Mirage: Generating Enormous Databases for Complex Workloads

Qingshuai Wang
ECNU, China
qswang@stu.ecnu.edu.cn

Hao Li
ECNU, China
haoli@stu.ecnu.edu.cn

Zirui Hu
ECNU, China
zrhu@stu.ecnu.edu.cn

Rong Zhang
ECNU, China
rzhang@dase.ecnu.edu.cn

Chengcheng Yang
ECNU, China
ccyang@dase.ecnu.edu.cn

Dian Qiao
Huawei Technologies Co., Ltd.
qiaodian@huawei.com

Xuan Zhou
ECNU, China
xzhou@dase.ecnu.edu.cn

Aoying Zhou
ECNU, China
ayzhou@dase.ecnu.edu.cn

ABSTRACT

In production application simulation is essential for application-driven database benchmarking or performance debugging. A rich body of query-aware database generators (QAG) are proposed for this purpose. The complex data dependencies hidden behind queries make previous work suffer from critical deficiencies in supporting complex operators and binding the simulation errors, which greatly limit their usages. In order to fill the gap between the existing QAGs and the emerging demands, we design a new data generator ***Mirage*** with the good property of regenerating applications with complex operators by a bound error. Specifically, *Mirage* leverages *Query Rewriting* and *Set Transforming Rules* to decouple dependencies and simplify the generation problem; it presents a uniform representation to various types of joins and formulates key population as a *Constraint Problem*, which can be solved by an existing *CP Solver*. We have run extensive experiments and the results show that *Mirage* has the widest support to operators and bound low errors to all simulations. More importantly, it is the first work that accomplishes simulation of the TPC-H application by all its 22 queries with near-*zero* error bound.

PVLDB Reference Format:
Qingshuai Wang, Hao Li, Zirui Hu, Rong Zhang, Chengcheng Yang, Dian Qiao, Xuan Zhou, and Aoying Zhou. Mirage: Generating Enormous Databases for Complex Workloads. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBHammer/Mirage.

1 INTRODUCTION

When customers plan to migrate their application to a new database, it is of great importance to benchmark the application's performance and service stability on the new database [2, 13, 14, 22]. Although classic benchmarks [17, 24, 27] have been widely used for performance evaluation, they could not well represent the characteristics of real-world applications. However, it is usually impossible to use the real-world application data in a test environment due to the concerns of privacy and liability [22].

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

In order to perform an application-driven benchmarking, the Query-Aware Database Generator (QAG) [2, 3, 7, 12–14, 22] is proposed to generate a synthetic data processing environment to simulate the real-world application with specific data and workload characteristics. It requires that the synthetic query can mimic the execution cost in real-world applications, which is closely related to the cardinality constraint [14] (i.e., the output size) of each operator on the query tree. In this way, the QAG is expected to well reconstruct the application scenario. Moreover, the QAG is also a prerequisite for performance debugging which occurs frequently during database upgrading. In this case, the database developers need to reproduce a performance regression which is highly related to the real-world application scenario. However, the in production data and workloads are usually not allowed to be taken out. Fortunately, they can obtain the in production execution metrics, which include the query plans and the output size of each operator. Then, by organizing the metrics as cardinality constraints, database developers can use QAG to regenerate the data processing environment such that the performance regression problem can be solved.

However, the QAG has been proven to be an *NP-complete* task [2]. This is because the cardinality constraints of various query operators in applications impose perplexing joint data distribution requirements among different columns. To reduce the computational complexity of the problem, existing solutions usually choose to either support less operators or tolerate more simulation errors as in Table 1, among which *Mirage* is our method. We observe that existing methods lack the support of: **1) Complex predicate connected by arithmetic and logical operators.** It introduces joint data distribution requirements among non-key columns [2]; **2) Outer/anti/semi join operators and projection on foreign keys in enormous database.** It introduces joint data distribution requirements between the primary and foreign key columns. Existing work for this issue usually relies seriously on memory consumption [3, 13, 14] and are not scalable. These joint data distributions have been challenging the design of QAG. As a result, none of the existing work [2, 12–14, 22] can even support to mimic a complete TPC-H scenario with its 22 queries as shown in Table 1.

In this paper, we propose a new generator *Mirage*, which not only can successfully solve the complex joint data distribution requirements in a light way, but also give a strict guarantee on simulation errors. Specifically, it utilizes different techniques to deal with join distributions between different columns.

Between key and non-key columns. It is caused by the uncertain execution order between select and join operators in a query plan, which leads to a bidirectional joint data distribution dependencies from selection to join or in reverse. We propose to take the

Table 1: Comparison of Current Query Aware Data Generators with *Mirage* in Operator Supportability

Related Work	Selection			Join				Projection	Theorical	Terabyte-	TPC-H
	Predicate	Arithmetic	Logical	Equi	Anti	Outer	Semi	Foreign Key	Relative Error	Generation	Query Support
QAGen	Arbitrary	F	Arbitrary	T	F	F	F	T	Zero	F	13
MyBenchmark	Arbitrary	F	Arbitrary	T	F	F	F	T	No Guarantee	F	13
DCGen	$>, \geq, <, \leq, =$	F	DNF	T	F	F	F	F	Low	T	8
Hydra	$>, \geq, <, \leq, =$	F	DNF	T	F	F	F	F	Zero	T	6
Touchstone	Arbitrary	T	Simple	T	F	F	F	F	No Guarantee	T	16
Mirage	**Arbitrary**	**T**	**Arbitrary**	**T**	**T**	**T**	**T**	**T**	**Zero**	**T**	**22**

common relational algebra-based *query rewriting* method to push down select operators without breaking the cardinality constraints in query plans. Therefore, we reduce the bidirectional distribution dependencies to a unidirectional one, i.e., from key column to non-key column.

Between non-key columns. It is introduced by logical or arithmetic predicates. An important observation of the QAG problem is that we are required to guarantee the output size of each operator instead of that of each sub-predicate. It inspires us to simplify the multi-column distribution dependencies by eliminating some sub-predicates but guarantee the output size, i.e., to reduce complexity. Specifically, for a logical predicate, the *set transforming rules* can help to trim sub-predicates; for an arithmetic predicate, we can evade the problem of the joint data distribution requirement by transforming it into a parameter searching problem in the sampled space of its involved columns.

Between the primary and foreign key columns. It is caused by PK-FK join operators. We define a uniform representation (join cardinality&distinct constraints) to cardinality constraints from various join operators. To avoid the common conflicts of the population solutions from different joins and reduce computational complexity, a multi-dimensional join visibility-oriented row representation and organization method is proposed for each relation, which partitions rows by their visibility to the involved joins. Based on the uniform join constraint representations and partition-based data organizations, the problem of plotting the joint distributions of key columns is converted into a classic *Constraint Programming* (CP) problem [20] which can be simply solved by existing *CP Solver* [19].

Based on above solutions, we implement *Mirage*, a novel and generic QAG for an application regeneration. The main contributions are summarized as followings.

- *Mirage* provides the most powerful and strong support to complex query operators or predicates. It is the first work which accomplishes regeneration of the complete TPC-H application.
- *Mirage* guarantees simulation with a theoretical *zero* error bound even having more supported operators.
- *Mirage* is able to generate a database of any data size in a linear way with controllable memory consumption.
- We launch extensive experiments on three classic application scenarios and compare with the state-of-the art work. *Mirage* totally conquers the related work by the lowest average errors as well as full support to all these scenarios.

2 PRELIMINARIES

In this section, we first introduce the two inputs of query aware database generation. Then, we describe the cardinality constraints of various query operators. Finally, we give the formal definition of the query aware database generation problem (QAG).

2.1 Database and Annotated Query Template

A database D consists of multiple tables (i.e., relations). Each table R_i in D contains a single primary key PK_i column, a certain number of non-key columns $A_{i_1}, \cdots, A_{i_q}$, and zero or more foreign keys columns $FK[R_{i_1}], \cdots, FK[R_{i_m}]$. Here, $FK[R_{i_1}]$ indicates that the foreign key of R_i refers to the primary key of R_{i_1}.

Cardinality Constraints of Table and Non-Key Column. Given a table R and a non-key column A in R, their cardinality constraints (denoted as $|R|$ and $|R|_A$) refer to the unique data values contained in R and A. Specifically, $|R|$ can be viewed as the row count of R and $|R|_A$ can be viewed as the domain size of A.

Annotated Query Template (AQT). A query template can be created by parameterizing the input value of each predicate in a query plan [12]. An annotated query template AQT [2] labels the cardinality constraint on the output of each operator op.

Example 2.1. Fig. 1 presents four annotated query templates. Consider the selection operator $\sigma_{s_1<p_1}$ in $AQT\ Q_1$. The input value of its predicate is parameterized as p_1, and its cardinality constraint is labeled as 2. This indicates that the output size of $\sigma_{s_1<p_1}$ is 2.

2.2 Cardinality Constraint of Query Operator

In this section, we first introduce the query operator view that represents the output of a query operator, and then give the definition of cardinality constraints for three kinds of common query operators, which are selection, join, and projection.

Query Operator View (V). A query operator view represents the output rows of a query operator, and can be classified into the leaf view and internal view. A leaf view covers all rows of a specific table, and an internal view covers the execution results of a query operator which operates on its input views. Specifically, the selection and projection operators have only one input view, while the join operator has two input views.

Example 2.2. Consider the $AQT\ Q_1$ in Fig. 1, it can be represented by query operator views from the bottom up as follows.

$$\begin{aligned} leaf\ views: V_1 &= S & V_2 &= T \\ internal\ views: V_3 &= \sigma_{s_1<p_1}(V_1) & V_4 &= \sigma_{t_1>p_2}(V_2) \\ V_5 &= \bowtie^{=}_{t_{fk}}(V_3, V_4) & V_6 &= \Pi_{t_{fk}}(V_5) \end{aligned}$$

Selection View ($\sigma_P(V)$). A selection view is generated by performing a selection operator on an input view V with a logical predicate P. For simplicity, we mainly discuss the case in which the predicate P follows the conjunctive normal form (CNF). Note that a predicate with any other form can be transformed to CNF [21]. That is,

$$P = clause_i \wedge ... \wedge clause_n \qquad s.t.\ clause_i = literal_i \vee ... \vee literal_n.$$

Here, $literal_i$ can be a unary or an arithmetic predicate. More specifically, a unary predicate follows the form of $A_i \bullet p_k$ with

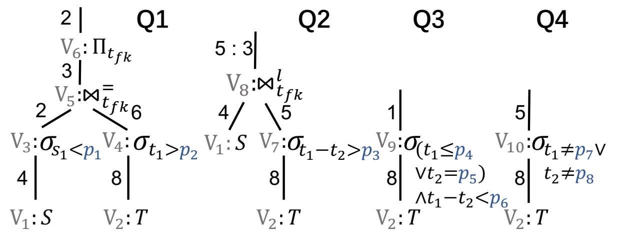

Q1
Q2
Q3
Q4
2
$V_6: \Pi_{t_{fk}}$
3
$V_5: \bowtie^{=}_{t_{fk}}$
2
6
$V_3: \sigma_{s_1<p_1}$
$V_4: \sigma_{t_1>p_2}$
4
8
$V_1: S$
$V_2: T$
5 : 3
$V_8: \bowtie^{l}_{t_{fk}}$
4
5
$V_1: S$
$V_7: \sigma_{t_1-t_2>p_3}$
8
$V_2: T$
1
$V_9: \sigma_{(t_1 \le p_4 \vee t_2 = p_5) \wedge t_1 - t_2 < p_6}$
8
$V_2: T$
5
$V_{10}: \sigma_{t_1 \neq p_7 \vee t_2 \neq p_8}$
8
$V_2: T$

Figure 1: Annotated Query Templates on Tables S and T

• $\in \{=, \neq, <, >, \le, \ge, (not)\ in, (not)\ like\}$, e.g., $s_1 < p_1$ in V_3. An arithmetic predicate might operate on multiple non-key columns $A_i, ..., A_k$ through an arithmetic function $g()$, which usually follows the form of $g(A_i, ..., A_k) \circ p_k$, $\circ \in \{<, >, \le, \ge\}$, e.g., t_1-t_2> p_3 in V_7.

Selection Cardinality Constraint (*SCC*). A selection cardinality constraint $|\sigma_P(V)| = m$ requires that there should exactly exist m rows in the input view V which satisfy the predicate P. Moreover, the *SCC* can be further classfied into three categories according to the predicate P, which are unary cardinality constraint (*UCC*), arithmetic cardinality constraint (*ACC*) and logical cardinality constraint (*LCC*). Specifically, the predicate in *UCC*/*ACC* is a simple unary/arithmetic predicate, while the predicate in *LCC* is a combination of multiple unary or arithmetic predicates by logical operators.

Join View $\bowtie^{type}$ (V_l, V_r). A join view is generated by performing a join operator on two input views V_l and V_r with a specified join type. In general, the join type includes *equi join* (=), *left/right/full outer join* ($l/r/f$), *left/right semi join*(ls/rs), and *left/right anti join*(la/ra). Following previous studies [2, 3, 7, 12–14, 22], we also focus on the *PK*-*FK* join in this paper.

Join Cardinality/Distinct Constraint (*JCC*/*JDC*). To represent the output size of different join types in a general way, we abstract the join constraints by join cardinality constraint and join distinct constraint. A join cardinality constraint (*JCC*) requires that there should exactly exist n_{jcc} matched pairs of rows in the two input views V_l and V_r. For ease of presentation, hereinafter we assume that V_l contains the primary key (pk) of the referenced table, and V_r contains the foreign key (fk) of the referencing table. Then, if the pk value of a row in V_l equals the fk value of a row in V_r, the two rows can be considered as matched. For the join distinct constraint (*JDC*), it requires that there should exactly exist n_{jdc} distinct pk/fk values in all matched pairs of rows from V_l and V_r. *JCC* and *JDC* are used together to determine the output size of any type of join (see Table 2). For example, suppose a table S left outer joins a table T, and the cardinality constraints are $n_{jcc} : n_{jdc}$. We can infer that there exist $|S| - n_{jdc}$ rows that are not matched with the rows in T and its output size is $|S| - n_{jdc} + n_{jcc}$. Note that, for some joins in Table 2, either *JCC* or *JDC* is sufficient to determine the output size. For example, the *equi join* only uses *JCC* to determine its output size for it only outputs rows matched in V_l and V_r.

Projection View $\Pi_{PK/FK/C}(V)$. A projection view is generated by performing a projection operation on the input view V. It retains the unique rows in the result set through duplicate elimination.

Projection Cardinality Constraint (*PCC*). A projection cardinality constraint $|\Pi_{PK/FK/C}(V)| = m$ requires that the size of the projection result should exactly be m. As each primary key uniquely identifies a row in a table, the input and output cardinalities of a *PK* column are identical. Thus, the *PK* column is not interesting in the projection cardinality constraint. In addition, the performance of projection has been observed to be only influenced by its input size if its output cardinality is not very huge [12, 15]. Since the projections on non-key columns in an OLAP database usually involve dimension columns [11] that are filled with category data, the output size is relatively small and then has a marginal effect on the performance [12]. For example, the cardinalities of projected columns *l_shipmode* and *o_orderpriority* in TPC-H are 7 and 5, respectively. However, the cardinality of an *FK* column is closely related to the cardinality of its referenced *PK* column, which may be huge and has a great impact on projection performance. For example, the cardinality of the *FK* column *l_orderkey* in TPC-H is equal to the cardinality of the *PK* column *o_orderkey*, which is 1.5 billion when the *scale factor* is 1000. Based on the above observations, we only focus *PCC* on *FK* columns to simplify the complexity of query aware database generation.

Table 2: Output Size of Joins on S and T with T Referencing S.

Join Type	Equi	Outer			Semi		Anti	
		Left	Right	Full	Left	Right	Left	Right
JCC	T	T	F	F	F	T	F	T
JDC	F	T	F	T	T	F	T	F
Size	n_{jcc}	$\lvert S\rvert - n_{jdc} + n_{jcc}$	$\lvert T\rvert$	$\lvert S\rvert - n_{jdc} + \lvert T\rvert$	n_{jdc}	n_{jcc}	$\lvert S\rvert - n_{jdc}$	$\lvert T\rvert - n_{jcc}$

Note that, the *PCC* on an *FK* column declares the unique number of foreign keys in the output, and it can be converted to a *JDC* on its child join view. For example, the *PCC* of $|\Pi_{t_{fk}}(V_6)| = 2$ in Fig. 1 can be converted to a *JDC* of V_5 with $n_{jdc} = 2$. If a projection does not have a descendant join view, we manually add a virtual right semi join view as its child, as shown in Fig. 2. Specifically, we set its left input view as the table referenced by the projected *FK* column, and set its right input view as the original input of the projection view. In this way, the output of the virtual join view is exactly consistent with the original input of the projection view. Then, we can convert the *PCC* to a *JDC* on its child virtual join view. Note, the virtual join views are only used to convert all the *PCC*s to *JDC*s such that the type of constraints can be reduced. They will finally be removed from query plans after our query aware database generation process is finished.

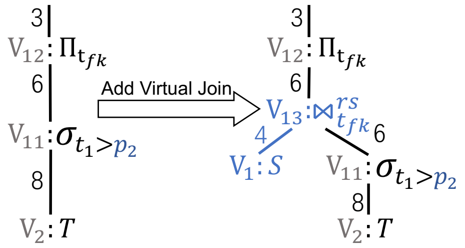


Figure 2: Add a Virtual Join View for the Projection View

2.3 Problem Definition

We are now ready to formulate the problem of query aware database generation as follows.

Definition 2.3. **Query Aware Database Generation**. Given the cardinality constraints of tables and non-key columns in a database D, the annotated query templates (*AQT*) with labeled cardinality constraints which are abstracted from traces of a workload W, the query aware database generation aims to (1) generate a synthetic database D', and (2) instantiate parameters in the *AQT*s to produce a new workload W', such that all the cardinality constraints are guaranteed if running W' on D'.

Example 2.4. Suppose the database D in production contains two tables S and T. S consists of a primary key s_{pk} and a non-key column s_1, T consists of a primary key t_{pk}, a foreign key t_{fk} which references s_{pk}, and two non-key columns t_1 and t_2. The cardinality constraints of D are $|S| = 4$, $|T| = 8$, $|S|_{s_1} = 4$, $|T|_{t_1} = 5$, $|T|_{t_2} = 4$, and the cardinality constraints of *AQT*s are listed in Fig. 1. Then, the query aware database generation aims to construct a synthetic database D' and instantiate all parameters in *AQT*s as in Fig. 3.

$p_1 = 30$	$p_2 = 2$
$p_3 = 0$	$p_4 = 1$
$p_5 = 0$	$p_6 = 5$
$p_7 = 4$	$p_8 = 2$

(a) Instantiated Parameters

s_{pk}	s_1
1	10
2	20
3	30
4	40

t_{pk}	t_1	t_2	t_{fk}
1	3	2	1
2	3	2	1
3	3	2	3
4	5	3	3
5	5	1	4
6	4	4	2
7	2	3	4
8	1	4	4

(b) D': Populated Tables S and T

Figure 3: Example of Query Aware Database Generation

3 DESIGN OVERVIEW

Mirage aims to simulate the application scenarios without exposing sensitive information inside the database. Fig. 4 shows an overview of our *Mirage* framework. Specifically, it uses a *workload parser* to collect two kinds of data from the production database as its inputs. The first one is the cardinality constraints of tables and columns, which can be directly obtained from the metadata table and statistics in the database. The second one is the annotated query templates (*AQT*) with labeled cardinality constraints, which are extracted from the query execution traces.

The existence of various cardinality constraints has imposed requirements of joint data distributions among different columns. In general, the joint data distributions can be classified into three types according to the involved columns, which are joint distributions between the primary key and foreign key, between key and non-key columns, and between different non-key columns. These requirements make it an NP-complete problem to generate the query aware database [2]. To address this issue, we propose to first decouple the joint distributions (i.e., dependencies) between key and non-key columns, and then generate data and instantiate parameters for these two kinds of columns by non-key generator and key generator separately as in Fig. 4.

Decouple Dependencies Between Key and Non-key Columns. The requirement of the joint distributions between key and non-key columns is from the uncertain execution order between selection on non-key columns and join on key columns. If the selection executes after the join, it indicates that the filter result relies on the join result, i.e., the distribution of non-key columns relies on that of key columns. Otherwise, the distribution of key columns relies on that of non-key columns. Generally, the current query optimizer usually tries to use its *rule based optimizer* module to push down select operators directly to each table, so as to reduce the volume of data transferred along the query tree. However, if we observe that there exists an *AQT* in which the selection executes after the join, then we can rewrite the query tree based on relational algebraic transformations so as to push down the selection without breaking the cardinality constraints. With this mechanism, we avoid the bidirectional dependencies between key and non-key columns. It enables us to generate all non-key columns firstly, and then to generate key columns based on the distribution of non-key columns. Note, the rewritten query tree is only used in our database generation phase, the user would finally use the original query plan and all the cardinality constraints in that plan are guaranteed when performing tests on the simulated database.

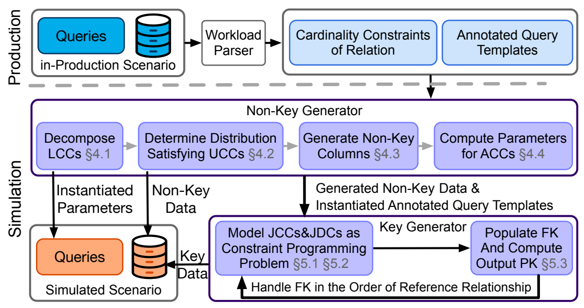
Example 3.1. Consider the query plan $\sigma_{P_S \vee P_T}(S \bowtie T)$ in which the selection executes after the join. Suppose it has two cardinality constraints $|S \bowtie T| = n_1$ and $|\sigma_{P_S \vee P_T}(S \bowtie T)| = n_2$, where P_S (resp. P_T) denotes a predicate P on the table S (resp. T). As the selection contains the logical *OR* operation, it could not be pushed down directly to tables S and T. Thus, we need to rewrite the query tree. Specifically, from $\neg(P_S \vee P_T) = \neg P_S \wedge \neg P_T$, we derive an equivalence transformation as following.

$$S \bowtie T - \sigma_{P_S \vee P_T}(S \bowtie T) = \sigma_{\neg(P_S \vee P_T)}(S \bowtie T) = \sigma_{\neg P_S}(S) \bowtie \sigma_{\neg P_T}(T)$$

Based on the transformation, we convert $|\sigma_{P_S \vee P_T}(S \bowtie T)| = n_2$ into three equivalent constraints, which are $|\sigma_{\neg P_S}(S)| = n_3$, $|\sigma_{\neg P_T}(T)| = n_4$ and $|\sigma_{\neg P_S}(S) \bowtie \sigma_{\neg P_T}(T)| = n_1 - n_2$. Note, n_3 and n_4 can be obtained by executing $\sigma_{\neg P}(S)$ and $\sigma_{\neg P}(T)$ in the original database through the workload parser in Fig 4. Finally, after generating the database satisfying the three constraints and $|S \bowtie T| = n_1$, both of the two constraints in the original query plan can be satisfied.

Example 3.2. Consider the query plan $\sigma_P(S \bowtie T)$ with two cardinality constraints, i.e., $|S \bowtie T| = n_1$ and $|\sigma_{P_1}(S \bowtie T)| = n_2$, which has its select operator executed after its join operator. If P is a predicate that can be pushed down to relation T, we replace $|\sigma_P(S \bowtie T)| = n_2$ with two equivalent cardinality constraints, i.e., $|\sigma_P(T)| = n_3$ and $|S \bowtie \sigma_P(T)| = n_2$. Note that *Mirage* obtains n_3 by executing $\sigma_P(T)$ in the original database.

Generate Non-Key Data. *Mirage* utilizes its *non-key generator* to populate non-key columns and instantiate the selection related parameters in *AQT*s, such that all the selection cardinality constraints (*SCC*s) can be satisfied. However, the logical cardinality constraints (*LCC*) and arithmetic cardinality constraints (*ACC*) in *SCC*s usually involve multiple non-key columns, which would bring the requirements of joint distributions between them. As a result, it makes non-key data generation still *NP-complete* [2]. The main reason is the domain size of a joint distribution is the cumulative product of all involved columns' domain sizes, which leads to expensive domain space searching cost when instantiating parameters in *AQT*s. To address this issue, we propose to take the following four steps to eliminate requirements of joint distributions between non-key columns and thus reduce the computational complexity. Firstly, we propose to apply set transforming rules to trim sub-predicates of *LCC*s without affecting the output size of each selection view. With this mechanism, we can decouple *LCC*s into individual unary cardinality constraints (*UCC*s) and *ACC*s (§4.1). Considering that *ACC*s would also lead to the requirements of joint distributions between non-key columns (e.g., $t_1 - t_2 < p_6$ in Q_3 in Fig. 1), we propose to first populate all the non-key columns according to the associated *UCC*s and then compute the parameter of each *ACC*



Production
Queries
in-Production Scenario
Workload Parser
Cardinality Constraints of Relation
Annotated Query Templates
Simulation
Non-Key Generator
Decompose LCCs §4.1
Determine Distribution Satisfying UCCs §4.2
Generate Non-Key Columns §4.3
Compute Parameters for ACCs §4.4
Instantiated Parameters
Non-Key Data
Generated Non-Key Data & Instantiated Annotated Query Templates
Queries
Simulated Scenario
Key Data
Key Generator
Model JCCs&JDCs as Constraint Programming Problem §5.1 §5.2
Populate FK And Compute Output PK §5.3
Handle FK in the Order of Reference Relationship

Figure 4: Design Overview of Mirage

based on the generated non-key column data independently. Secondly, when populating a non-key column, we need to first derive its data distributions according to the *UCC*s on that column. For this purpose, we abstract it as a classic bin packing problem. Specifically, the column's domain space is considered as a container, and each *UCC* which specifies a specific constraint on the column's data distribution is considered as an item with a specific size. Then, solving the bin packing problem is equivalent to deriving a valid data distribution for the column such that all the associated *UCC*s are satisfied (§4.2). Next, following the derived data distribution, we instantiate parameters in *UCC*s and generate data for each non-key column (§4.3). Lastly, to instantiate the parameter in each *ACC*, we first use its arithmetic function to compute the result view based on the involved non-key columns, which have been generated above. Then, we search the one dimensional result space and find a valid parameter value such that the *ACC* is satisfied (§4.4).

Generate Key Data. The *key generator* of *Mirage* is used to populate key columns which satisfy all the join cardinality constraints. As the primary keys usually serve as an identifier of each row and there exists no distribution requirement on the *PK* column, we propose to follow previous studies [12] and generate them by an auto-incrementing integer generator. Then, our main aim is to populate *FK* columns according to the join constraints. For this purpose, we first formalize three *FK* populating rules for each *JCC*/*JDC* on two joined tables. Specifically, for any join view, the rules specify how to select primary keys in its left input view V_l and how to use them to populate foreign keys in its right input view V_r (§5.1). However, we observe that there might exist some conflicts if we populate foreign keys according to the *JCC*/*JDC* of each join view independently (e.g., use different primary keys to populate a foreign key). To address this issue, one naive method is to first enumerate all possible ways of populating foreign keys based on the *JCC*/*JDC* of each join view. Then, it selects one populating way from each join view and check whether there exists a conflict between them. However, it is highly inefficient. Thus, we propose to partition the table according to the overlaps between all the join input views V_l and V_r. Specifically, for the table whose primary keys are referenced by the foreign keys in another table, if its two rows are partitioned into the same partition, they must appear or disappear in any given left input view V_l at the same time. Similarly, the rows in the referencing table are partitioned according to their appearances in the right input views. In this way, each input view V_l/V_r can be represented by several partitions. Next, we integrate the partitions into the three *FK* populating rules for each join view described above, and then the problem can be modeled as a classic

$|V_4| = |\sigma_{t_1>p_2}(T)|$=6 $\quad$ $|V_9| = |\sigma_{(t_1\le p_4\vee t_2=p_5)\wedge t_1-t_2<p_6}(T)|$=1

$|V_7| = |\sigma_{t_1-t_2>p_3}(T)|$=5 $\quad$ $|V_{10}| = |\sigma_{t_1\ne p_7\vee t_2\ne p_8}(T)|$=5

(a) Selection Cardinality Constraints of Table T

UCCs on t_1: $|V_4| = |\sigma_{t_1>p_2}(T)| = 6 \quad |V_9^{1(1)}| = |\sigma_{t_1\le p_4}(T)| = 1$

$|\overline{V_{10}^{1(1)}}| = |\sigma_{t_1=p_7}(T)| = 3$

UCC on t_2: $|\overline{V_{10}^{1(2)}}| = |\sigma_{t_2=p_8}(T)| = 3$

ACC on t_1, t_2: $|V_7| = |\sigma_{t_1-t_2>p_3}(T)| = 5$

(b) Selection Cardinality Constraints After View Eliminations

$f_{t_1}(1)$=12.5%	$f_{t_1}(2)$=12.5%	$F_{t_1}(2,4)$=62.5%	$f_{t_1}(4)$=37.5%
$f_{t_1}(5)$=12.5%	$f_{t_2}(1)$=12.5%	$f_{t_2}(2)$=37.5%	$F_{t_2}(2,4)$=50%

(c) Distribution For Table *T*

Figure 5: Selection Cardinality Constraints on Table T

Constraint Programming (*CP*) problem [20], which can be solved by existing *CP Solver* [19] (§5.2). Finally, to support the case of multi-table join, we propose to build a directed graphs according to the references between tables and perform a topological sorting on the graph. Then, we populate the *FK* column of each table in topological order with the help of population method designed for two joined tables (§5.3).

4 NON-KEY GENERATOR

In this section, we first introduce how to decouple *LCC*s into *UCC*s and *ACC*s (§4.1). Then, we discuss how to use the bin packing method to deal with *UCC*s, including instantiating parameters in each *UCC* and generating non-key column data (§4.2-§4.3). Finally, we present how to instantiate parameters in each *ACC* (§4.4).

4.1 Decouple Logical Dependencies

As the existence of *LCC*s would make it *NP-complete* to construct non-key columns, we first propose to decouple *LCC*s into *UCC*s and *ACC*s. Then, the joint data dependencies between non-key columns caused by *LCC*s can be decoupled. Recall that the logical predicate is assumed to be a *CNF* formula (see §2.2), then we can convert a selection view V containing a logical predicate $\wedge_{k=1}^{n} clause_k$ into the intersection of sub-selection views $\cap_{k=1}^{n} V^k$, where V^k is constructed by the sub-predicate $clause_k$:

$$V = \sigma_{\wedge_{k=1}^{n} clause_k}(R) = \cap_{k=1}^{n}\sigma_{clause_k}(R) = \cap_{k=1}^{n} V^k$$

Moreover, as each $clause_k$ in a *CNF* formula is a disjunction of $literal$s, then each sub-selection view V^k can be decomposed as:

$$V^k = \sigma_{\vee_{i=1}^{m} literal_i}(R) = \cup_{i=1}^{m}\sigma_{literal_i}(R) = \cup_{i=1}^{m} V^{k(i)}$$

Here, we use $V^{k(i)}$ to represent a subsub-selection view constructed by $literal_i$ in $clause_k$.

Example 4.1. In the selection view V_9 in Fig. 5a, it has a logical predicate P=$(t_1 \le p_4 \vee t_2{=}p_5) \wedge t_1{-}t_2 < p_6$. Then, we can decompose it into one arithmetic view V_9^2 and two unary views $V_9^{1(1)}$ and $V_9^{1(2)}$.

$$V_9 = \underline{\sigma_{t_1\le p_4\vee t_2=p_5}(T)} \cap \underline{\sigma_{t_1-t_2<p_6}(T)} \qquad = V_9^1 \cap V_9^2$$
$$V_9^1 = \underline{\sigma_{t_1\le p_4}(T)} \cup \underline{\sigma_{t_2=p_5}(T)} \qquad = V_9^{1(1)} \cup V_9^{1(2)}$$

Table 3: Assigned Boundary Values for View Elimination

Comparator		$>, \geq$	$<, \leq$	*in, like,* $=$	*not in, not like,* $\neq$
Assigned	U	$-\infty$	$+\infty$	/	*NULL*
Value	$\emptyset$	$+\infty$	$-\infty$	*NULL*	/

Since our goal is to guarantee the output size of a whole selection view instead of guaranteeing the concrete size of each sub-selection or subsub-selection view, it provides us a good opportunity to eliminate some sub-selection and subsub-selection views and thus reduce the complexity of generating non-key columns. Specifically, we leverage the following two rules in the area of set theory to guide the elimination procedure:

$$rule_1 : V_i \cap V_j = V_j \quad if \quad V_i \leftarrow U$$
$$rule_2 : V_i \cup V_j = V_j \quad if \quad V_i \leftarrow \emptyset.$$

Rule$_1$ and *rule*$_2$ indicate that we can eliminate any sub-selection or subsub-selection view without affecting the output size if it meets certain criteria, such as the universal set U or empty set $\emptyset$. For example, the union result view V_9^1 is only determined by $V_9^{1(2)}$ if $V_9^{1(1)}$ is set as $\emptyset$. To this end, we try to assign boundary values to the parameters of each sub/subsub-selection view's predicate. Table 3 lists the boundary values of typical predicates. For example, consider $V_9^{1(1)}$ which contains a comparator "$\geq$", it would be $\emptyset$ if we set p_4 as $-\infty$ since the column t_1 has no value less than $-\infty$.

More specifically, for any selection view V containing a logical predicate, our elimination procedure consists of two steps. 1) we try to set each sub-selection view V^k constructed by $clause_k$ as the universal set U. If all the V^k can be set as U, we only keep one sub-selection view and eliminate other sub-selection views; otherwise, we eliminate all the sub-selection views that can be set as U. 2) for each remaining V^k, we try to set each of its subsub-selection view $V^{k(i)}$ constructed by $literal_i$ in $clause_k$ as $\emptyset$. Then, we eliminate the subsub-selection views in a similar way to that of the first step.

Example 4.2. Consider the selection view V_9 in Fig. 5a with the cardinality constraint $|V_9| = 1$. Firstly, we observe that both V_9^1 and V_9^2 can be U if we set p_4=$+\infty$ and p_6=$+\infty$. Then, we keep one sub-selection view and eliminate the other one. Suppose V_9^2 is eliminated by setting p_6=$+\infty$. Secondly, we try to set subsub-selection views of V_9^1 as $\emptyset$. Specifically, both $V_9^{1(1)}$ and $V_9^{1(2)}$ can be $\emptyset$ if we set $p_4 = -\infty$ and $p_5 = null$. Then, we eliminate one subsub-selection view. Suppose $V_9^{1(2)}$ is eliminated, then we simplify the cardinality constraint from $|V_9| = 1$ to $|V_9^{1(1)}| = 1$.

Note, if all the predicates in a *clause* only contain comparators of *not in*, *not like* and $\neq$, then none of the subsub-selection view can be set as $\emptyset$ (see Table 3). To address this issue, we make use of the *De Morgan's law* [5] shown in *rule*$_3$ to convert it to an equivalent constraint which is easier to deal with. Here, $|U_V|$ denotes the universal set size of the selection view V. In our case, $|U_V|$ equals the number of rows in the table that is operated by V.

$$rule_3 : |V^{k(1)} \cup ... \cup V^{k(m)}| = n \Leftrightarrow |\overline{V^{k(1)}} \cap ... \cap \overline{V^{k(m)}}| = |U_V|\text{-}n$$

Example 4.3. Consider the selection view V_{10} operating on table T in Fig. 5a. Its universal set size equals the size of table T, which is $|T| = 8$. Then, we can convert the cardinality constraint of $|V_{10}| = |V_{10}^{1(1)} \cup V_{10}^{1(2)}| = 5$ as following.

$$|\overline{V_{10}^{1(1)}} \cap \overline{V_{10}^{1(2)}}| = |\sigma_{t_1=p_7}(T) \cap \sigma_{t_2=p_8}(T)| = |T| - |V_{10}| = 8 - 5 = 3$$

Theorem 4.4 shows that our elimination procedure can reduce a selection view V to its subsub-selection view $V^{k(i)}$ (the first case) or the conjunction of unary views $\cap_{j=1}^{\omega} V_e^j$ (the second case) whose comparators are *in*, *like* or $=$. Note that the second case requires that some values must coexist in the same row. For example, the conversion of V_{10} requires that there must exactly exist three rows whose $t_1 = p_7$ and $t_2 = p_8$. Nevertheless, we can first derive valid cardinality constraints for each of these values in the individual column. Then, after all the non-key columns are generated, we add a post-processing step to bound these values into same rows. Fig. 5b shows the elimination result of selection views V_4, V_7, V_9 and V_{10}.

Theorem 4.4. *Given a selection view V, our elimination procedure can reduce it to one of the following two cases: $V^{k(i)}$ or $\cap_{j=1}^{\omega} V_e^j$, where $V^{k(i)}$ is a subsub-selection view of V, V_e^j is a unary view whose comparator is in, like or $=$, and ω is the number of unary views.*

Proof. In our elimination procedure, the first step is to set each sub-selection view as U. Given a sub-selection view V^k, from table 3, we can see that it cannot be set as U if each literal in $clause_k$ only contains comparators of *in*, *like* and $=$. Suppose there exists q sub-selection views which cannot be set as U.

If $q > 0$, the first step would reduce V to $\cap_{j=1}^{q} V^{i_j}$, where all the literals in the i_j^{th} clause (i.e., $clause_{i_j}$) only contain comparators of *in*, *like* and $=$. As a result, in the second step, each subsub-selection view of V^{i_j} can be as $\emptyset$. Next, *Mirage* only keeps one subsub-selection view for each V^{i_j}, and finally reduces V to $\cap_{j=1}^{q} V_e^j$.

If $q = 0$, the first step would reduce V to a single sub-selection view V^k. Suppose there exist s subsub-selection views of V^k which cannot be set as $\emptyset$. If $s = 0$, the second step of *Mirage* would reduce it to $V^{k(i)}$. Otherwise, it reduces V^k to $\cup_{j=1}^{s} V^{k(i_j)}$, where the i_j^{th} literal (i.e., $literal_{i_j}$) in $clause_k$ only contains the comparator of *not in*, *not like* and $\neq$. From *rule*$_3$, we can see that $\cup_{j=1}^{s} V^{k(i_j)}$ can be converted to $\cap_{j=1}^{s} \overline{V^{k(i_j)}}$, where $\overline{V^{k(i_j)}}$ is a unary view whose comparator is *in*, *like* or $=$.

The theorem is proved. □

4.2 Solve Unary Selection Operators

With the decoupling procedure described above, all the *LCC*s can be converted to *UCC*s and *ACC*s. Then, we only need to deal with these two kinds of cardinality constraints. As each *UCC* involves a single column, we propose to first use *UCC*s to derive the data distribution of each non-key column independently, and then instantiate the parameters of *ACC*s based on generated data.

Specifically, for each non-key column A, we use the cumulative distribution function CDF_A to represent its data distribution. Moreover, as different columns usually have various data types and domain spaces, we first normalize the original domain space of each non-key column to its cardinality space of integral type such that all the *UCC*s can be resolved in a general way. For example, consider a non-key column A whose cardinality constraint is $|R|_A$ (i.e., the domain size of A) . We assume that all the parameters in *UCCs* are integers in $(0, |R|_A]$. In this way, deriving *CDF* in the original domain space is converted to deriving *CDF* in the cardinality space.

Definition 4.5. **Cumulative Distribution Function of Non-Key Column.** Given a non-key column A, the cumulative distribution function $F_A(p)$ is defined as the probability of a row whose attribute A takes a value less than or equal to p. Moreover, $F_A(p_i, p_j)$ is defined as the probability of a row whose attribute A lies in the interval $(p_i, p_j]$, where $p_i < p_j$, and $f_A(p)$ is defined as the probability of a row whose attribute A takes a value equaling to p.

To further simplify the process of deriving CDF_A, we also propose to convert all the comparators in *UCC*s (see Table 3) into two kinds of comparators, which are = and ≤. Specifically, the *UCC* with a comparator *in* can be converted into the union of multiple *UCC*s whose comparators are =, and the *UCC* with a comparator *like* can be converted to a comparator *in* by querying the number of distinct matching values in the original database. In addition, based on commutativity property from *De Morgan's Law* [5], the comparators >, ≥ and ≠ can be easily converted to ≤ and =. For example, $|\sigma_{A>p}(R)| = k$ is equal to $|\sigma_{A\le p}(R)| = |R|-k$.

Then, we can use the two kinds of *UCC*s to derive $F_A(p)$ and $f_A(p)$ respectively. If $|\sigma_{A\le p}(R)|=k$, we have $F_A(p)=k/|R|$, and if $|\sigma_{A=p}(R)| = k$, we have $f_A(p)=k/|R|$. For example, consider the three *UCC*s on column t_1 in Fig. 5b. From $|\sigma_{t_1>p_2}(T)|=6$, $|\sigma_{t_1\le p_4}(T)|=1$, $|\sigma_{t_1=p_7}(T)|=3$ and $|T|=8$, we can infer that $F_{t_1}(p_2)=(8-6)/8 = 25\%$, $F_{t_1}(p_4)=1/8=12.5\%$ and $f_{t_1}(p_7)=3/8=37.5\%$. Next, given a non-key column A and its associated *CDF*s $F_A(p)$ and $f_A(p)$, we take three steps to instantiate all the parameters of *UCC*s on column A. Note, we will discuss data generation for column A in next section.

(1) Determine the Partial Order for Each Parameter p in $F_A(p)$. As $F_A(p)$ monotonically increases with p, we can infer that p_j must be greater than p_i if $F_A(p_i) < F_A(p_j)$. Then, for each parameter p and its corresponding *CDF* $F_A(p)$, we propose to sort the pair $(F_A(p), p)$ in ascending order of $F_A(p)$. Suppose we have n parameters $p_1, \cdots, p_n$ in cloumns A's *UCC*s whose comparators can be converted to ≤. After ordering, the partial order of these n parameters is represented by $p_{h_1}, \cdots, p_{h_n}$, and each p_{h_i} corresponds to one of the parameters. Then we can divide the cardinality space $(0, |R|_A]$ into $n+1$ ranges which satisfy

$$F_A(0, p_{h_1}) + \sum_{j=1}^{n-1} F_A(p_{h_j}, p_{h_{j+1}}) + F_A(p_{h_n}, |R|_A) = 1$$

(2) Determine the Partial Order for Each Parameter p in $f_A(p)$. Our second step is to put each parameter p in $f_A(p)$ into one of the $n+1$ ranges properly. Specifically, suppose we would put k parameters $p_{e_1}, \cdots, p_{e_k}$ into the range $(p_i, p_j]$, we should guarantee that the sum of data existence probabilities of $p_{e_1}, \cdots, p_{e_k}$ does not exceed the data existence probability of the range $(p_i, p_j]$, i.e., $\sum_{j=1}^{k} f_A(p_{e_j}) \le F_A(p_i, p_j)$. Then, the problem can be regarded as the conventional bin packing problem, which has been proven to be NP-hard [6]. To address this issue, we propose a greedy method. For each parameter p in $f_A(p)$, we always find the range $(p_i, p_j]$ which can accommodate p validly and has the smallest value of $F_A(p_i, p_j) - \sum_{j=1}^{k} f_A(p_{e_j})$. Here, $p_{e_1}, \cdots, p_{e_k}$ are parameters already put into the range $(p_i, p_j]$. After putting p into $(p_i, p_j]$, we repeat the above process until all parameters are in the $n+1$ ranges.

Note that, our greedy method might fail to find valid ranges for a parameter p if it cannot be accommodated by any range $(p_i, p_j]$ validly. Nevertheless, since our aim is to guarantee the cardinality size of each *UCC*, we can first look up whether there exists a parameter p' which has been put in a range and its $f_A(p')$ equals $f_A(p)$. If so, we put p into the same range and specifies $p = p'$. Otherwise, we convert $f_A(p) = x$ into $f_A(p_1) = x_1, \cdots, f_A(p_q) = x_q$, where $x_1 + \cdots + x_q = x$. This can be implemented by converting the *UCC* from $|\sigma_{A=p}(R)| = x \cdot |R|$ to $|\sigma_{A\ in\ (p_1,\cdots,p_q)}(R)| = x \cdot |R|$. Obviously, a requirement of smaller cardinality granularity in the *UCC* would have a higher probability to be satisfied.

In addition, after putting k parameters $p_{e_1}, \cdots, p_{e_k}$ into a range $(p_i, p_j]$, we should specify the data existence probability of each newly split range. As we only need to guarantee $F_A(p_{e_i}, p_{e_{i+1}}) \ge f_A(p_{e_{i+1}})$, there are various ways to allocate the rest probability $F_A(p_i, p_j) - \sum_{j=1}^{k} f_A(p_{e_j})$ to each range. Specifically, we propose to allocate it randomly to the newly split ranges.

(3) Instantiate Parameters According to Partial Orders. Suppose that there are n extra ranges divided by $F_A(p)$ and m extra ranges divided by $f_A(p)$, respectively. With the two steps above, we have divided the cardinality space $(0, |R|_A]$ into $n+m+1$ ranges. As the data existence probability of each range $(p_i, p_j]$ (i.e., $F_A(p_i, p_j)$) is greater than zero, we first assign one unique value to each range. For the rest $|R|_A - m - n - 1$ unique values, we can assign them to each range arbitrarily. In our case, we assign these values to each range in a uniform way. Note that, when assigning unique values to a range $(p_i, p_j]$, the data existence probability $F_A(p_i, p_j)$ should not be violated. For example, consider a range $(p_i, p_j]$, if its parameter p_j has a constraint $f_A(p_j)$, then the number of assigned unique values to that range is at most $|R| \cdot (F_A(p_i, p_j) - f(p_j))$. Then, we instantiate parameters according to the number of assigned unique values to each range. Specifically, each parameter p_i is instantiated as the number of unique values which does not locate after p_i.

Example 4.6. Consider the column t_1 of table T in Example 2.4. As its domain size is assumed to be 5, then we have $F_{t_1}(0) = 0\%$ and $F_{t_1}(5) = 100\%$. Fig. 5b shows that we have three *UCC*s regarding t_1 after decoupling all the *LCC*s on table T. Moreover, as discussed before, we can derive that $F_{t_1}(p_2) = 25\%$, $F_{t_1}(p_4) = 12.5\%$ and $f_{t_1}(p_7) = 37.5\%$ based on the three *UCC*s and $|T| = 8$.

The first step is to sort the pair $(F_{t_1}(p), p)$ for each parameter p in $F_{t_1}(p)$. As $F_{t_1}(p_4) < F_{t_1}(p_2)$, then we have the partial order of parameters as $p_4 < p_2$ and the cardinality space can be divided into three ranges $(0, p_4]$, $(p_4, p_2]$ and $(p_2, 5]$, where $F_{t_1}(0, p_4) = 12.5\%$, $F_{t_1}(p_4, p_2) = 12.5\%$ and $F_{t_1}(p_2, 5) = 75\%$. Next, our second step is to find a proper range for each parameter p in $f_{t_1}(p)$. In our case, there exists only one $f_{t_1}(p_7)$ and only the third range has a data existence probability that is greater than $f_{t_1}(p_7)$, then we put p_7 in the range $(p_2, 5]$. This further splits the range into $(p_2, p_7]$ and $(p_7, 5]$. Note, we only need to guarantee that $F_{t_1}(p_2, p_7) \ge f_{t_1}(p_7)$, and the rest probability $F_{t_1}(p_2, 5) - f_{t_1}(p_7) = 37.5\%$ can be allocated arbitrarily to $(p_2, p_7]$ and $(p_7, 5]$. For example, if we allocate 25% and 12.5% to the two ranges, then we have $F_{t_1}(p_2, p_7) = 62.5\%$ and $F_{t_1}(p_7, 5) = 12.5\%$. Finally, our last step is to instantiate parameters. As the first two steps have divided the cardinality space into 4 ranges, we first assign one unique value to each range. Then, only one unique value left in the domain space. Moreover, from $|T| * F(0, p_4) =$

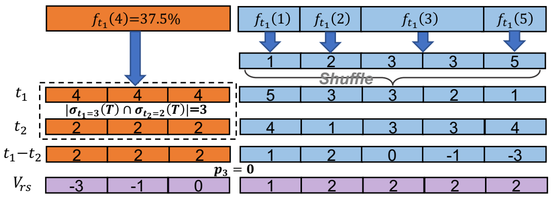


Figure 6: Generate Values for the Table T

$|T| * F(p_4, p_2) = |T| * F(p_7, 5) = 1$, we observe that only the data existence probability $F_{t_1}(p_2, p_7)$ is not violated if we assign an extra value to that range. Thus, we can infer that only the range $(p_2, p_7]$ contains more than one (i.e., 2) unique value. Thus, we have $p_4 = 1$, $p_2 = 2$ and $p_7 = 4$. Fig. 5c shows the *CDF*s for non-key columns t_1 and t_2 after processing t_2 in a similar way.

Time Complexity Analysis. Let N be the total number of parameters in *UCC*s on the non-key column A. In the first step, the time complexity of sorting each parameter p in $F_A(p)$ is $O(NlogN)$. In the second step, for each parameter p in $f_A(p)$, we use a binary search to find the smallest range which can accommodate it. Thus, the time complexity of the second step is $O(NlogN)$. In the third step, the time complexity of assigning unique values and instantiate parameters is $O(N)$. Thus, the total time complexity is $O(NlogN)$.

4.3 Generate Data Based on CDF

After determining the data existence probability $F_A(p_i, p_j)$ and the number of unique values for each range $(p_i, p_j]$, we are now ready to leverage them to guide the data generation procedure. In this section, we first discuss how to generate data for each non-key column, and then introduce how to arrange values from different non-key columns in each relational table.

Generate Data for Each Non-Key Column. Given a non-key column A, we take two steps to generate its data. Firstly, we deal with each parameter p in $f_A(p)$. Suppose p is instantiated as a_p (see §4.2), then we can infer that the existence probability of the data a_p is $f_A(p)$. To this end, we generate $|R| \cdot f_A(p)$ data items with the value of a_p. Secondly, for the other data items, we first use $F_A(p_i, p_j)$ to derive the volume of data items in the range $(p_i, p_j]$. Then, we generate data according to the number of unique values in that range. Suppose there exist μ unique values in $(p_i, p_j]$. Since we only concern about the number of data items regarding $F_A(p_i, p_j)$, we can generate totally $|R| \cdot F_A(p_i, p_j)$ data items with μ unique values based on any given distribution, e.g., uniform distribution.

Example 4.7. Consider the range $(2, 4]$ with $F_{t_1}(2, 4) = 62.5\%$ in Fig. 5c. As $f_{t_1}(4) = 37.5\%$, we first generate $|T| \cdot f_{t_1}(4) = 3$ data items with value of 4. Moreover, as there exist two unique values 3 and 4 in the range $(2, 4]$, we can also derive that $f_{t_1}(3) = 25\%$. Thus, we generate $|T| \cdot f_{t_1}(3) = 2$ data items with value of 3. Fig. 6 shows the data generation result for columns t_1 and t_2 based on their *CDF*s in Fig. 5c. Note, we generate the data of the range $(2, 4]$ in t_2 with a uniform distribution.

Arrange Values from Different Non-Key Columns. After generating data for non-key columns, our next step is to arrange them in each table. Recall that our decoupling procedure might reduce a selection view V into $\cap_{j=1}^{\omega} V_e^j$, where V_e^j is a unary view whose comparator is *in*, *like* or =. Thus, the $\cap$ operator and = comparator in the cardinality constraint $| \cap_{j=1}^{\omega} V_e^j| = n$ require that the column values associated with each view V_e^j must be bound to n same rows. To this end, we propose to first populate rows with values meeting the cardinality constraints of selection views that follow the format of $\cap_{j=1}^{\omega} V_e^j$. Then we populate rows generated to satisfy the other selection constraints to the non-key columns. Further, the primary key for each row can be generated alongside populating its non-key columns. As primary keys usually have no semantic meaning, we propose to generate them by an auto-incremental integer generator. For example, Fig. 7 shows the non-key column t_1 and t_2 generated by our *non-key generator* according to their distribution in Fig. 5c.

Example 4.8. Fig. 6 shows the populating result of relation T, i.e., t_1 and t_2. Note that the conversion of V_{10}, i.e, $|\overline{V_{10}^{1(1)}} \cap \overline{V_{10}^{1(2)}}| = |\sigma_{t_1=p_7}(T) \cap \sigma_{t_2=p_8}(T)|$, imposes three bounded rows of $p_7 = 4$ (on t_1) and $p_8 = 2$ (on t_2). Therefore, Mirage first puts three rows $(4, 2)$ to the head of t_1 and t_2, and then shuffles and populates the remaining data into t_1 and t_2 respectively.

4.4 Solver of Arithmetic Selection Operator

The arithmetic cardinality constraint (*ACC*) can be satisfied by taking advantage of the column data populated above. Specifically, given an *ACC* $|\sigma_{g(A_i...A_j)\bullet p}(R)| = n$, where $g()$ is an arithmetic function, $A_i, ..., A_k$ are the non-key columns operated by $g()$, and $\bullet$ is a comparator. We first calculate the result view $g(A_i...A_j)(R)$ based on the generated non-key columns. Then, our aim is to find a parameter p such that there exactly exist n items in the result view which satisfies the predicate $\bullet p$. Take the comparator $\leq$ as an example, we can set p as the n^{th} largest value in the result view.

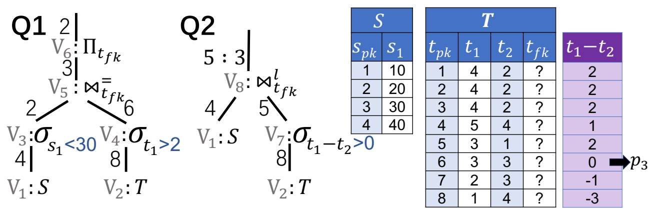
Example 4.9. Consider the *ACC* $|V_7| = |\sigma_{t_1-t_2>p_3}(T)| = 5$ in Fig. 5b. We first calculate the result view $V_{rs} = g(t_1, t_2) = t_1 - t_2$ based on the populated data in t_1 and t_2. As the comparator is >, we compute the result as shown in Fig. 6 and then set p_3 as the $3rd$ largest value in the result (i.e., $p_3 = 0$).

However, as the real-world table referred by our generator is usually rather large, this makes it expensive to calculate the result view. Moreover, the large table might also lead to large volumes of data in the result view that could not be fully accommodated in memory. As a result, it further increases the complexity of finding parameters. To address this issue, we propose to sample a small batch of rows from the table to approximate its data distributions. Then, we perform the above parameter instantiation process based on the sampled rows. In addition, based on Hoeffding's Inequality [8], the number of sampled rows can be calculated as $\frac{ln2-ln(1-\alpha)}{2\delta^2}$ according to the given error bound δ and confidence level α.

After solving all the *ACC*s, we have finished generating non-key column data and instantiating all their parameters. Moreover, the parameters in $Q_1 \sim Q_4$ can be instantiated as

$$p_1 = 30 \quad p_2 = 2 \quad p_3 = 0 \quad p_4 = 1 \quad p_5 = 0 \quad p_6 = 5 \quad p_7 = 4 \quad p_8 = 2.$$

Next, for the queries containing join operators (e.g., Q_1 and Q_2), our *non-key generator* can directly compute the inputs of their join views based on the populated data and instantiated parameters. Then, it transfers them to the *key generator*.



S	
s_{pk}	s_1
1	10
2	20
3	30
4	40

T				
t_{pk}	t_1	t_2	t_{fk}	t_1-t_2
1	4	2	?	2
2	4	2	?	2
3	4	2	?	2
4	5	4	?	1
5	3	1	?	2
6	3	3	?	0
7	2	3	?	-1
8	1	4	?	-3

Figure 7: Non-key Column Data and Instantiated Queries

5 KEY GENERATOR

In this section, we discuss our method of populating foreign keys for all tables. We first introduce how to deal with a single join cardinality and distinct constraints (*JCC&JDC*) on two tables in §5.1. Then, we discuss how to fuse multiple *JCCs&JDCs* on two tables without conflicts in §5.2. Finally, we extend the solution to joins on multiple tables in §5.3.

5.1 Single Join Constraint on Two Tables

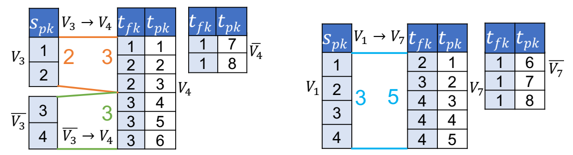
As defined in §2.2, a join view $\bowtie^{type}(V_l, V_r)$ has two child views V_l and V_r as its inputs. For a single join view, its *JCC* requires that there exist n_{jcc} matched pair of rows in two input views (i.e. the primary key of a row in V_l equals the foreign key of another row in V_r). Its *JDC* requires that there exactly exist n_{jdc} distinct pk/fk values in all matched pair rows. Moreover, as shown in Table 2, the cardinality constraints from all join types can be represented by n_{jcc} and n_{jdc}. Next, we discuss how to populate foreign keys on the joined table according to the join constraint on two tables.

From the definition of *JCC*, we can infer that we should select some primary keys from V_l and then populate them to the foreign key column of n_{jcc} rows in V_r (denoted as $PF_{V_l \to V_r} = n_{jcc}$). In the meanwhile, we can also infer that the rest of $|V_r| - n_{jcc}$ foreign keys in V_r should not match any primary key in V_l. To this end, we need to populate these foreign keys with primary keys which do not exist in V_l (denoted as $PF_{\overline{V_l} \to V_r} = |V_r| - n_{jcc}$). In addition, *JDC* imposes an additional constraint which requires that there exist n_{jdc} distinct foreign keys in all matched pair of rows from V_l and V_r. Thus, we must exactly select n_{jdc} primary keys in V_l when populating foreign keys in V_r (denoted as $PF^d_{V_l \to V_r} = n_{jdc}$). To summarize, as shown in Equation 1, we can derive three foreign key populating rules from the constraint of *JCC*/*JDC*.

$$PF_{V_l \to V_r} = n_{jcc} \quad PF_{\overline{V_l} \to V_r} = |V_r| - n_{jcc} \quad PF^d_{V_l \to V_r} = n_{jdc} \tag{1}$$

Recall that we always firstly push down selection operators (see §3) and then both V_l and V_r can be immediately constructed after generating non-key column data and instantiating the corresponding parameters. Next, we can populate the foreign keys in V_r by applying the populating rules in Equation 1. For the foreign keys which are not in V_r, they can be populated by any primary keys in the referenced table. This is because they would not join with the primary keys in V_l and do not affect the output of $\bowtie^{type}(V_l, V_r)$.

Example 5.1. Consider the join view $V_5 = \bowtie^{=}_{t_{fk}}(V_3, V_4)$ in Fig. 7. From its join constraints n_{jcc}=3, we know there exist 3 matched pairs of rows from V_3 and V_4. Additionally, from $|V_6|$=$|\Pi_{t_{fk}}|$=2, we can infer that 2 distinct values in all the foreign keys of V_4 could match the primary keys of V_3. This indicates that n_{jdc}=2. Then, the three foreign key populating rules for V_4 are listed as follows.



(a) Populate t_{fk} for V_5 (b) Populate t_{fk} for V_8

Figure 8: Populate t_{fk} Based on Populating Rules

$$PF_{V_3 \to V_4} = n_{jcc}=3 \quad PF_{\overline{V_3} \to V_4} = |V_4| - n_{jcc}=3 \quad PF^d_{V_3 \to V_4} = n_{jdc}=2$$

Fig. 8a shows an example of populating foreign keys in V_4. For ease of presentation, we only list the primary and foreign keys. According to the rules $PF_{V_3 \to V_4} = 3$ and $PF^d_{V_3 \to V_4} = 2$, we should select 2 primary keys in V_3 and use them to populate 3 foreign keys in V_4. As V_3 only has primary keys 1 and 2, then we use both of them for population. As shown in Fig. 8a, we can populate the first three foreign keys in V_4 as 1, 2 and 2. According to the rule $PF_{\overline{V_3} \to V_4} = 3$, we should populate the rest foreign keys in V_4 by any primary key in $\overline{V_3}$. For example, we can use primary key 3 in $\overline{V_3}$ for population. Finally, since $\overline{V_4}$ does not affect the output of $\bowtie^{=}_{t_{fk}}(V_3, V_4)$, the foreign keys in it can be populated with any primary key in table S, e.g., the primary key 1.

5.2 Multiple Join Constraints on Two Tables

In this section, we discuss how to fuse multiple join constraints on two tables without conflicts. Suppose there exist m join views on two tables S and T, where S is a referenced table and T is a referencing table. If we populate the foreign keys according to the constraints of each join view individually, we might use different primary keys to populate the foreign key at the same row of table T. For example, Fig. 8a and Fig. 8b show two population results of table T according to the join constraints of V_5 and V_8 in Fig. 7. For the row with primary key of 1, its foreign key is populated as 1 and 2 if we apply the population method described in §5.1.

To address this issue, one naive method is to firstly enumerate all possible foreign key population results, and then find a feasible result which does not lead to any conflict. However, it's computationally prohibitive to enumerate population results for two joined tables. Specifically, for each foreign key in the referencing table T, it can be populated by any one of the primary keys in the referenced table S. Thus, there exist $|S|$ population results for each foreign key. Considering that the referencing table has $|T|$ foreign keys, then the total number of population results is $|S|^{|T|}$.

Note that, there exist some overlaps of primary keys between the input left child views. For example, consider the left child views V_1 and V_3 in Fig. 8, both of them contain the primary keys 1 and 2. Similarly, there also exist some overlaps of foreign keys between the input right child views. For example, both V_4 and V_7 contain the foreign keys in rows 1~5 of table T. This motivates us to design a table partitioning based method to reduce the computational complexity. The basic idea is that we can first partition each table into disjoint partitions according to the overlaps of primary/foreign keys between input views. Then, for each partition in the referenced table, we derive its populating rules regarding partitions in the referencing table. By combining all the populating rules, we can formalize it as the *Constraint Programming* (*CP*) problem. Finally, we take advantage of the existing solution of *CP* problem to get the result of fusing multiple join constraints.

(1) Partition Tables. As a row can be uniquely identified by its primary key, for ease of presentation, we use the primary key to represent a row. In addition, we denote the left and right child view of the k^{th} join view as V_l^k and V_r^k, respectively. If a primary key in the referenced table S is contained in V_l^k, it has the chance to (be input to) participate in the join, then it is an input join candidate for the k^{th} join view. Specifically, for each row in S, we use a status value 0/1 to indicate whether it is a join candidate of a given join view. Suppose there exist m join views in total, then each row in S is associated with a m dimensional status vector. Similarly, each row in the referencing table T also has a m dimensional status vector, which indicates whether it is an input join candidate in each of the m right child views, i.e., whether V_r^k contains the row's foreign key.

Example 5.2. Figure 9 shows the 2 dimensional status vectors of tables S and T regarding two join views $V_5 = \bowtie^{=}_{t_{fk}} (V_3, V_4)$ and $V_8 = \bowtie^{l}_{t_{fk}} (V_1, V_7)$ in Fig. 7. Specifically, the status vectors of table S are constructed based on left child views V_1 and V_3, while the status vectors of table T are constructed based on right child views V_4 and V_7. Fig. 8 shows the rows contained in four child views V_1, V_3, V_4 and V_7. Consider the referenced table S with 4 rows, as its primary keys 1 and 2 are contained in V_1 and V_3, we set both the status vectors of the two rows as $(1, 1)$. Moreover, both the status vectors of rows 3 and 4 are set as $(1, 0)$. This is because their primary keys are only contained in V_3. Similarly, for the referencing table T with 8 rows, the status vector of row 6 is set as $(1, 0)$. This is because its foreign key is only contained in V_4.

Next, we propose to partition the two joined tables according to their status vectors. Specifically, if the status vectors of two rows have the same value, we put them into the same partition. For example, consider table S in Fig. 9. The status vectors of its 4 rows are $(1, 1)$, $(1, 1)$, $(1, 0)$ and $(1, 0)$, respectively. Then, we partition S into two partitions S_1 and S_2, where S_1 consists of rows $1 \sim 2$ and S_2 consists of rows $3 \sim 4$. Similarly, table T can be partitioned into T_1, T_2 and T_3. Since the dimension of status vectors is m, the number of partitions is at most 2^m. Note, we observe that the number of partitions is much smaller than 2^m in real-world applications.

Note, we observe that the number of partitions is much smaller than 2^m in real-world applications. More specifically, recall that we always firstly push down selection operators (see §3), then each child view V_l^k (resp. V_r^k) is the output of the selection view in S (resp. T). Thus, we can infer that the status of a row in table S/T is determined by the selection result on that table. For example, given a row in S, if the k^{th} bit in its status vector is set as 1, then it must be in the output of the selection view under V_l^k. Further, as described in §4.1, our selection view elimination procedure can reduce any selection view with a logical predicate into the view with a unary or an arithmetic predicate. Suppose there exist j unary and $m - j$ arithmetic selection views after reduction. We assume that the j unary selection views cover α columns in the partitioned table, and each of these columns is associated with x_1, $\cdots$, x_α unary selection views, $s.t. \sum_{i=1}^{\alpha} x_i = j$. Then, according to our cumulative distribution function based analysis described in §4.2, the parameters in x_i unary selections would divide the domain space of a column into at most $x_i + 1$ ranges. Since all the rows in a range must be selected/filtered together by a given unary selection, we can infer that the j bits that are associated with unary selection views in the status vector would at most have $\Pi_{i=1}^{\alpha}(x_i + 1)$ distinct values. For the $m - j$ bits associated with arithmetic selection views, they have at most 2^{m-j} distinct values even if they are independent of each other. Taken together, the number of partitions is at most $\Pi_{i=1}^{\alpha}(x_i + 1) \cdot 2^{m-j}$. In real-world applications, we observe that there usually exist a few arithmetic selections on each table. For example, the number of arithmetic selections on a table is only 3 in the TPC-H workload. In addition, we also observe that the unary selection views usually cover a small number of columns in each table (i.e., α is small). For example, the unary selection views cover at most 4 columns in any given table in the TPC-H workload. Thus, the actual number of partitions is much less than 2^m.

(2) Derive Populating Rules Based on Partitions. Consider the status vector of a referenced partition S_i, if its k^{th} bit is set as 1, then we can infer that all the primary keys in S_i are contained in V_l^k. For a referencing partition T_i, setting its k^{th} bit as 1 indicates that all the foreign keys in T_i are contained in V_r^k. For ease of presentation, hereinafter we use S_i/T_i to represent the PK/FK partitions. Then, for any join view V^k, key population from V_l^k to V_r^k is

$$V_l^k \rightarrow V_r^k = \bigcup_{S_i \subseteq V_l^k} S_i \rightarrow \bigcup_{T_j \subseteq V_r^k} T_j = \bigcup_{S_i \subseteq V_l^k, T_j \subseteq V_r^k} S_i \rightarrow T_j \qquad (2)$$

Based on the decomposition in Equation 2, we further formalize two populating rules for each pair of partitions S_i and T_j. Specifically, the two rules require that we should select z primary keys in S_i and use them to populate y foreign keys in T_j.

$$\begin{aligned} PF_{S_i \rightarrow T_j} &= y \qquad s.t. \quad y \in \{0,\ 1,\ \ldots,\ |T_j|\} \\ PF^d_{S_i \rightarrow T_j} &= z \qquad s.t. \quad z \in \{0,\ 1,\ \ldots,\ |S_i|\} \end{aligned} \qquad (3)$$

In addition, from Equation 2, we can also convert the join constraints on any join view (see Equation 1) into the join constraints on their associated partitions. Here, n^k_{jcc} and n^k_{jdc} denote the join cardinality and join distinct constraints on the k^{th} join view.

$$PF_{V_l^k \rightarrow V_r^k} = \sum_{S_i \subseteq V_l^k,\ T_j \subseteq V_r^k} PF_{S_i \rightarrow T_j} = n^k_{jcc} \quad PF^d_{V_l^k \rightarrow V_r^k} = \sum_{S_i \subseteq V_l^k,\ T_j \subseteq V_r^k} PF^d_{S_i \rightarrow T_j} = n^k_{jdc}$$

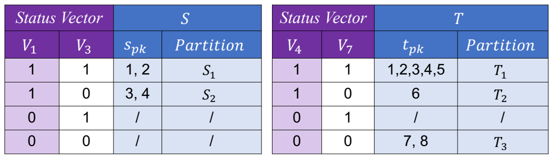
$$PF_{\overline{V_l^k} \rightarrow V_r^k} = \sum_{S_i \subseteq \overline{V_l^k},\ T_j \subseteq V_r^k} PF_{S_i \rightarrow T_j} = |V_r^k| - n^k_{jcc} \qquad (4)$$

Note, for any partition S_i in V_l^k, if its primary keys are selected to populated foreign keys in more than one partitions in V_r^k, we should avoid selecting a primary key repeatedly to populate different partitions in V_r^k. If not, we would have $\sum_{S_i \subseteq V_l^k,\ T_j \subseteq V_r^k} PF^d_{S_i \rightarrow T_j} < n^k_{jdc}$.

Example 5.3. Consider the join view $V_5 = \bowtie^{=}_{t_{fk}} (V_3, V_4)$ in Example 5.1 again. From Fig. 9, we can see that $V_3 = S_1$, $\overline{V_3} = S_2$ and $V_4 = T_1 \cup T_2$. Then, its three join constraints can be converted as

$$PF_{V_3 \rightarrow V_4} = PF_{S_1 \rightarrow T_1} + PF_{S_1 \rightarrow T_2} = 3 \quad PF_{\overline{V_3} \rightarrow V_4} = PF_{S_2 \rightarrow T_1} + PF_{S_2 \rightarrow T_2} = 3$$
$$PF^d_{V_3 \rightarrow V_4} = PF^d_{S_1 \rightarrow T_1} + PF^d_{S_1 \rightarrow T_2} = 2.$$

Obviously, our populating rules should also make sure that the foreign keys in each partition should be exactly covered. That is,



Status Vector		S	
V_1	V_3	s_{pk}	*Partition*
1	1	1, 2	S_1
1	0	3, 4	S_2
0	1	/	/
0	0	/	/

Status Vector		T	
V_4	V_7	t_{pk}	*Partition*
1	1	1,2,3,4,5	T_1
1	0	6	T_2
0	1	/	/
0	0	7, 8	T_3

Figure 9: Partitions of S and T According to Status Vectors

for any partition T_j, its total number of foreign keys populated by partitions in table S must equal to its number of rows.

$$\sum_{S_i \subseteq S} PF_{S_i \to T_j} = |T_j| \tag{5}$$

(3) Construct Constraint Programming Problem. As shown in Equations 3 ∼ 5, each populating rule specifies a specific constraint on fusing multiple join constraints on partitions. By combining all of these populating rules (i.e. equations), we can formalize the problem as a classic *Constraint Programming* (*CP*) problem [20]. Then, after finding a feasible solution for a set of variables stated in the constraint equations, we can follow it to populate foreign keys in each partition T_j. However, decomposing populating rules of join views into partitions would enlarge the solution space and lead to contradictory solutions.

Example 5.4. Suppose there exist two join views $V_5 {=} \Join^{=}_{t_{fk}} (V_3, V_4)$ and $V_8 {=} \Join^{l}_{t_{fk}} (V_1, V_7)$ on tables S and T (see Fig. 7). According to the table partitions in Fig. 9 and populating rules in Equations 3 ∼ 5, we can find a solution as follows.

$$PF_{S_1\to T_1}{=}3 \quad PF_{S_1\to T_2}{=}0 \quad PF_{S_2\to T_1}{=}2 \quad PF_{S_2\to T_2}{=}1$$
$$PF^d_{S_1\to T_1}{=}0 \quad PF^d_{S_1\to T_2}{=}2 \quad PF^d_{S_2\to T_1}{=}3 \quad PF^d_{S_2\to T_2}{=}0$$

Note, as the partition T_3 is not contained in any of the four child views V_1, V_3, V_4 and V_7, then it does not affect the output of V_5 and V_8. Thus, its foreign keys can be populated with arbitrary primary keys in S. However, we still observe three contradictions in this solution. First, as $PF_{S_1\to T_2} < PF^d_{S_1\to T_2}$, we cannot use 2 different primary keys from S_1 to populate 1 foreign key in T_2. Second, $PF^d_{S_1\to T_1} = 0$ but $PF_{S_1\to T_1} > 0$, it indicates that there is no primary keys available in S_1 to populate foreign keys in T_1. Finally, $PF^d_{S_2\to T_1} > |S_2|$, it means that there are not enough distinct primary keys in S_2 to meet the requirement of foreign key population in T_1.

To address this issue, we further introduce three constraints to ensure the validity of populating results.

- Composability. The number of populated foreign keys must not be less than the number of populated distinct foreign keys, i.e., $PF_{S_i\to T_j} \geq PF^d_{S_i\to T_j}$
- Expressibility. If a partition S_i uses its primary keys to populate foreign keys in another partition T_j, then at least one primary key is used, i.e., $PF_{S_i\to T_j} > 0 \Rightarrow PF^d_{S_i\to T_j} > 0$
- Coverability. The total number of selected primary keys must not exceed the size of primary key candidate set (recall that we cannot select a primary key repeatedly to populate foreign keys of different partitions in V^k_r), i.e., $|S_i| \geq \sum_{T_j \subseteq V^k_r} PF^d_{S_i\to T_j}$

After integrating all the three constraints into the *CP* problem, we can take advantage of existing *CP solver* to find a feasible solution. Specifically, we propose to use the *Or-Tools* [19] as our *CP solver*. It improves its efficiency by utilizing the constraint propagation method to prune the search space.

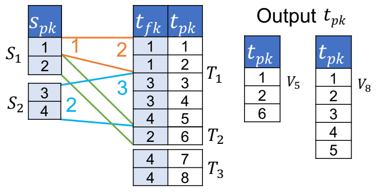


Figure 10: Unified Population for t_{fk}

Example 5.5. Consider the two join views V_5 and V_8 in Example 5.4. If we include three constraints described above, we would find a feasible solution without contradictions as follows.

$$PF_{S_1\to T_1}{=}2 \quad PF_{S_1\to T_2}{=}1 \quad PF_{S_2\to T_1}{=}3 \quad PF_{S_2\to T_2}{=}0$$
$$PF^d_{S_1\to T_1}{=}1 \quad PF^d_{S_1\to T_2}{=}1 \quad PF^d_{S_2\to T_1}{=}2 \quad PF^d_{S_2\to T_2}{=}0$$

Fig. 10 presents the result of populating foreign keys based on this solution. Specifically, from $PF_{S_1\to T_1} = 2$ and $PF^d_{S_1\to T_1} = 1$, we should select one primary key from S_1 and use it to populate two foreign keys in T_1. In Fig. 10, we use the first primary in S_1 to populate the first two foreign keys in T_1. Similarly, we use the second primary in S_1 to populate the foreign key in T_2, and use two primary keys in S_2 to populate the last three foreign keys in T_1. In addition, recall that the foreign keys in T_3 can be populated by arbitrary primary keys in table S, we populate with primary key 4.

5.3 Join Constraints on Multiple Tables

We now extend our method to support join constraints on multiple tables. For any referenced table S and referencing table T, after fusing all their join constraints and populating foreign keys in T, we can directly get the join result of each join view on S and T. Moreover, we can also derive the primary keys of table T output by each join view, and further use them to populate the foreign keys in another table which references T. For example, as shown in Fig. 10, after populating foreign keys in table T, we can derive its primary keys t_{pk} in the output of two join views V_5 and V_8 in Fig. 7.

Based on this observation, we propose to build a directed graph according to the *PK-FK* references between all tables. Specifically, each table is represented as a vertex in the graph, and if the primary keys in a given table S are referenced by the foreign keys in another table T, we add a link between the corresponding vertices. Then, we find a topological order of the graph by using the graph sorting algorithm [10]. Finally, we populate foreign keys in each table according to their topological order. More specifically, after populating foreign keys in a specific table T, we compute the intermediate join result regrading T and the table referenced by T for all the multi-table join views containing T. Then, we collect T's primary keys in the join result. Next, by applying the method described in §5.2, we use these primary keys to populate the foreign keys in another table referencing T.

6 DISCUSSION ON THE ERROR BOUND

In this section, we discuss the bound of errors introduced by our non-key column generator and key column generator. For ease of presentation, hereinafter we use $|V_i|$ to represent the required output size imposed by the cardinality constraint on the i^{th} query operator view, and use $|\hat{V_i}|$ to represent its actual output size on our simulated database.

Error Bound of the Non-key Column Generator. There exist three kinds of cardinality constraints involved in non-key columns, which are *LCC*, *UCC* and *ACC*. First, based on the decoupling rules $rule_1 \sim rule_3$ in §4.1, all *LCC*s can be decomposed to *UCC*s and *ACC*s with the help of equivalent transformations. Thus, for the V_i from a *LCC*, we can guarantee that $|V_i| = |\hat{V_i}|$ as long as we ensure that the decomposed *UCC*s and *ACC*s are satisfied. Second, for the case of V_i associated with a *UCC*, as the data distribution of each non-key column is derived according to all the *UCC*s on that column. Theorem 6.1 proves that our data generation and parameter instantiation methods based on *CDF*s (see §4.2 and §4.3) could guarantee $|V_i| = |\hat{V_i}|$. Finally, for the case of V_i associated with an *ACC*, if we calculate the parameters of an *ACC* based on the whole of generated data (in §4.4), we have $|V_i| = |\hat{V_i}|$. However, to avoid memory overflows caused by large volumes of data, small sampling data is used to calculate the parameters. Even so, we still have the error bound as δ in a confidence level α if the sampling data size is no less than $\frac{ln2-ln(1-\alpha)}{2\delta^2}$.

Theorem 6.1. *For any query operator view V_i whose cardinality constraint is a UCC, our non-key generator guarantees that $|\hat{V_i}| = |V_i|$.*

Proof. Revisit our algorithm in §4.2. First, we transfer *UCC* into distribution requirements $F_A(p)$ or $f_A(p)$ in §4.2, where p can a parameter. Secondly, we divide whole domain space of the non-key column into multiple sub-ranges by ordering the distribution requirements from $F_A(p)$. Therefore, $F(p)$ is exactly equal to the accumulation of the probablities from preceding sub-ranges, i.e., the following equation. Then, the step dose not introduce any error for $F(p)$.

$$\sum [F_A(p_i) - F_A(p_j)] = [F_A(p) - F_A(0)] = F_A(p)$$
$$s.t.,\ 0 \le p_i < p_j \le p\ \&\ \nexists p \in (p_i, p_j)$$

Then, we put the distribution requirements from $f_A(p)$ in these sub-ranges by solving bin-packing problem. Because putting $f_A(p_k)$ into range $F(p_i, p_j)$ dose not change the probability of $f_A(p_k)$, the step dose not lead any error of $f_A(p)$. After putting $f_A(p_k)$ into $F(p_i, p_j)$, the sub-range $(p_i, p_j]$ can be divided into $(p_i, p_k - 1]$, $(p_k - 1, p_k]$ and $(p_k, p_j]$. We update the probabilities of the new ranges with the remaining probability, i.e., $F(p_i, p_k - 1) + F(p_k, p_j) = F(p_i, p_j) - f_A(p_k)$. Therefore, the total probability of sub-range $(p_i, p_j]$ dose not change, i.e., the step dose not still introduce any error for $F(p)$.

In summary, our probability assignment does not introduce any error for distribution requirements $F(p)$ and $f(p)$. In §4.3, we exactly generate $|R| * F(p_i, p_j)$ values for each range $(p_i, p_j]$. Therefore, the non-key generator can satisfy *UCC*, i.e., $|UCC| = |\hat{UCC}|$. □

Error Bound of the Key Column Generator. As all the equations in §5.2 are strictly constructed to satisfy the *JCC*s and *JDC*s of all join views, the procedure of fusing populating results from multiple join views would not introduce any error. Thus, we focus on the error bound of dealing join constraints for a single join view. Suppose the join view V_i has the constrains of n_{jcc_i} and n_{jdc_i}, and its left and right child input views are V_l^i and V_r^i, respectively. According to the Equation 1 in §5.1, the population of a join view can be achieved when and only when the number of input primary keys in V_l^i is no less than n_{jdc_i} and the number of input rows in V_r^i is no less than n_{jcc_i}, i.e.,

$$|V_l^i| \ge n_{jdc_i} \quad |V_r^i| \ge n_{jcc_i}\ \ s.t.\ \ n_{jcc_i} \ge n_{jdc_i} \tag{6}$$

If the data sampling method is not applied when solving the *ACC*s, we can guarantee that the output size of each selection view in the simulated database is exactly the same as that of the original database. Then, as the join executes after selection, the input size of each join view also equals to that of the original database. Thus, we must have a solution satisfying the constraints of n_{jcc_i} and n_{jdc_i}. Similarly, there also exists no error for the subsequent join views in the same query. However, the sampling method would lead to the case in which the input size of a join view (i.e., the output size of a selection view) in the simulated database is smaller than that of the original database.

More specifically, if there exists that $|\hat{V_r^i}| < |V_r^i|$, it might lead to the case of $|\hat{V_r^i}| < n_{jcc_i}$, breaking the requirement in Equation 6. In this case, we propose to resize the constraint n_{jcc_i} to $|\hat{V_r^i}|$, which is the largest valid constraint that can be satisfied by the current input size $|\hat{V_r^i}|$. In this way, we can reduce the errors to the utmost extent. Then, from $|V_r^i| \ge n_{jcc_i}$ in Equation 6, we can derive that relative error introduced by such a resizing is also bounded by δ:

$$\frac{|(n_{jcc_i} - |\hat{V_r^i}|)|}{n_{jcc_i}} = |1 - \frac{|\hat{V_r^i}|}{n_{jcc_i}}| \le |1 - \frac{|\hat{V_r^i}|}{|V_r^i|}| \le \delta \tag{7}$$

Similarly, we have the same conclusion for the relative error of the constraint n_{jdc_i}.

Note that the relative error for any join view can be calculated in the same way as Eqn. 7. Therefore, the relative error of each join view is no bigger than the relative error introduced by its input views. So the relative error of any subsequent join view is no bigger than the error bound δ of the initial select views.

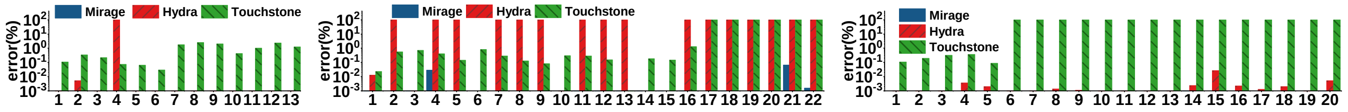
In summary, when *Mirage* does not sample data for deciding parameters of *ACC*, it can generate data satisfying all *CC*s without any error, or else the relative error of each view cannot exceed δ, which can be adjusted considering the memory limitation (in §4.4). In our experiment, it is verified to be an especially small number.

7 RELATED WORK

Data-aware generators and query-aware generators are two types of synthetic database generation methods.

Data-aware generators [1, 4, 9, 23, 25] aim to generate a database closely related to the real one in data characteristics and the generation of a database instance is independent of workload. For example, *Alexander* [1] designs pseudo-random number generators and *Torlak* [25] uses kinds of multi-dimensional models. Although they have similar data as in distribution, the performance cannot be guaranteed by running the in-production workload due to the inconsistency of database instance [12].

Query-aware generators [2, 3, 7, 12–14, 22] aim to generate a database that satisfies the cardinality constraints extracted from the original workload. This method expects an accurate simulation of the performance of original queries. *QAGen* [3] is the first work for query aware database generation. It has powerful support in



(a) Query of SSB (b) Query of TPC-H (c) Query Group of TPC-DS

Figure 11: Relative Errors for Various Workload including SSB, TPC-H and TPC-DS (100% error means no support)

operators, but can only generate one database for one query at a time. Subsequent work focuses on query aware database generation of multiple queries, which has been proved NP-Complete [2]. Then these works make a great effort to reduce the computational complexity rather than improve the support capability to operators. Following QAGen, *MyBenchmark* [14] adopts a Query-Oriented Divide and Conquer solution. It first uses QAGen to generate an individual database for each query and then tries its best to merge them. However, MyBenchmark can not guarantee to generate one single database. *Touchstone* [22] then proposes to generate one database by launching a k-round of random distribution samples satisfying some specific rules and then choosing the one with the minimum simulation error. But it cannot guarantee a low simulation error theoretically or satisfy the cardinality constraints from the arbitrary logical predicate. *DCGen* [2] and *Hydra* [22] transfer the generation task into multiple *Linear Programming* (LP) problems. However, the LP model cannot be adapted for the arithmetic filter operator, outer join or semi join. Compared to these works, *Mirage* can give the most powerful support to complex operators and guarantee simulation error to be *zero* theoretically.

8 EXPERIMENT

We conduct extensive experiments on *Mirage*. 1) To tell whether *Mirage* can conquer the state-of-the-art work in application scenario simulation; 2) To expose the effectiveness of the technology designed for *Mirage*.

Database: Since all generators emphasize to guarantee the cardinality constraints of operators, we can take any database to present the simulation results. Here we select PostgreSQL (v.14.2) as the test database for it provides a convenient interface to check the cardinality constraints of relations and annotated query templates.. The deployed machine is equipped with 2 × Intel(R) Xeon(R) Gold 6240R CPU, 390 GB memory, and 2.5 TB disk.

Workloads: We compare *Mirage* with the two most recent work, i.e., *Touchstone* [12] and *Hydra* [22] with their open source codes. Three classic benchmarks, i.e., SSB [18], TPC-H [27] and TPC-DS [26], play as real application scenarios, which are easy to reproduce the experiment results. SSB is a benchmark with 13 simple structured queries. It is almost supported by *Touchstone* and *Hydra*. TPC-H is the most popular OLAP benchmark with 22 queries having complex operators, e.g., various joins. It cannot be fully simulated by any previous work. In order to test the scalability of workload, *Hydra* produces a scenario with a large scale of workload from TPC-DS, but removes its unsupported complex operators, e.g., *arithmetic* selections. For fair comparison, we take all 100 distinct queries.

We take the official generation tools designed for these three benchmarks [16, 26, 27] to compose the input real applications, i.e., database instances and queries. In all figures, the benchmark name means the corresponding real application. Due to the slow data import speed of database, when comparing performance on the database, we take a small scale factor SF=1; for demonstrating the generation efficiency of tools, we use a large default SF=200.

Metrics: We adopt *relative error* $= \frac{\sum ||V_i|-|\hat{V}_i||}{\sum |V_i|}$ [12] to measure the simulation fidelity (accuracy) for query Q. $|V_i|$ (resp. $|\hat{V}_i|$) represents its output size of the i^{th} view V_i (resp. $\hat{V}_i$) in query Q (resp. the instantiated query $\hat{Q}$) on the real database (resp. the simulated database). *Relative error* represents the cardinality deviation between Q and $\hat{Q}$, and the smaller is better. Specially, the relative error of an unsupported query is 100%.

Setting: To make evaluation general for both small/large databases, we take the sampling-based parameter instantiation for ACC. A default sampling size is 4 million (4M) rows for calculating parameters of *ACC* with its theoretical error bound 0.1% in a confidence level 99.9% according to Hoeffding's Inequality. In order to control memory usage for enormous database generation, we use a batch generation strategy. Specifically, we generate non-key columns based on their distribution for each batch, and then populate foreign keys for the batch based on the join constraints scaled by batch size. The default batch size is set as 7M rows.

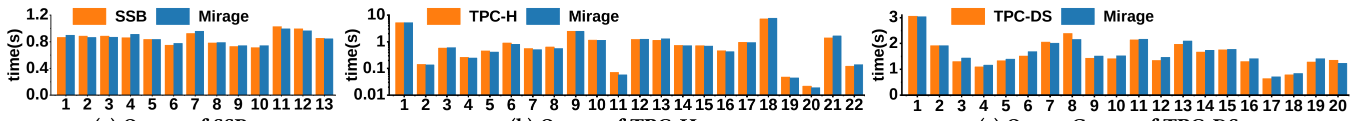
8.1 Comparing with Existing Methods

We first compare workload support ability as well as the simulation fidelity with the state-of-the-art work, i.e., *Touchstone* and *Hydra*, in §8.1.1; we then compare their generation efficiency in §8.1.2.

8.1.1 Comparison of Workload Support ability and Generation Fidelity. *Touchstone* and *Hydra* have inconsistent supports to operators. We plot the relative errors for the benchmark queries in Fig. 11. Note that queries with relative error=100% are not supported. To plot the errors clearly for TPC-DS, we divide every 5 queries of TPC-DS into a group and show the relative errors of each group.

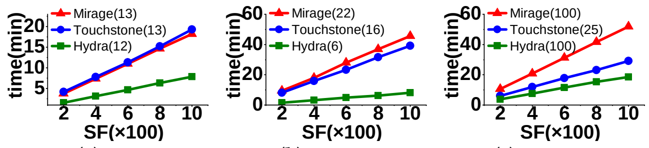
For SSB (in Fig. 11a). *Touchstone* and *Mirage* can support all queries of SSB. *Touchstone* has simulation errors for all SSB queries introduced by its random sampling-based data generation mechanism, but all errors are small (< 2.51%) for the simplicity of workload. *Mirage* achieves an accurate simulation without any error. For *Hydra*, it divides query aware generation into several linear programming (*LP*) tasks based on table reference relationships, which are processed independently and then combined into a single solution. It may introduce slender deviations for the simulation result even for the simple SSB workload, e.g., 7 rows of deviation of Q_2; the worst thing is that *Hydra* can not support >, < comparator on string column causing a severe deviation (100% relative error) for Q_4.

For TPC-H (in Fig. 11b). *Touchstone* can simulate an application scenario for the first 16 queries of TPC-H, i.e., Q_{1-16}. But a lack of the support of complex *logical predicate, non-equal join* and *projection on FK*, *Touchstone* is not applicable for the remaining 6 queries, i.e., Q_{17-22}. Though its random sampling-based data generation method cannot give a theoretical error guaranteeing in simulation, for the first 16 queries, it still has stable low errors (< 5%). Compared to *Touchstone*, besides the mentioned unsupported predicates



(a) Query of SSB (b) Query of TPC-H (c) Query Group of TPC-DS

Figure 12: Comparison of Query Latency for Various Workload



(a) SSB (b) TPC-H (c) TPC-DS

Figure 13: Comparison of Generation Efficiency

and operators, *Hydra* can not support *arithmetic* or *pattern matching* predicates (e.g., *like*). Finally, it can only process 6 queries of TPC-H, i.e., $Q_{1,3,6,10,14,15}$, though with relative low errors. *Mirage* supports application simulation to all 22 queries with almost *ZERO* errors ($< 0.001\%$ for 19 queries); Q_4 and Q_{21} meet a higher relative error but still $< 0.08\%$. These errors are introduced by the batch-based parameter instantiation algorithm for *ACC* used to control memory usage, but the theoretical error will be bound by 0.1% (as discussed in §4.4).

For TPC-DS (in Fig. 11c). *Touchstone* only processes simple logical predicates and requires a table refers to the referenced table at most once, and finally it supports 45 queries of TPC-DS. Both *Hydra* and *Mirage* can simulate operators in TPC-DS completely. To plot the errors clearly, we divide every 5 queries from TPC-DS into a group, and show the relative errors of each group. The relative errors for *Mirage* are almost *ZERO* (i.e.,as small as 3 rows) for all queries, which is caused by its batch-based generation method as analyzed above. *Hydra* still has simulation errors even for its own preferred workload, which are introduced by merging small *LP* tasks. *Touchstone* generates non-key columns by random sampling, based on which to populate the foreign keys (if any). Though its maximum error for the first 5 groups of queries is less than 1%, it is impossible for *Touchstone* to finish populating foreign keys (or achieve data generation) after 25 queries (i.e., 5 groups), because it cannot scheme a feasible solution to populate foreign key columns caused by the ignorance of non-key distributions. It means *Touchstone* may not be scalable to a large number of queries.

In summary, *Mirage* totally conquers the state-of-the-art work in workload support ability with confidently low errors. It **gives the widest support to operators** and is **the first work** that accomplishes simulation of TPC-H with near-*zero* errors.

8.1.2 Comparison of Generation Efficiency. OLAP database is usually of a big volume, which requires the generation tool to have a high generation speed. We compare generation time among *Mirage*, *Touchstone* and *Hydra* in Fig. 13 by changing *SF* from 200 to 1000 for all scenarios. Note that each tool only generates the application scenario with its supported queries (size labelled on figures).

Among them, it looks like *Hydra* performs the best as in Fig. 13, but it has bypassed too many complex operators or predicates. For the simplest SSB (in Fig. 13a), it can not support the simulation on the whole workload; and for TPC-H, it can only support 6 out of 22 queries. *Mirage* and *Touchstone* have almost the same generation efficiency for SSB and TPC-H, but *Touchstone* is not scalable for a large size of queries, i.e., supporting only 25 queries in TPC-DS. Though *Mirage* is 2× slower than *Hydra* on TPC-DS, it has elaborate designs for complex operators, i.e., the strongest support to all queries and still guarantees a linear generation speed.

In summary, *Mirage* performs the best to simulate a **complex application scenario in a linear way**.

8.2 Technical Design of *Mirage*

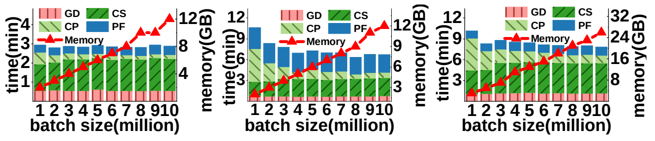
After comparison with related work, we run experiments to expose the effectiveness of our technical design in *Mirage*.

8.2.1 Fidelity of Simulation. When running the instantiated synthetic query $\hat{Q}$ in the same environment as the original query Q, if we have a lower deviation of the query latencies between Q and $\hat{Q}$, we have a better fidelity. To avoid additional deviations introduced by other factors, we make the following two settings to PostgreSQL: 1) turn off parallel query feature to avoid the deviations caused by multi-thread task scheduling; 2) set *shared_buffer* to 20GB and warm up two rounds to refrain from undesirable side-effects caused by page swapping. For TPC-DS, we present the accumulated latency for each group. For all these workloads, the query latencies of the instantiated queries by *Mirage* do not produce significant deviations from the original ones as in Fig. 12. The mean deviations of latencies for SSB, TPC-H and TPC-DS are all less than 6%.

In summary, *Mirage* guarantees **a high fidelity** on simulating an application scenario.

8.2.2 Memory Consumption. Maintaining the complex data dependency, e.g., PK-FK dependency, has always bottlenecked the data generation by consuming too much memory [12, 22]. To balance memory usage and generation performance, *Mirage* can generate data in batch. Given a table, a large batch usually occupies more memory, but it decreases the total rounds of generations. Here, we illustrate the impact of batch size on generation efficiency and memory consumption in Fig. 14. We divide the entire generation phase into the following four stages: 1) to generate a batch of data based on *CDF* (*GD*); 2) to compute join statuses according to all the outputs of selection operators (*CS*); 3) to solve *Constraint Programming* Problem (*CP*); 4) to populate foreign keys (*PF*). Tables are generated in their partial order of references, or else can be totally in parallel.

We change the batch sizes from 1*M* to 10*M* rows in Fig. 14. For



(a) SSB (b) TPC-H (c) TPC-DS

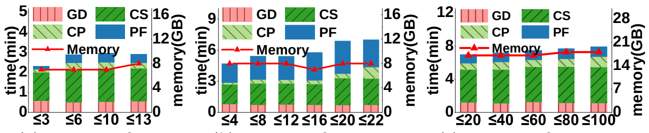
Figure 14: Batch Size *vs.* Generation Efficiency

a given workload, the generation time for *GD*, *CS*, and *PF* is only relevant to batch size, which is stable. However, a larger batch size reduces the number of rounds to solve *CP*, which decreases generation time for *CP* gradually. But it also shows that the gain decreases as n increases. For example, the turning point is $4M$ rows for TPC-H, after which we obtain only a slender performance improvement from *CP*. Fig. 14 also shows the memory required to guarantee the generation is linearly related to the batch size. Since TPC-DS has wider tables, it consumes memory for data generation about twice as much as that of TPC-H or SSB. To trade off the overall generation efficiency against memory usage, taking the batch sized $7M$ rows (the default setting) almost uniformly reaches the peak performance for all benchmarks. In such a setting, the maximum memory used for generation is no more than 18GB.

In summary, *Mirage* can reach its peak performance with **conservative memory** usages decided by the batch size. The batch size trades off memory usage against generation performance.

8.2.3 *Generation Efficiency under Different Workload Scale.*

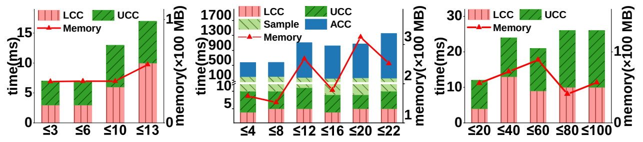
We expose the generation efficiency under different numbers of queries in Fig. 15. The input queries are increased stepwise for each workload.



(a) Query of SSB (b) Query of TPC-H (c) Query of TPC-DS

Figure 15: The Number of Queries *vs.* Generation Efficiency

Given the table size and the batch size, it has always the stable data generation time for batch data generation and foreign key population, i.e., *GD* and *PF*. Though we may have more join operators as increasing queries, *Mirage* assigns each bit of status in the join status vector in parallel and it has been stable in computing status vectors (*CS*). However, the time for solving the *constraint problem* (*CP*) is related to the complexity of constraints from workload. Except for TPC-H (in Fig. 15b), the other two workloads (in Fig. 15a and Fig. 15c) contain only equal join cardinality constraints (*JCC*s); TPC-H workload covers both *JCC*s and *JDC*s. As increasing queries, the time for solving the *CP* problem *w.r.t JDC*s (in Fig. 15b) has about 6× more time increasing than the one for *JCC*s (in Fig. 15a and Fig. 15c). However, the proportion of time for *CP* to the whole generation time is small, so its variation hardly affects the overall performance. Therefore, *Mirage* achieves efficient generation scalability for these worloads. More queries may increase the dimensions of status vectors, i.e., more joins. Since each row of data needs only one bit for marking its existence status, the memory consumption is far less than the raw data. Finally, the memory consumption for these workloads is stable as queries increased stepwise in Fig. 15.



(a) Query of SSB (b) Query of TPC-H (c) Query of TPC-DS

Figure 16: The Number of Queries *vs.* the Time for Portraying Distribution as well as Memory Usage

Besides the generation efficiency analysis from key columns, in Fig. 16,we expose the efficiency for portraying distributions of non-key columns and the parameter instantiation for *ACC* on these columns together with the corresponding memory consumption. Constructing a non-key generator includes: 1) decoupling dependencies from *LCC*; 2) dealing with *UCC* for data distribution; 3) sampling data for *ACC* (*Sample*) and 4) instantiating parameters of *ACC* (only exist in TPC-H). Our decoupling and distribution sketching algorithms are efficient enough to obtain *CDF* of any non-key column ($\leq 20ms$). Solving *ACC* costs more time for its data sampling and a result view computation. But it still can be completed within $2s$. The maximum memory cost is only about 300MB for TPC-H as in Fig. 16b.

In summary, the generation time of Mirage grows little under more queries in all three workloads, so Mirage achieves good scalability for workload. Portraying non-key column distributions is with conservative memory usages but fast enough to be negligible compared to the total data generation time.

9 CONCLUSION AND FUTURE WORK

In this paper, we propose *Mirage*, a query-aware data generator (*QAG*) provides the most powerful support to complex OLAP workload. To reduce the high computational complexity of data dependencies from various cardinality constraints, it designs a set of dependency decoupling methods for non-key columns and key columns respectively, and achieves data generation in a linear way with conservative memory usages. Moreover, our data generation scheme guarantees a *zero* simulation error bound theoretically. Compared with *Touchstone* and *Mirage*, we can totally dominate them in operator support ability and fidelity. And the experiments also confirm the design effectiveness of *Mirage*.

REFERENCES

[1] Alexander Alexandrov, Kostas Tzoumas, and Volker Markl. 2012. Myriad: scalable and expressive data generation. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1890–1893.

[2] Arvind Arasu, Raghav Kaushik, and Jian Li. 2011. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 685–696. https://doi.org/10.1145/1989323.1989395

[3] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 341–352. https://doi.org/10.1145/1247480.1247520

[4] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible database generators. In *Proceedings of the 31st international conference on Very large data bases*. VLDB, Trondheim, 1097–1107.

[5] Irving Copi, Carl Cohen, and Victor Rodych. 2016. Introduction to logic.

[6] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

[7] Amir Gilad, Shweta Patwa, and Ashwin Machanavajjhala. 2021. Synthesizing Linked Data Under Cardinality and Integrity Constraints. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 619–631. https://doi.org/10.1145/3448016.3457242

[8] Wassily Hoeffding. 1994. Probability inequalities for sums of bounded random variables. In *The collected works of Wassily Hoeffding*. Springer, America, 409–426.

[9] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and realistic data generation. In *Proceedings of the 32nd international conference on Very large data bases*. VLDB, Seoul, 1243–1246.

[10] A. B. Kahn. 1962. Topological Sorting of Large Networks. *Commun. ACM* 5, 11 (nov 1962), 558–562. https://doi.org/10.1145/368996.369025

[11] Ralph Kimball and Margy Ross. 2013. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd ed.). Wiley Publishing, America.

[12] Yuming Li, Rong Zhang, Xiaoyan Yang, Zhenjie Zhang, and Aoying Zhou. 2018. Touchstone: Generating Enormous Query-Aware Test Databases. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 575–586. https://www.usenix.org/conference/atc18/presentation/li-yuming

[13] Eric Lo, Nick Cheng, and Wing-Kai Hon. 2010. Generating Databases for Query Workloads. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 848–859. https://doi.org/10.14778/1920841.1920950

[14] Eric Lo, Nick Cheng, Wilfred W.K. Lin, Wing Kai Hon, and Byron Choi. 2014. MyBenchmark: generating databases for query workloads. *VLDB Journal* 23, 6 (2014), 895–913. https://doi.org/10.1007/s00778-014-0354-1

[15] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1123–1136. https://doi.org/10.1145/2723372.2747644

[16] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2010. Start Schema Benchmark. https://github.com/electrum/ssb-dbgen

[17] P E O'Neil, E J O'Neil, and X Chen. 2009. The Star Schema Benchmark (SSB).

[18] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, Springer, Springer, 237–252.

[19] Laurent Perron and Vincent Furnon. 2021. OR-Tools. https://developers.google.com/optimization/

[20] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. Handbook of constraint programming.

[21] Stuart J Russell and Peter Norvig. 2010. Artificial Intelligence (A Modern Approach).

[22] Anupam Sanghi, Raghav Sood, Jayant R. Haritsa, and Srikanta Tirthapura. 2018. Scalable and Dynamic Regeneration of Big Data Volumes. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose (Eds.). OpenProceedings.org, Vienna, Austria, 301–312. https://doi.org/10.5441/002/edbt.2018.27

[23] Entong Shen and Lyublena Antova. 2013. Reversing statistics for scalable test databases generation. In *Proceedings of the Sixth International Workshop on Testing Database Systems*. DBTest, New York, 1–6.

[24] Utku Sirin and Anastasia Ailamaki. 2020. Micro-architectural analysis of OLAP: limitations and opportunities. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 840–853. https://doi.org/10.14778/3380750.3380755

[25] Emina Torlak. 2012. Scalable test data generation from multidimensional models. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. SIGSOFT, Cary, 1–11.

[26] TPC. 1999. TPC-DS Benchmark. http://www.tpc.org/tpcds/.

[27] TPC. 1999. TPC-H Benchmark. http://www.tpc.org/tpch/.