, i.e., $x_1, ..., x_{|R|_A}$, where each item x_i represents the frequency of the corresponding value with $\sum_1^{|R|_A} x_i = |R|$. So, there are total $C_{|R|-1}^{|R|_A-1}$ possible distributions for column A. In Prop. 1, it proves *Parameter Instantiation* is **NP-Complete**. Therefore, we cannot determine whether a distribution is valid in polynomial-time. At worst, we must enumerate all distributions of the domain to find a valid distribution. It means that we have to deal with $C_{|R|-1}^{|R|_A-1} \times K$ NP-Complete problems, where K is the total number of cardinality constraints from match operators.

Proposition 1. *The decision problem version of Parameter Instantiation is* **NP-Complete** *in its computational complexity.*

Proof. To simplify, we denote the data distribution as an integer multiset U. First, consider the CC$_i$ of an IN operator and it contains k_i parameters and requires c_i output rows. *Parameter Instantiation* for the CC$_i$ is to decide whether a subset of distinct k_i items from U can sum up to exactly c_i. It is the classic fixed-size k-knapsack problem, which has been proved to be NP-Complete [4]. Second, consider the CC$_j$ of a LIKE operator, it requires c_j output rows. *Parameter Instantiation* for the CC$_j$ is to find a subset of distinct items from U which sum up to c_j. It is a classic subset sum problem, and its decision problem version is also proved to be NP-Complete [4]. In summary, Proposition 1 is proved.

3 Design Overview

Since QAGM is NP-Complete (proved in §2), then there is unacceptable complexity to search for a solution directly. Note that any NP-Complete problem is reducible to another one in polynomial time [4]. Since *Constraint Programming* (CP) problem has been well studied to model various classic NP-Complete problems [11] and has various CP-solvers [9], it inspires us to model QAGM into a CP problem. Specifically, QAGM solves two sub-problems, i.e., **Data Distribution Characterization** and **Parameter Instantiation**. We represent the data distribution of column A by a vector X. Specifically, x_j in X represents the frequency of the j^{th} value, and the total item frequency in X is the table size.

$$X = (x_1, x_2, ..., x_{|R|_A}) \qquad s.t. \quad \forall j \in [1, |R|_A] \quad x_j > 0 \qquad \textstyle\sum_{j=1}^{|R|_A} x_j = |R|$$

We propose a parameter status vector S_i to represent the parameter values assignment for i^{th} cardinality constraint, i.e., CC$_i$. Specifically, s_{i_j} in S_i represents whether the j^{th} value in X is the parameter of the i^{th} match operator. Then we can represent the requirement from CC$_i$ by $X \times S_i = c_i$. It means that the total frequencies of all the items (instantiated parameter values) for CC$_i$ is equal to the required output size c_i for the i^{th} match operator.

$$S_i = (s_{i_1}, s_{i_2}, ..., s_{i_{|R|_A}})^T \qquad s.t. \quad \forall j \in [1, |R|_A] \quad s_{i_j} \in \{0,\ 1\}$$

$$X = \begin{pmatrix} 1 & 2 & 1 & 1 \end{pmatrix}$$

$$S = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Fig. 2: A Valid Solution for Example 3

$$X = \begin{pmatrix} 1 & 2 & 1 & 1 \end{pmatrix}$$

$$S = \left(\begin{array}{|c|cccc} \cline{1-1} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ \cline{1-1} \end{array}\right)$$

Fig. 3: An Illegal Solution for Example 3

If we have K cardinality constraints, we organize their parameter status vectors into a parameter status matrix, i.e., $S = (S_1, S_2, ..., S_K)$. Correspondingly, we organize the cardinality requirements from all the cardinality constraints as a cardinality vector, i.e., $C = (c_1, c_2, ..., c_K)$. Therefore, we extend $X \times S_i = c_i$ for a single cardinality constraint to all the cardinality constraints by $X \times S = C$.

Example 3. For column t_1 of Table T in Example 1, it has $|T|_{t_1} = 4$ and its data distribution vector is $X = (x_1, x_2, x_3, x_4)$ with $\sum_{j=1}^{4} x_j = 5$ and $x_j > 0$. For CC$_1$, its status vector is $S_1 = (s_{1_1}, s_{1_2}, s_{1_3}, s_{1_4})^T$. Then we represent the cardinality constraint as $X \times S_1 = 3$. Finally, we represent all five cardinality constraints in Example 1 as $X \times (S_1,\ S_2,\ S_3,\ S_4,\ S_5) = (3,\ 4,\ 2,\ 1,\ 3)$.

After solving CP problem of $X \times S{=}C$, the result gives the frequency of each value in column and the assignment of each value to matching operators., i.e., whose corresponding status is labelled '1' in S_i for CC$_i$ from i^{th} match operator. For example, consider the valid solution in Fig. 2 for Example 3, the frequency of the first value is 1 and the value is assigned to p_{1_1} of CC$_1$ according to its status vector $S_1{=}(1, 1, 0, 0)$.

Since it may exist multiple solutions for a CP problem, to guarantee the validity of the solution, we design to formulate more parameter constraints in § 4. In § 5, we introduce the detailed algorithms for data population and parameter instantiation according to the solution of the CP problem. To reduce the computation cost, we provide two optimization techniques in § 6.

4 Parameter Constraints Formulation

Note that not any solution for the CP problem is meaningful. For example, Fig. 3 presents an invalid solution for Example 3. In the solution, three statuses in S_1 are set as 1, which means to populate three values to the match operator of CC$_1$. However, the population violates its parameter requirement because there are only two parameters (p_{1_1} and p_{1_2}) in CC$_1$. Therefore, we introduce **parameter constraints** (PCs) to specify the parameter instantiation requirement and limit the solution space for the CP problem formulated from match operators.

Specifically, for the i^{th} cardinality constraint (CC$_i$) from an IN operator, the calculated status vector S_i presents the values selected for its parameters. But the number of position labelled '1' in S_i should be equal to the number of parameters in the IN operator. Therefore, its constraint of a PC is expressed as $\sum_{j=1}^{|R|_A} s_{i_j} = k_i$ with $k_i > 0$. For example, CC$_1$ in Example 1 contains two parameters, so we have an additional parameter constraint PC$_1$: $\sum_{j=1}^{4} s_{1_j} = 2$.

Next, we formulate PC constraints from LIKE operators. LIKE operators can be classified into three categories based on the match pattern in predicates with str as the match string on column t_1 of table T: 1) prefix match operator, i.e., $\sigma_{t_1\ like\ str\%}(T)$, 2) suffix match operator, i.e., $\sigma_{t_1\ like\ \%str}(T)$, 3) arbitrary-subsequence match operator, i.e., $\sigma_{t_1\ like\ \%str\%}(T)$. We first discuss the PC constraints from prefix match operators and then extend to the other.

Consider u^{th} and v^{th} cardinality constraints (corresponding to CC$_u$ and CC$_v$) with prefix match operators as CC$_u$: $|\sigma_{t_1\ like\ p_u}(T)|$=$c_u$, and CC$_v$: $|\sigma_{t_1\ like\ p_v}(T)|$ =c_v. In general, let M_u (resp. M_v) denotes the values matched by p_u (resp. p_v) and $c_u \geq c_v$. There are only two possible relationships between the result sets M_u and M_v (proved in Theorem 1), which are:

(1) **Containment** relationship: one result is the subset of the other one, i.e., $M_u \cap M_v = M_v$. For example, $\underline{\sigma_{t_1\ like\ a\%}(T)} \cap \underline{\sigma_{t_1\ like\ ab\%}(T)} = \underline{\sigma_{t_1\ like\ ab\%}(T)}$.

(2) **Exclusion** relationship: there is no intersection between the two results, i.e., $M_u \cap M_v = \emptyset$. For example, $\underline{\sigma_{t_1\ like\ a\%}(T)} \cap \underline{\sigma_{t_1\ like\ b\%}(T)} = \emptyset$.

Theorem 1. *Given two cardinality constraints with prefix match operators on column A of table T, i.e., $|\sigma_{t_1\ like\ p_u}(R)|$=$c_u$ and $|\sigma_{t_1\ like\ p_v}(R)|$=$c_v$, where $c_u \geq c_v$. Let M_u (resp. M_v) denotes the values matched by p_u (resp. p_v), then the results of M_u and M_v satisfy $M_u \cap M_v = M_v$ or $M_u \cap M_v = \emptyset$.*

Proof. Suppose p_u (resp. p_v) has the prefix as $str_u\%$ (resp. $str_v\%$). We discuss three cases as follows.

1. str_u starts with str_v. It means that all the values in M_u are belong to M_v, i.e., $M_u \subseteq M_v$. Note that $c_u \geq c_v$, it means that $|M_u| >= |M_v|$. So we can infer that $M_u = M_v$, i.e., $M_u \cap M_v = M_v$.

2. str_v starts with str_u. It means that all the values in M_v belong to M_u, i.e., $M_v \subseteq M_u \Rightarrow M_u \cap M_v = M_v$.

3. str_u does not start with str_v and str_v does not start with str_u. It means that all the values in M_u do not belong to M_v and vice versa, i.e., $M_u \cap M_v = \emptyset$.

Denote the parameter status vectors for these two constraints are S_u and S_v. We present additional two PC constraints to satisfy above two relationships.

For **Containment** relationship, supposing the k^{th} value is matched by p_v, i.e., $s_{v_k} = 1$, the PC constraint requires the value must be matched by p_u, i.e., $s_{u_k} = 1$. Then we can have that if s_{v_k}=1, then $s_{u_k} \times s_{v_k}$=1 $= s_{v_k}$ while if $s_{v_k} = 0$, then $s_{u_k} \times s_{v_k} = 0 = s_{v_k}$. Then PC from a containment relationship between S_u and S_v can be formalized by the *hadamard product* as:

$$\forall j \in [1, |R|_A] \quad s_{u_j} \times s_{v_j} = s_{v_j} \quad \Rightarrow \quad S_u \odot S_v = S_v$$

For **Exclusion** relationship, supposing the k^{th} value is matched by p_v, i.e., s_{v_k}=1, the PC constraint requires the value cannot be matched by p_u, i.e.,

$s_{u_k} = 0$ and vice versa. In the same way, the PC constraint from an exclusion relationship between S_u and S_v can be formalized by the *hadamard product* as:

$$\forall j \in [1, |R|_A] \quad s_{u_j} \times s_{v_j} = 0 \qquad \Rightarrow \quad S_u \odot S_v = 0$$

According to Theorem 1, between any two PC constraints, there can only exist either a **Containment** or an **Exclusion** relationship of their status vectors, so we only take one of them into our CP problem to restrict the solution, which is formulated by the exclusive disjunction between two PC constraints as follows,

$$(S_i \odot S_j = S_j) \oplus (S_i \odot S_j = 0) = T \qquad s.t. \quad 1 \leq i < j \leq K \tag{3}$$

So there are additional $n(n-1)/2$ constraints (formulated by Equation 3) for total n cardinality constraints from the prefix match operators.

Example 4. In Example 1, there are three cardinality constraints with prefix match operator, i.e., CC_2, CC_3 and CC_4. Correspondingly, we formulate $\frac{3*(3-1)}{2} =$ 3 PC constraints for the CP problem in Example 3, which are:

$$\mathbf{PC_1}: (S_2 \odot S_3{=}S_3) \oplus (S_2 \odot S_3{=}0){=}T \;\; \mathbf{PC_2}: (S_2 \odot S_4{=}S_4) \oplus (S_2 \odot S_4{=}0){=}T$$
$$\mathbf{PC_3}: (S_3 \odot S_4{=}S_4) \oplus (S_3 \odot S_4{=}0){=}T.$$

For either the suffix match or arbitrary-subsequence match operators, they can be rotated to be prefix constraints, which can then follow the same way to construct the PC constraints. For example, we can rotate $\%b$ as $p_1(b)\%$ and $\%a\%c$ as $p_2(c\%a)\%$, and we then assign a unique content for p_2 to match $c\%a$. Finally, by applying all the PC constraints to our CP problem, we can reduce the solution space and generate the solution.

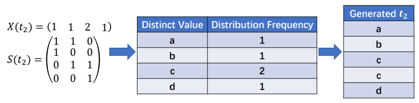
5 Data Population and Parameter Instantiation

In the section, we discuss to populate values for non-key columns and instantiate parameters in match operators based on the solution of CP problem. We first give the solution for CCs only from IN operators with Example 5.

Example 5. Consider a non-key column t_2 in table T, which contains 4 distinct values among 5 rows, i.e., $s,\ x,\ t,\ t,\ l$. Suppose there are three queries containing the IN operator to t_2, which are:

```
select count(*) from T where t2 in (s, x);       -- 2
select count(*) from T where t2 in (s, t);       -- 3
select count(*) from T where t2 in (t, l);       -- 3
```

Following the method introduced in § 3, we give an example distribution vector $X(t_2)$ and calculate a status matrix for three parameters $S(t_2)$ in Fig. 4. Based on $X(t_2)$, we generate four distinct values (a,b,c,d). Next, we populate t_2 according to the distribution in $X(t_2)$, i.e., 'generated t_2' in Fig. 4. Finally, we instantiate parameters of the three IN operators based on each status vector in $S(t_2)$. For example, since $S_1(t_2)$= $(1,1,0,0)^T$, we instantiate the first two values 'a' and 'b' to the first IN operator.



$X(t_2) = (1 \quad 1 \quad 2 \quad 1)$

$S(t_2) = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$

Distinct Value	Distribution Frequency
a	1
b	1
c	2
d	1

Generated t_2
a
b
c
c
d

Fig. 4: Data Population into Non-key Column t_2 of Example 5

However, the LIKE operator complicates the population and instantiation for the **Containment/Exclusion** relationship between PC constraints. Take the prefix match operators for example. We must guarantee all the values matched by the same prefix match operator starting with the same prefix; and we must ensure the instantiation of parameters follows the containment or exclusion PC constraints. Here, we first discuss the population and instantiation for the prefix match operators and then extend the method to the other two match operators.

Consider the parameters p_u and p_v from two prefix match operators, and their result sets M_u and M_v satisfy the containment relationship, i.e. $S_u \odot S_v = S_v$. Supposing we instantiate p_u (resp. p_v) with $str_u\%$ (resp. $str_v\%$), we must guarantee all the values matched by p_u (resp. p_v) start with str_u (resp. str_v) and str_v starts with str_u. The contain relationships among prefixes can be represented by a trie tree structure with mapping results from the parent tree node (p_u) containing the ones from the child node (p_v), donated as ($p_u \rightarrow p_v$).

Algorithm 1 constructs a trie based on the cardinality constraints from prefix match operators in a bottom-up way. Traditionally, a trie tree is built in a top-down way. However, if we insert a new node in a top-down way, it has to compare with all nodes to find its parent; in a bottom-up way, when inserting a new node, we just need to check the node with no parent. Note that the parent node has a shorter prefix requirement than the child node, then the parent node matches more values. It then inspires us to sort all the cardinality constraints in an ascending order of their cardinalities (Line 1). Then, we initialize a directed graph G for accommodating all parameter nodes in CCs (Line 2). We then pop p_i from the head of the queue (Line 4) and try to link it to its child. Based on Theorem 2, we have the containment relationship between any child with its ancestor, i.e., if $p_i \rightarrow p_k$ and $p_k \rightarrow p_j$, we can infer $p_i \rightarrow p_j$. Thus, if the node with in-degree 0 (p_k) satisfies containment relationship with p_i (Line 6), we add an edge from p_i to the existing node p_k (Line 7). At last, we add a root parameter p_0 to G with an all-ones status vector S_0 (Line 8). Then, we link it with all other nodes with in-degree 0 (Line 9-11).

Theorem 2. *For three status vectors S_u, S_v, S_w, if $S_u \odot S_v = S_v$ and $S_v \odot S_w = S_w$, then $S_u \odot S_w = S_w$.*

Proof. $S_u \odot S_w = S_u \odot (S_v \odot S_w) = (S_u \odot S_v) \odot S_w = S_v \odot S_w = S_w$

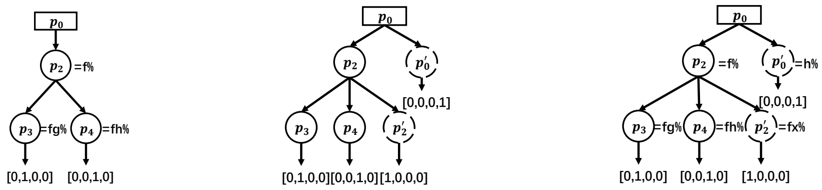
Example 6. Consider the cardinality constraints with prefix match operators in Example 1 and the solution in Fig. 2. In ascending order of their cardinality constraints, we have CC$_4$, CC$_3$ and CC$_2$. First, parameter p_4 is used to initialize the graph G. Then we put parameter p_3 into G. Since $S_3 \odot S_4 = 0$, no edge

Algorithm 1 Construct a trie tree for parameters in LIKE operators

```
Require: CCs of LIKE operators.
 1: Sort CCs into a queue.
 2: Initialize graph $G$ for parameters in CCs.
 3: while queue is not empty do
 4:     Pop queue and put $p_i$ into G.
 5:     for $p_k$ ($\neq p_i$) in graph $G$ do
 6:         if $p_k$.in-degree=0$\wedge S_i \odot S_k = S_k$ then
 7:             Add an edge $p_i \rightarrow p_k$.
 8: Set root $p_0$ with $S_0 = \mathbf{1}^T$ to G.
 9: for $p_k$ in graph $G$ do
10:     if $p_k \neq p_0 \wedge p_k$.in-degree =0 then
11:         Add an edge $p_0 \rightarrow p_k$.
12: return $G$
```

Algorithm 2 Complete a trie tree *w.r.t* the domain space

```
 1: procedure ADD_VIRTUAL_NODE( trie G, parameter node $p_{cur}$)
 2:     for child $p_k$ of $p_{cur}$ in G do
 3:         ADD_VIRTUAL_NODE(G, $p_k$).
 4:     if $p_{cur}$ has children then
 5:         Initialize vector $S_{tmp} \leftarrow S_{cur}$.
 6:         for child $p_k$ of $p_{cur}$ in $G$ do
 7:             $S_{tmp} \leftarrow S_{tmp} \oplus S_k$.
 8:         if $S_{tmp} \neq NULL$ then
 9:             Add a virtual node $p'_{cur}$ with status vector $S_{tmp}$.
10:             Add an edge $p_{cur} \rightarrow p'_{cur}$.
```



(a) Construct a Trie (b) Add Virtual Parameters (c) Instantiate Parameters

Fig. 5: Construct a Prefix Trie & Instantiate Its Parameters.

is added into G. Thirdly, we add parameter p_2 into the graph, which becomes the parent node for both p_3 and p_4, i.e., $p_2 \rightarrow p_3$ and $p_2 \rightarrow p_4$, based on the containment relationship from the solution. Finally, we add a root parameter p_0 and an edge $p_0 \rightarrow p_2$ to complete the trie as in Fig. 5a.

After constructing the prefix trie, we can instantiate these parameters (on the trie) from top to bottom. For a parameter, we instantiate its prefix value by assigning any letter to current node which is concatenated after the prefix value of its parent. Meanwhile, parameter nodes with the same parent should assign distinct letters. Specially, the prefix value of p_0 is empty. For example, in Fig. 5a, we assign f to p_2 and the prefix of p_2 is $f\%$ after combining with the prefix of p_0. Further, we assign letters 'g' and 'h' to p_3 and p_4, which have their prefixes as $p_3 = fg\%$ and $p_4 = fh\%$ after combining with p_1. Then we take the instantiated trie to populate the prefixes of distinct values in the column. Note that only the prefixes of leaf nodes can be taken to generate the data prefix, for only their prefixes can guarantee to be mutually exclusive *w.r.t* the distinct requirement of the generated values. To Example 1, we assign the prefix fg of leaf node p_3 as the prefix of the second value of column t_1 due to $S_3 = (0, 1, 0, 0)$.

However, the leaves in the constructed trie may not cover the entire domain space (i.e., 4 distinct values for column t_1 in Example 1), so we cannot decide all prefixes of the column, i.e., the prefixes for the 1^{st} and 4^{th} values in Fig. 5a. Therefore, we present Algorithm 2 to fill all distinct prefixes for column t_1 by putting virtual parameter nodes. Algorithm 2 takes DFS (Depth-First-Search) to complete the trie tree(Line 2-3). For each parameter node, we first calculate the prefixes uncovered by all children, then put a virtual parameter node with a status vector for its remaining uncovered prefix values. Specifically, we initialize a parameter status vector S_{tmp} to record the prefix values matched by the current parameter p_{cur} (Line 5). Then we execute an exclusive disjunction between S_{tmp} and each status vector of its child S_k (Line 7). At last, if $S_{tmp} \neq Null$, it represents there are some prefix values of current parent p_{cur} can not be matched or covered by its children (Line 8). Then, we put a virtual parameter node p'_{cur} with a status vector S_{tmp} to the trie and add an edge $p_{cur} \to p'_{cur}$ (Line 9-10).

Example 7. Based on Algorithm 2, for the trie in Fig. 5a $w.r.t$ Example 1, we first handle the parameter node p_2 with its status vector as $S_2 = (1, 1, 1, 0)$. For its two children p_3 and p_4, their status vectors are $S_3 = (0, 1, 0, 0)$ and $S_4 = (0, 0, 1, 0)$. After we carry out the exclusive disjunction $S_{tmp} = S_2 \oplus S_3 \oplus S_4 = (1, 0, 0, 0)$, we put a virtual parameter node p'_2 with its status vector S_{tmp} and add an edge from p_2 to p'_2, i.e., $p_2 \to p'_2$ as in Fig. 5b. In the same way, we add the virtual parameter p'_0 as the child of p_0 in Fig. 5b.

Next, we instantiate prefix values for virtual parameters p_2' and p_0' similarly to p_2, p_3 and p_4. Suppose we have $p'_0 = h\%$ and $p'_2 = fx\%$ as in Fig. 5c. Fortunately, either the suffix or the arbitrary-sequence string can be rotated to a prefix string, and then we can take the similar way to construct the trie tree for them. But after we have the corresponding prefixes, they should be rotated reversely to the suffix or arbitrary-sequence match requirement. Suppose the trie tree for parameter p_5 of the suffix match operator is in Example 1. Suffix $\%z$ with status vector (0,1,1,0) is put to the 2^{nd} and 3^{rd} values. Finally the prefixes and suffixes for Example 1 are presented in Fig. 6. Then we concatenate the generated prefixes and suffixes as the final values of each distinct value of the column as shown in Fig. 7. Based on the distribution of the non-key column in X, we can populate the values with the existence frequency into the column.

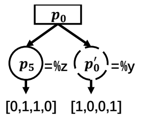
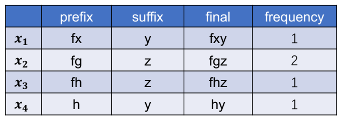


Fig. 6: Example Suffix Trie



	prefix	suffix	final	frequency
x_1	fx	y	fxy	1
x_2	fg	z	fgz	2
x_3	fh	z	fhz	1
x_4	h	y	hy	1

Fig. 7: Prefixes and Suffixes Table for Each Value

6 Discussion

6.1 Optimizations to Solve the CP Problem

We present two optimizations to reduce the complexity of our CP problem.

Allowing errors in the solution. Allowing some errors in the solution can ameliorate the cost of computing a solution. Supposing a maximum error is ϵ $(0 \leq \epsilon \leq 1)$, $X \times S = C$ can be reformulated as $(1-\epsilon) \cdot C \leq X \times S \leq (1+\epsilon) \cdot C$. For the constructed CP problem, there may be multiple feasible solutions, but we need only one solution. So, this optimization increases the number of feasible solutions and facilitates solving the problem, but to assign an appropriate ϵ.

Narrowing the domain of each variable. Consider a simple CP problem seeks variables x_1 and x_2 in the range $[0, 10]$ such that $x_1 + x_2 = 10$, which has a search space of $|x_1| \times |x_2| = 11 \times 11 = 121$. Note that we can scale down the domain of variables and constraints in a consistent way to shrink the search space. For instance, reducing them by a factor of 0.1 transforms the original problem into an equivalent one where $x'_1 \in [0, 1]$ and $x'_2 \in [0, 1]$ and $x'_1 + x'_2 = 1$. The search space becomes $|x'_1| \times |x'_2| = 2 \times 2 = 4$, which is easier to solve. Based on the observation, for the formulated CP problem $X \times S = C$, if we scale down the domain of variables and the constraints with the same ratio $1/\rho (\rho > 1)$, we have an equal CP problem, i.e., $X/\rho \times S = C/\rho$. Supposing we donate $X' = \frac{X}{\rho}$, we then have $X' \times S = C/\rho$. Further, the variables in the CP problem must be of an integer type, we then have the constraint as $X' \times S = \lfloor C/\rho \rceil$. Upon obtaining the result for X', we have $X = \rho \times X'$. In such a way, we introduce at most $\rho/2$ rows of deviation for each CC$_i$, calculated by Equation 4. If we set $\rho = 0.02 * c_{min}$ (c_{min} is the smallest value in C), the most relative error is $\rho/2/c_i = 0.02 * c_{min}/2/c_i \leq 0.02 * c_i/2/c_i = 1\%$.

$$c_i - \rho/2 = (c_i - \rho/2)/\rho \times \rho < \lfloor c_i/\rho \rceil \times \rho \leq (c_i + \rho/2)/\rho \times \rho = c_i + \rho/2 \quad (4)$$

6.2 Implementation based on Touchstone

Touchstone [7] determines the data distribution of non-key columns through multiple iterations. For each iteration, Touchstone first initializes an empty data distribution table and randomly instantiates parameters of cardinality constraints. Next, it fills the data distribution table based on the instantiated parameter values and calculates the overall relative error of cardinality constraints under current data distribution. Finally, it outputs the distribution with the minimized relative error. Touchstone$^+$ first extracts all match operators, and then create an initial data distribution table satisfying their cardinality constraints by Algorithm 1$\sim$2. Meanwhile, Touchstone assumes that the sub-predicates of a logical predicate are independent of each other, and their cardinalities are of the same. For example, $|\sigma_{t_1\ like\ a\% \wedge t_2\ like\ b\%}(R)|/|R|{=}0.04 \Rightarrow |\sigma_{t_1\ like\ a\%}(R)|/|R|{=}|\sigma_{t_2\ like\ b\%}(R)|/|R| = 0.2$. Touchstone$^+$ deals with the query with multiple matching operators based on the same assumptions. For the cardinality constraints from other operators, including other predicates, equi-join and non-equi join [7], Touchstone$^+$ takes the same method as Touchstone.

7 Experimental Studies

We have integrated the implementations of match operators into Touchstone [7] and released Touchstone$^+$ [12]. We launch experiments to answer following questions. **(RQ1)** Can Touchstone$^+$ outperform Touchstone? (§ 7.1) **(RQ2)** How is the scalability of our designs for IN and LIKE operators? (§ 7.2) **(RQ3)** What is the effectiveness of our performance optimizations? (§ 7.3)

Baseline: Touchstone [7] is the state-of-the-art work which is the first work supporting the generation based on the first 16 queries in TPC-H. Touchstone$^+$ is an extension of Touchstone by supporting the important match operators IN and LIKE. Currently, Touchstone is the only work that supports like operator under enormous database generation [7], so we just select it for comparison.

Environment: We run all experiments on PostgreSQL (v.14.5) that supplies an easy interface to obtain cardinality constraints. The machine is CentOS (v.7.9) with 2×Intel(R) Xeon(R) Gold 6240R CPU, 390 GB memory and 2.5 TB disk. We choose the *Choco Solver* [9] (Touchstone compatible) as CP solver.

Workload: To present the performance of Touchstone$^+$, we construct multiple micro benchmarks with more IN and LIKE operators on String-typed columns under two diverse data distributions. Specifically, $uniformData$ is a uniform distribution, having 2×10^5 entries with 150 distinct values; $realData$ is a zipfian distribution, constructed from a TPC-DS [13] having 5×10^4 entries with 1,846 distinct values. We construct three types of micro benchmarks on both the $uniformData$ and $realData$, which are (1) *only* IN *micro benchmark* (100 queries with only IN operator) and each IN selects 1 to 5 distinct values as its search objects, (2) *only* LIKE *micro benchmark* (100 queries with only LIKE operator), and (3) *combined micro benchmark* which covers 50 queries with only IN operator and 50 queries with only LIKE operator. For queries with LIKE operators, we take the same method in TPC-H to form the match patterns. Specifically, the $uniformData$ includes three kinds of substrings: the prefix string, middle string, and suffix string. Subsequently, we utilize the prefix/middle/suffix string to construct the prefix/arbitrary-sequence/suffix match operators. When comparing with Touchstone, we also use the workload in Touchstone, i.e., the first 16 queries from TPC-H.

Metrics: We take two metrics to evaluate the generation performance, which are *average relative error* and *computation time*. Specifically, *average relative error* measures the simulation fidelity (accuracy) for all cardinality constraints, defined by $\frac{\sum_{i=1}^{K} \frac{|c_i - \hat{c_i}|}{c_i} \times 100\%}{K}$ with $c_i/\hat{c}_i$ as the output size of the real/instantiated operator. *Computation time* is the time to solve the CP problem.

Setting: When evaluating the optimization methods mentioned in § 6, the allowable error is ϵ=5% and $\rho = 0.02 * c_{min}$ is selected with c_{min} as the smallest output among all queries. Finally, the total error bound is $\epsilon + \frac{\rho/2}{c_{min}}$=5%+1%=6%.

7.1 Comparing with Touchstone

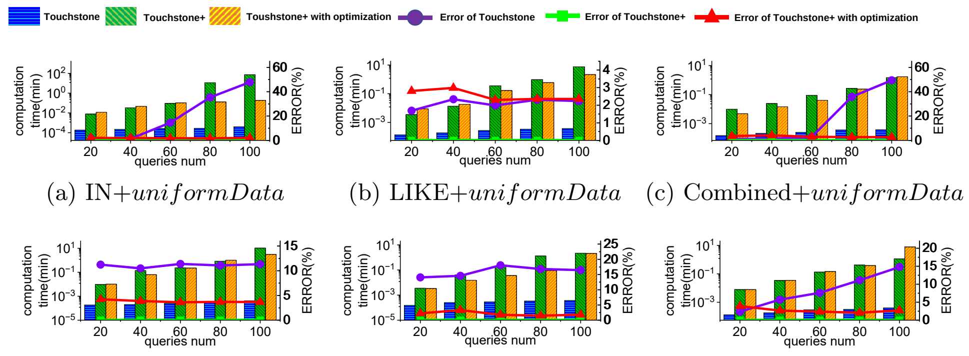
First, we compare generation fidelity between Touchstone$^+$ and Touchstone with the queries of TPC-H. Touchstone can only support the scenario generation

Table 1: Relative Errors of Touchstone$^+$ and Touchstone on TPC-H.

Operator Type		IN		LIKE			
Query		12	16	2	9	13	16
Relative Error	Touchstone	0.039%	0.750%	0.289%	0.854%	0.667%	1.590%
	Touchstone+	0	0	0	0	0	0

with the first 16 queries of TPC-H, i.e., Q1-Q16. Among these queries, Q12 and Q16 have IN operators, while Q2, Q9, Q13, and Q16 have LIKE operators. We generate data and workload by both Touchstone and Touchstone$^+$ and collect the *average relative error* only for IN and LIKE operators to make the comparison clear. As shown in Table 1, Touchstone does have relative errors for these six queries, while Touchstone$^+$ has no errors. Because Touchstone populates data based on a data distribution table, which records the existence probability of each distinct value. It may lead to errors due to computational precision loss. Touchstone exhibits a small relative error ($< 1\%$) for the small cardinality constraints from match operators. In contrast, Touchstone$^+$ records the frequency of each distinct value, which provides accurate results.

Next, we compare the fidelity with all the micro benchmarks constructed on both $uniformData$ and $realData$ in Fig. 8. We increase 20 queries for each round of running. Due to the heuristic rules employed by Touchstone, it will accumulate errors as we increase the queries with IN and LIKE operators, but Touchstone$^+$ still exhibits no simulation errors. However, the *computation time* of Touchstone is extremely short, much lower than that of Touchstone$^+$. It is interesting that Touchstone demonstrates a better simulation fidelity with lower *average relative error* on $realData$. Because the sum of all cardinality constraints on $uniformData$ exceeds 50% of the original table size. Due to the heuristic approach of Touchstone, it leads to larger accumulated errors. On the other hand, the sum of all cardinality constraints on $realData$ only involves 10% of the



(a) IN+$uniformData$ (b) LIKE+$uniformData$ (c) Combined+$uniformData$

(d) IN+$realData$ (e) LIKE+$realData$ (f) Combined+$realData$

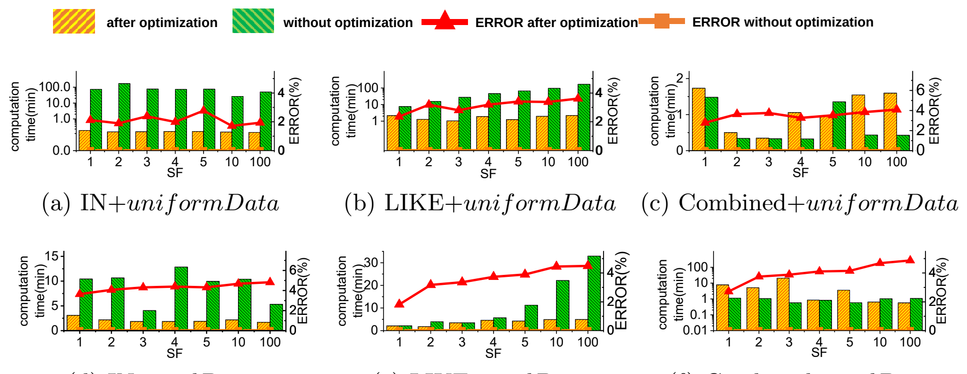
Fig. 8: Performance Comparison on micro benchmarks

original table size, then a smaller error is obtained. Further, we compare Touchstone with Touchstone$^+$ with optimization. It can be seen that both optimization methods significantly reduce computation time in most of the micro-benchmarks on $uniformData$ or $realData$. However, at small workload size or cardinality sizes, there may be negative optimization effects due to the randomness of CP solver. For the *combined micro benchmark*, the optimization may lead to longer *computation time*. Because allowing error in the solution not only increases the number of feasible solutions but also enlarges the search space. For example, consider to search $x_1, x_2 \geq 0$ for $x_1 + x_2 = 1$, the search space is $x_1, x_2 \in \{0, 1\}$. After applying the optimization by replacing $x_1 + x_2 = 1$ with $0 \leq x_1 + x_2 \leq 2$, the search space enlarges, i.e., $x_1, x_2 \in \{0, 1, 2\}$. Then the proportion of feasible solutions in the search space decreases, the solving time increases. Note that all the relative errors are less than the error bound 6%.

7.2 Scalability of Touchstone$^+$

We change the data scales (SFs) to expose the scalability of Touchstone$^+$ with *average relative error* and *computation time*. For each micro benchmark, we first change its SF from 1 to 5 (low SF), then set SF to 10 and 100 (high SF).

Fig.9 shows the performance of Touchstone$^+$ on $uniformData$ under different SFs. By the *only* IN *micro benchmark* in Fig.9a, we observe that the *computation time* consistently exceeds one hour without optimization. However, after optimization, the *computation time* is reduced to less than one minute and the *average relative error* remains within the error bound 6%. Fig.9b shows the *computation time* for the *only* LIKE *micro benchmark*, which rises as SF increases. Specifically, it only takes two minutes at SF=1, while at SF=100, it requires over three hours. Because the CCs from IN operators give the number of parameters, i.e., the count of '1's in vector S_i. It triggers internal optimization mechanisms of the *Choco Solver*; In contrast, the CCs from LIKE operators do not give,



(a) IN+$uniformData$ (b) LIKE+$uniformData$ (c) Combined+$uniformData$

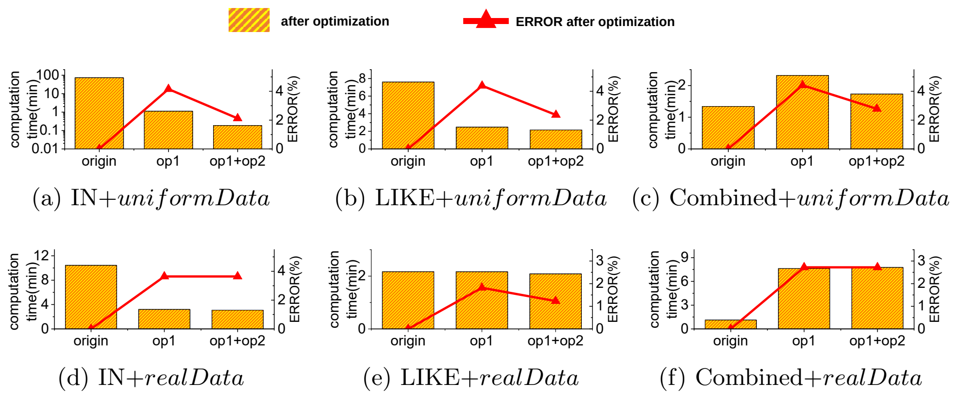
(d) IN+$realData$ (e) LIKE+$realData$ (f) Combined+$realData$

Fig. 9: The Impact of Data Scale on Performance

causing its *computation time* increasing with the growth of SF. However, after applying optimizations, the computation time is reduced to approximately one minute. In Fig.9c, we observe that the *computation time* for Touchstone$^+$ on the *combined micro benchmark* is consistently much lower than the *computation time* on the *only* IN *micro benchmark* and *only* LIKE *micro benchmark*. Because *computation time* is exponentially correlated with the number of CCs from IN/LIKE operators (refer to Fig.8). In the *combined micro benchmark*, the number of CCs from IN/LIKE operators is half less than the first two micro benchmarks. Further, Fig.9 shows performance of Touchstone$^+$ on three micro benchmarks constructed on *realData* under different SFs. Since *realData* has more distinct values but a smaller dataset compared to $uniformData$, the cardinality of each CC is smaller. Consequently, the *computation time* on the three micro benchmarks is relatively lower. Moreover, when running on *combined micro benchmark*, the optimization methods may lead to a negative optimization effect with the same reason analyzed in § 7.1.

7.3 Optimization

To illustrate the impact of the two optimizations on performance, we run Touchstone$^+$ in three scenarios which are without optimization (origin), with " Allowing error in the solution" (OP1) and OP1 together with "Narrowing the domain of each variable" (OP1+OP2). As shown in Fig. 10, it is effective to apply OP1 in reducing computation time for both the *only* IN *micro benchmark* and *only* LIKE *micro benchmark* on $uniformData$. In the *combined micro benchmark*, this method may lead to longer *computation time* with the same reason analyzed in §. 7.1. Meanwhile, we find that OP2 has a minimal impact on micro benchmarks on *realData*. This is because the output sizes from queries in these three micro benchmarks are typically small and the effectiveness of ρ is weak as shown in § 6, then this optimization is ineffective.



(a) IN+$uniformData$ (b) LIKE+$uniformData$ (c) Combined+$uniformData$

(d) IN+*realData* (e) LIKE+*realData* (f) Combined+*realData*

Fig. 10: The Impact of Two Optimizations on Performance with *realData*

8 Related work

Currently, extensive research is ongoing to optimize the performance of IN and LIKE operators [5,6,19]. Optimization efforts for IN operators primarily focus on enhancing data structures like Hash Tables and indexes. Optimizations of the LIKE operator emphasize finding more efficient pattern-match algorithms. However, the experimental datasets employed in these studies either rely on traditional benchmarks [19] or artificially constructed datasets [5], which lack the representative characteristics of real-world application scenarios.

Then query-aware database generators are introduced [2,8], which propose to simulate an application based on some simple statistics from the in-production environment. Thus, they compensate for the shortcomings of traditional benchmarks [14,13,18].But they can only generate a single database for each query and support limited SQL operators. Recent work including DCGen [1], Touchstone [7,15], and HYDRA [10], has improved their capabilities to simulate one application scenario/database for multiple queries on multiple tables. Specifically, Touchstone generates the application without using the actual parameter values in query plans, while HYDRA and DCGen take the constraints with the real values. However, all this work still falls short of processing IN and LIKE operators due to their reliance on heuristic rules for parameter instantiation or data population. Given the diversity of operator types, query-aware database generators are still in the early stage of practical applicability.

9 Conclusion

In this paper, we propose Touchstone$^+$ (providing source codes [12]) to solve the query aware database generator problems for match operators. By modelling the cardinality constraints from match operators into a CP problem, we can successfully obtain the solution by existing CP solver. And we provide serveral optimizations to the CP problem to reduce the computation time of CP solver. Our experimental results have verified the effectiveness of Touchstone$^+$.

Acknowledgement. The work is supported by NSFC (62072179, 62307014), Shanghai Sailing Program (22YF1411300) and Ant Group Research Fund.

References

1. Arasu, et al.: Data generation using declarative constraints. In: SIGMOD (2011)
2. Binnig, et al.: Qagen: generating query-aware test databases. In: SIGMOD (2007)
3. Blizard, et al.: Multiset theory. In: Notre Dame Journal of formal logic. vol. 30, pp. 36–66 (1989)
4. Garey, et al.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co. (1979)
5. Hu, et al.: A fast algorithm for multi-string matching based on automata optimization. In: FCC. vol. 2, pp. V2–379 (2010)

6. Khan, et al.: A Transformation For Optimizing String-Matching Algorithms For Long Patterns. In: The Computer Journal. vol. 59, pp. 1749–1759 (2016)
7. Li, Y., et al.: Touchstone: Generating enormous query-aware test databases. In: ATC 2018. (2018)
8. Lo, et al.: Mybenchmark: generating databases for query workloads. In: VLDBJ. vol. 23, pp. 895–913 (2014)
9. Prud'homme, et al.: Choco-solver. Journal of Open Source Software **7**(78), 4708 (2022)
10. Sanghi, A., et al.: Scalable and dynamic regeneration of big data volumes. In: EDBT 2018. (2018)
11. Schrijvers, et al.: Monadic constraint programming. Journal of Functional Programming **19**(6), 663–697 (2009)
12. Touchstone$^+$: Codes and technical report. https://github.com/DBHammer/Touchstone-plus
13. TPC-DS: TPC-DS benchmark. http://www.tpc.org/tpcds/ (1999)
14. TPC-H: TPC-H benchmark. http://www.tpc.org/tpch/ (1999)
15. Wang, Q., et al.: A scalable query-aware enormous database generator for database evaluation. IEEE Transactions on Knowledge and Data Engineering **35**(5), 4395–4410 (2022)
16. Yang, Z., et al.: Oceanbase: A 707 million tpmc distributed relational database system. Proc. VLDB Endow. **15**(12), 3385–3397 (2022)
17. Yang, Z., et al.: Oceanbase paetica: A hybrid shared-nothing/shared-everything database for supporting single machine and distributed cluster. Proc. VLDB Endow. **16**(12), 3728–3740 (2023)
18. Zhang, C., et al.: Benchmarking on intensive transaction processing. Frontiers of Computer Science **14**, 1–18 (2020)
19. Zheng, et al.: Saha: a string adaptive hash table for analytical databases. In: Applied Sciences. vol. 10, p. 1915. MDPI (2020)