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

A rich body of query-aware database generators (QAG) are proposed for application-driven database benchmarking or performance debugging. The complex data dependencies hidden behind queries make previous work suffer from critical deficiencies in supporting complex operators and controlling 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 PK-FK 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 with 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 database developers develop the query execution engine, it is of great importance to benchmark the performance and service stability of the new engine across various real application scenarios [2, 16, 17, 29]. Although classic benchmarks [24, 32, 36] 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 [29].

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, 4, 11, 15–17, 29] is proposed to generate a synthetic data processing environment to simulate the real-world application with specific data and workload characteristics. More specifically, given a number of annotated query templates (*AQT*) with labeled cardinality constraints and anonymized parameter variables that are abstracted from query traces, it requires that the synthetic query can mimic the execution cost in real-world applications. Note, the cost is closely related to the cardinality constraint [17] (i.e., the output size) of each operator on the query tree. Moreover, the parameter variables need to be anonymized because unanonymized ones could be leveraged to derive the distribution of real-world application data [22, 38, 39]. In addition, the QAG is also a prerequisite for performance debugging which occurs frequently when optimizing the query execution engine. In this case, the database developers need to reproduce a performance regression under the same query plan and the real-world application scenario such that they can understand whether their optimizations would accelerate the execution of some specific operators. Take the join operator in MariaDB as an example, its earlier version [19] builds a hash table for the hash join operator by reusing the table partitioning module since each partition can be regarded as a hash bucket. However, an expensive B+-tree is also built on hash buckets as the original partitioning module uses it to perform range scans on partitions. We can leverage the QAG to reproduce some serious performance issues when the join operator in real-world workloads has rather large-sized inputs.

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 [4, 16, 17] and are not scalable. These joint data distributions have been challenging the design of QAG. As a result, none of the existing work [2, 15–17, 29] can even support to mimic a complete TPC-H scenario with its 22 queries as shown in Table 1.

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**

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 selection and join operators in a query plan. If the join executes after the selection, it indicates that the join result relies on the selection result and the distribution of key columns relies on that of non-key columns, and vice versa. We propose to take the common relational algebra-based *query rewriting* method to push down selection 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 cardinality constraints on selections with logical or arithmetic predicates which involve multiple non-key columns connected by logical operators (e.g., $\wedge$) or arithmetic operators (e.g., $+$). 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 cardinality constraints on PK-FK join operators which impose some requirements on how primary and foreign key columns are matched. 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 [27] which can be simply solved by existing *CP Solver* [26].

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

1) *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.

2) *Mirage* guarantees simulation with a theoretical *zero* error bound even having more supported operators.

3) *Mirage* is able to generate a database of any data size in a linear way with controllable memory consumption.

4) We run extensive experiments. *Mirage* conquers the related work by a wide operator support and low simulation errors.

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 [15]. An annotated query template AQT [2] labels the cardinality constraint on the output of each operator op. For example, 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.

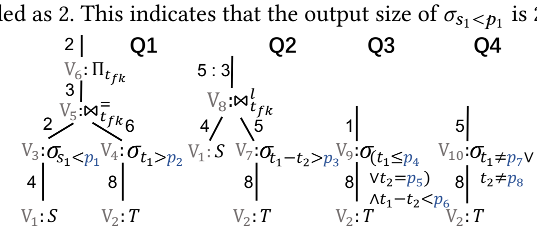


Figure 1: Annotated Query Templates on Tables S and T

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. Note that we ignore the operators whose output size does not vary with the data distribution because these operators do not impose any additional requirements of data distributions. For instance, the *orderby* operator retains the same number of output rows as its input.

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 2.1. 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 [28]. That is,

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

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 $\bullet \in \{=, \neq, <, >, \leq, \geq, (not)\ in, (not)\ like\}$, 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 $g(A_i, ..., A_k) \circ p_k$, $\circ \in \{<, >, \leq, \geq\}$, 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, 4, 11, 15–17, 29], 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), so *Mirage* can not simulate the workload if either of them is missing. For example, suppose view V_l left outer joins view V_r, and the cardinality constraints are $n_{jcc} : n_{jdc}$. We can infer that there exist $|V_l| - n_{jdc}$ rows that are not matched with the rows in V_r and its output size is $|V_l| - n_{jdc} + n_{jcc}$.

Table 2: Output size of joins on V_l and V_r.

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}	$\|V_l\|$-n_{jdc}+n_{jcc}	$\|V_r\|$	$\|V_l\|$-n_{jdc}+$\|V_r\|$	n_{jdc}	n_{jcc}	$\|V_l\|$-n_{jdc}	$\|V_r\|$-n_{jcc}

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 [15, 20]. Note that the projections on non-key columns in an OLAP database usually involve dimension columns. Although the dimension column can be of any data type, e.g., *time* [18], *textual data* [33], its cardinality is few for it is filled with category data. For example, the max cardinality of all the dimension columns in the dataset [18] is 2112, which is far less than the table size of 8.6 million. Therefore, the output size of projections is relatively small and then has a marginal effect on the performance. 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. Based on the observations, we only focus *PCC* on *FK* columns to simplify the complexity of query aware database generation.

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.

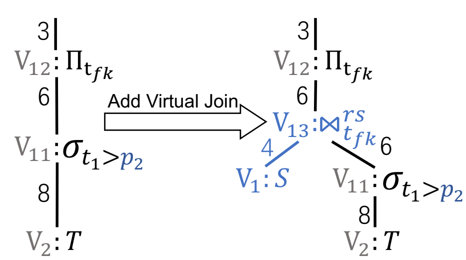


Figure 2: Add a Virtual Join View for the Projection View

2.3 Problem Definition

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

Definition 2.2. **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 2.3. 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.

3 DESIGN OVERVIEW

Fig. 3 shows an overview of our *Mirage* framework. Specifically, it uses a *workload parser* to collect the cardinality constraints in § 2.1 from the production database. For simplicity, we mainly discuss queries without subqueries. Note, our proposed techniques can be also extended to the case of subqueries (more details in § 7.2).

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. 3.

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, we observe that there exists some *AQT*s in which the selection executes after the join. For example, the residual predicate in TPC-H Q19 always executes after the join operator since it operates on multiple tables. To address this issue, we propose to rewrite the query tree based on algebraic transformations so as to push down the selection without breaking the cardinality constraints (more details in § 7.1). 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 (i.e., selection executes after the join) and all the cardinality constraints in that plan are guaranteed when performing tests on the simulated 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 selections with logical predicates and arithmetic predicates usually operate on multiple non-key columns, then the corresponding logical cardinality constraints (*LCC*) and arithmetic cardinality constraints (*ACC*) would bring the requirements of joint distributions between these columns. 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, based on the observation that the QAG problem is required to guarantee the output size of each operator instead of that of each sub-predicate, 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 by only considering the *UCC*s on each column (§4.2 and §4.3). After that, our aim is to find a valid parameter value for each *ACC* based on the populated non-key columns and its specified cardinality constraints (§4.4). Secondly, to populate a non-key column, we need to first derive its data distributions according to all 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 generated above (e.g., compute $t_1 - t_2$ for all rows in columns t_1 and t_2). 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 [15] and generate them by an auto-incrementing integer generator. Then, our main aim is to

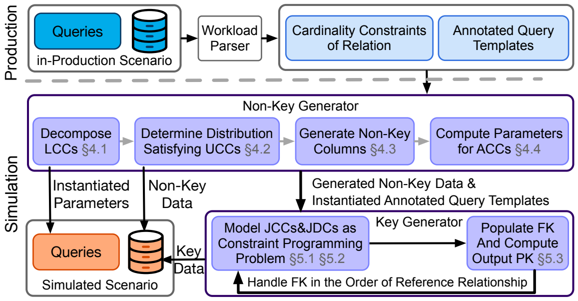


Figure 3: Design Overview of *Mirage*

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 *Constraint Programming* (*CP*) problem [27], which can be solved by existing *CP Solver* [26] (§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 (§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 $clause_k$:

$$|V_4| = |\sigma_{t_1>p_2}(T)|=6 \qquad |V_9| = |\sigma_{(t_1\le p_4\vee t_2=p_5)\wedge t_1-t_2<p_6}(T)|=1$$
$$|V_7| = |\sigma_{t_1-t_2>p_3}(T)|=5 \qquad |V_{10}| = |\sigma_{t_1\ne p_7\vee t_2\ne p_8}(T)|=5$$

(a) Selection Cardinality Constraints of Table T

$$UCCs\ on\ t_1{:}\ |V_4| = |\sigma_{t_1>p_2}(T)| = 6 \qquad |V_9^{1(1)}| = |\sigma_{t_1\le p_4}(T)| = 1$$
$$|\overline{V_{10}^{1(1)}}| = |\sigma_{t_1=p_7}(T)| = 3$$
$$UCC\ on\ t_2{:}\ |\overline{V_{10}^{1(2)}}| = |\sigma_{t_2=p_8}(T)| = 3$$
$$ACC\ on\ t_1, t_2{:}\ |V_7| = |\sigma_{t_1-t_2>p_3}(T)| = 5$$

(b) Selection Cardinality Constraints After View Eliminations

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

(c) Distribution For Table T

Figure 4: Selection Cardinality Constraints on Table T

Table 3: Assigned Boundary Values for View Elimination

Comparator		$>, \ge$	$<, \le$	*in, like,* $=$	*not in, not like,* $\ne$
Assigned Value	U	$-\infty$	$+\infty$	/	*NULL*
	$\emptyset$	$+\infty$	$-\infty$	*NULL*	/

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

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 $clause_k$. For example the selection view V_9 in Fig. 4a, 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 as follows.

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

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, we leverage the following two rules in the area of set theory to eliminate some subsub-selection views:

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

$Rule_1$ 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 $\emptyset$. 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. 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 4.1. Consider the selection view V_9 in Fig. 4a 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=+∞ and p_6=+∞. Then, we keep one subselection view and eliminate the other one. Suppose V_9^2 is eliminated by setting p_6=+∞. Secondly, we try to set subsub-selection views of V_9^1 as ∅. Specifically, both $V_9^{1(1)}$ and $V_9^{1(2)}$ can be ∅ 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 ≠, then none of the subsub-selection view can be set as ∅ (see Table 3). To address this issue, we make use of the *De Morgan's law* [6] 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. For example, consider the selection view V_{10} operating on table T in Fig. 4a. Its universal set size is $|T| = 8$. Then, we can convert the cardinality constraint of $|V_{10}| = 5$ as $|\sigma_{t_1=p_7}(T) \cap \sigma_{t_2=p_8}(T)| = 3$.

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

Theorem 4.2 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. 4b shows the elimination result of selection views V_4, V_7, V_9 and V_{10}.

THEOREM 4.2. *Given a selection view V, our elimination procedure can reduce it to one of the 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 ∅. 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 ∅. 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 ≠. 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

As each *UCC* involves a single column, we propose to use *UCC*s to derive the data distribution of each non-key column independently. Specifically, for each non-key column A, we use the cumulative distribution function CDF_A to represent its data distribution. Since different columns usually have various data types and domain spaces, we first normalize the original domain space of each nonkey 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 *UCC*s 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 4.3. **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 transformed into the union of multiple *UCC*s with the = comparator. Similarly, a *UCC* with a comparator *like* can be converted into an *in* comparator by querying the number of distinct matching values in the original database. In addition, based on commutativity property from *De Morgan's Law* [6], the comparators >, ≥ and ≠ can be easily converted to ≤ and =. For example, $|\sigma_{A>p}(R)| = k$ is equal to $|\sigma_{A\leq p}(R)| = |R| - k$.

Then, we can use the two kinds of *UCC*s to derive $F_A(p)$ and $f_A(p)$ respectively. If $|\sigma_{A\leq p}(R)|$=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. 4b. From $|\sigma_{t_1>p_2}(T)|$=6, $|\sigma_{t_1\leq p_4}(T)|$ =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 nonkey 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., $\sum_{j=1}^{k} f_A(p_{e_j}) \leq F_A(p_i, p_j)$. Then, the problem can be regarded as the conventional bin packing problem, which has been proven to be NP-hard [10]. 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 fit within any range $(p_i, p_j]$. To ensure the cardinality size of each *UCC*, we can first check whether there exists a parameter p' which has been in a range with its $f_A(p')$=$f_A(p)$. If so, we specify $p = p'$. Otherwise, we transform $f_A(p) = x$ into $f_A(p_1) = x_1, \cdots, f_A(p_q) = x_q$, with $x_1 + \cdots + x_q = x$. 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, the *UCC* with a smaller granularity of cardinality requirement is more likely 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 $F_A(p_{e_i}, p_{e_{i+1}}) \geq f_A(p_{e_{i+1}})$, 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 of $|R|_A - m - n - 1$ 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 4.4. Consider the column t_1 of table T in Example 2.3, and its domain size is 5, then we have $F_{t_1}(0) = 0\%$ and $F_{t_1}(5) = 100\%$. 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 $|T| = 8$.

An example is shown in Fig. 5. 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

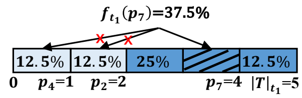


Figure 5: *CDF*s **of Column** t_1

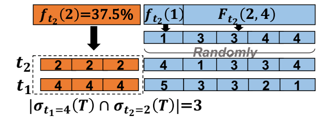


Figure 6: Generate Table T

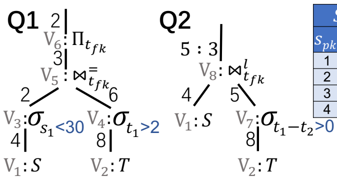
that is greater than $f_{t_1}(p_7)$, then we put p_7 in the range $(p_2, 5]$. This further splits the range into $(p_2, p_7]$ and $(p_7, 5]$. Note, we only need to guarantee that $F_{t_1}(p_2, p_7) \geq f_{t_1}(p_7)$, so we can allocate 25% and 12.5% to the two ranges, then we have $F_{t_1}(p_2, p_7)$ =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 $|T|^*F(0, p_4) = |T|^*F(p_4, p_2) = |T|^*F(p_7, 5) = 1$, 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. 4c shows the *CDF*s for non-key columns t_1 and t_2 after processing t_2.

4.3 Generate Data Based on CDF

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. For example in Fig. 6, column t_2 is populated with 3=$|T|^* f_{t_2}(2)$ rows valued 2 based on the *CDF*s in Fig. 4c. 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 are only concerned 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. For example in Fig. 6, we assign 4=$|T|^* F_{t_2}(2, 4)$ rows valued 3 or 4 uniformly into column t_2.

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 $|\cap_{j=1}^{\omega} V_e^j| = n$ requires 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 $\cap_{j=1}^{\omega} V_e^j$. For example in Fig. 6, based on the requirement $|\sigma_{t_1=p_7=4}(T) \cap \sigma_{t_2=p_8=2}(T)|$ from V_{10}, we bind 3 rows valued (4,2). Then we populate rows generated randomly 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.



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 → p_3
7	2	3	?	-1
8	1	4	?	-3

Figure 7: Non-key Column Data and Instantiated Queries

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 $|\sigma_{g(A_i \ldots A_j) \bullet p}(R)| = n$, where $g()$ is an arithmetic function, $A_i, \ldots, A_k$ are the non-key columns operated by $g()$, and $\bullet$ is a comparator. We first calculate the result view $g(A_i \ldots A_j)(R)$ 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. For example, consider the ACC $|V_7| = |\sigma_{t_1 - t_2 > p_3}(T)| = 5$ in Fig. 4b. 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. 7 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 [12], 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 ACCs, 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*.

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 input child views V_l and V_r. Next, we discuss how to populate foreign keys on the joined table according to the join constraint on two views.

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.

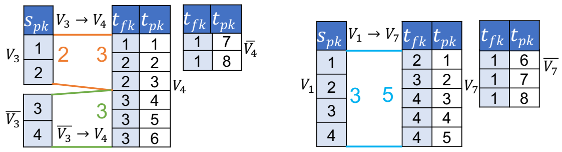
$$PF_{V_3 \to V_4} = n_{jcc}{=}3 \quad PF_{\overline{V_3} \to V_4} = |V_4| - n_{jcc}{=}3 \quad PF^d_{V_3 \to V_4} = n_{jdc}{=}2$$

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

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.

Note that, there exist some overlaps of primary keys between the input left child views. For example, consider the left child views



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

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

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. 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. Fig. 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 $(1, 1)$.

Next, we 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

Status Vector		S		Status Vector		T	
V_1	V_3	s_{pk}	Partition	V_4	V_7	t_{pk}	Partition
1	1	1, 2	S_1	1	1	1,2,3,4,5	T_1
1	0	3, 4	S_2	1	0	6	T_2
0	1	/	/	0	1	/	/
0	0	/	/	0	0	7, 8	T_3

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

is at most 2^m. 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

$$V_l^k \to V_r^k = \bigcup_{S_i \subseteq V_l^k} S_i \to \bigcup_{T_j \subseteq V_r^k} T_j = \bigcup_{S_i \subseteq V_l^k, T_j \subseteq V_r^k} S_i \to T_j \tag{2}$$

Based on Equation 2, we further formalize two populating rules for each pair of partitions S_i and T_j. The two rules require to use z primary keys in S_i for populating y foreign keys in T_j.

$$PF_{S_i \to T_j} = y \;\; s.t. \;\; y \in \{0, 1, \ldots, |T_j|\} \quad PF^{d}_{S_i \to T_j} = z \;\; s.t. \;\; z \in \{0, 1, \ldots, |S_i|\} \tag{3}$$

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_{jcc}^k and n_{jdc}^k denote the join cardinality and join distinct constraints on the k^{th} join view.

$$PF_{V_l^k \to V_r^k} = \sum_{S_i \subseteq V_l^k,\, T_j \subseteq V_r^k} PF_{S_i \to T_j} = n_{jcc}^k \quad PF^d_{V_l^k \to V_r^k} = \sum_{S_i \subseteq V_l^k,\, T_j \subseteq V_r^k} PF^d_{S_i \to T_j} = n_{jdc}^k$$
$$PF_{\overline{V_l^k} \to V_r^k} = \sum_{S_i \subseteq \overline{V_l^k},\, T_j \subseteq V_r^k} PF_{S_i \to T_j} = |V_r^k| - n_{jcc}^k \tag{4}$$

Note, for any partition S_i in V_l^k, if its primary keys are selected to populate foreign keys in more than one partition 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 $\sum_{S_i \subseteq V_l^k,\, T_j \subseteq V_r^k} PF^d_{S_i \to T_j} < n_{jdc}^k$. For example 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

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

Obviously, our populating rules should also make sure that the foreign keys in each partition should be exactly covered. That is, 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. By combining all of these populating rules in Equations 3 ~ 5, we can formalize the problem as a *Constraint Programming* (*CP*) problem [27]. 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. 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_r^k), i.e., $|S_i| \geq \sum_{T_j \subseteq V_r^k} PF^d_{S_i \to T_j}$

After integrating all the three constraints into the *CP* problem, we propose to use the *Or-Tools* [26] as our *CP solver.* It improves its efficiency by utilizing the constraint propagation method to prune the search space. Note that *CP* Problem may be unsolvable due to the non-key data distribution is different with the real one. If so, we keep multiple foreign key columns in the table, the number of which is the number of conflict distributions. And each *AQT* uses the corresponding column based on its distribution requirement.

Example 5.3. Consider the join constraints from two join views V_5 and V_8. If we include the three constraints described above, we would find a feasible solution without contradictions as follows.

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

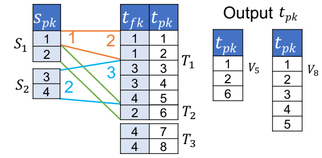


Figure 10: Unified Population for t_{fk}

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 to populate two foreign keys in T_1, e.g., {1, 1}. In addition, recall that the foreign keys in T_3 can be populated by arbitrary primary keys, we populate T_3 with 4.

5.3 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 use them to populate the foreign keys in another table that references T. Based on the observation, we propose to build a topological order among tables based on their *PK-FK* references [14], and populate foreign keys in each table according to the topological order. 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 [14]. 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

6.1 Discussion of 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.

6.2 Discussion of Time Complexity

In the section, we discuss the time complexity of *Mirage*. Suppose there exist $K+J$ literals in all the annotated query templates, among them K literals are unary predicates and J literals are arithmetic predicates. Moreover, the generated database is supposed to consist of M columns and L rows in total. For the non-key generator, the logical dependency decoupling process needs to assign boundary values to the parameters in literals, thus its time complexity is $O(K+J)$. For solving *UCC*s, let N be the total number of *UCC*s on a given non-key column A. Firstly, we sort each parameter p in $F_A(p)$, whose time complexity of is $O(NlogN)$. Secondly, for each parameter p in $f_A(p)$, we use a binary search to find the smallest range that can accommodate it and then the time complexity is $O(NlogN)$. Thirdly, the time complexity of assigning unique values and instantiating parameters is $O(N)$. Thus, the total time complexity of *UCC*s on a non-key column is $O(NlogN)$. Since there are K *UCC*s on all the non-key columns, the total time complexity is $O(KlogK)$. For solving *ACC*s, the time complexity of computing the result view of the i^{th} *ACC* is $O(g_i * L)$, where g_i represents the computation cost of the i^{th} *ACC*'s arithmetic function. The time complexity of selecting a certain number of items in the result view that satisfy the predicate is $O(L)$. Since there are J *ACC*s on all the non-key columns, the total time complexity of solving *ACC*s is $O(\sum_1^J g_i L)$. Lastly, the time complexity of generating L rows for at most M non-key columns is $O(L * M)$. Taken together, the total time complexity of generating non-key columns is $O(J + KlogK + L(\sum_{i=1}^J g_i + M))$.

For the key generator, the time complexity of computing the status vectors according to the outputs of selection operators is $O(L(K + \sum_1^J g_i))$. This is because the time complexity of computing the result of K UCCs over L rows is $O(K * L)$, and the time complexity of computing the result of J ACCs over L rows is $O(\sum_1^J g_i L)$. Therefore, the time complexity of computing the status vectors is $O(L(K + \sum_1^J g_i))$. As the *Constraint Programming (CP)* problem is NP-hard, its time complexity is unavailable. Finally, the time complexity of generating L rows for at most M foreign key columns based on the result of the CP solver is $O(L * M)$.

6.3 Discussion of Assumption And Conflict

We implicitly include two assumptions of data correlation in our paper.

(1) The result of a logical predicate is equivalent to the result of its kept sub-predicates after decoupling. The assumption is used to decouple logical cardinality constraints (LCC). Therefore, we can simplify the complex joint distribution requirements among columns from LCC and just deal with unary cardinality constraints (UCC) for each column independently.
(2) The parameter of UCC with comparator = ($UCC_=$) can be put into any appropriate range satisfying its data existence probability. The assumption is used to put the parameter of $UCC_=$. Since parameters in the annotated query template (AQT) are anonymized, we can not determine the original range that locates the parameter of the $UCC_=$. Based on the assumption, we just put the parameter into any appropriate range instead of the exact original range.

Next, we discuss the conflicts introduced by the two assumptions and discuss how to resolve them.

The first assumption may introduce conflicts when the non-key generator solves UCC. Specifically, if a cardinality constraint of $UCC_=$ is obtained after decoupling, it may cause a totally different data space requirement from the original one to the involved column. Therefore, we may not be able to find a feasible space for placing the parameter of the $UCC_=$ in the divided ranges of the column. To solve the conflict, we propose to divide the $UCC_=$ into multiple small-sized $UCC'_=$. Then it is easy to find ranges to locate the parameters in the small sized $UCC'_=$, which aggregates to satisfy the cardinality requirement in $UCC_=$. We revise the content in § **4.2** to highlight the solution. An example is provided in Example 6.2. After dividing, the comparator = of $UCC_=$ is converted to comparator *in*.

Example 6.2. Consider a table T containing two columns t_1 and t_2, its cardinality constraints are $|T| = 4$, $|T|_{t_1} = 3$ and $|T|_{t_2} = 3$. The following three cardinality constraints are from two queries over table T.

$$|V_1| = |T| = 4 \quad |V_2| = |\sigma_{t_1=p_1 \vee t_2>p_2}(T)| = 3 \quad |V_3| = |\sigma_{t_1=p_3}(T)| = 2$$

According to our decomposing algorithm in § 4.1, the second view V_2 can be decomposed into two unary views, i.e., $V_2^1 = \sigma_{t_1=p_1}(T)$ and $V_2^2 = \sigma_{t_2>p_2}(T) = V_2^1 \cup V_2^2$.

$$V_2 = \sigma_{t_1=p_1}(T) \cup \sigma_{t_2>p_2}(T) = V_2^1 \cup V_2^2$$

After decoupling, we can set $p_2 = \infty$ and then $V_2^2 = \emptyset$. Finally, we eliminate view V_2^2 and keep the following two unary cardinality constraints on column t_1.

$$|V_2^1| = |\sigma_{t_1=p_1}(T)| = 3 \qquad |V_3| = |\sigma_{t_1=p_3}(T)| = 2$$

We represent their corresponding CDF requests as $f(p_1) = \frac{3}{4} = 75\%$ and $f(p_3) = \frac{2}{4} = 50\%$. Since $f(p_1) + f(p_3) \geq 100\%$, we cannot locate the parameters in the two $UCC_=$ in domain range $(0, 3]$ simultaneously. Therefore, we divide $UCC_=$ of V_2^1 into two small sized $UCC'_=$, which introduces two parameters, and then we have $\sigma_{t_1 \ in \ (p'_1, p_4)}(T)$, and its cardinality constraint is $|V_2^1| = |\sigma_{t_1 \ in \ (p'_1, p_4)}(T)| = 3$. Therefore, we can obtain the following two CDF requests.

$$f(p_1) = f(p'_1) + f(p_4) = \frac{3}{4} = 75\% \quad f(p_3) = \frac{2}{4} = 50\%$$

If the generated data is 1,2,3 for column t_1, a feasible instantiation is $p'_1 = p_3 = 1$ and $p_4 = 2$, which divide the space into three ranges, i.e., $(0, p'_1 = p_3]$, $(p'_1 = p_3, p_4]$, $(p_4, 3]$. The corresponding CDF is $f(1) = 50\%$, $f(2) = 25\%$ and $f(3) = 25\%$.

The second assumption may introduce conflicts when Key Generator merges multiple distribution requirements to a single foreign key column. Specifically, the Constraint Programming Problem described in § **5.2** may be unsolvable due to the conflict, that is we may not be able to merge the distribution requirements into one column. An example is provided in Example 6.3. In such a case, we will split one foreign key column into multiple foreign key columns in a table, the number of which is the number of conflict distributions (can not be merged together), and each AQT uses the corresponding column generated based on its distribution requirement. We add the discussion in § **5.2** to highlight the solution.

Example 6.3. 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}. The original data in D is presented in Fig. 12a with the cardinality constraints $|S| = 2$, $|T| = 1$, $|S|_{s_1} = 2$. The example AQTs are illustrated in Fig. 11. Noted that if we set p_1 =3 and p_2=3 based on the original data in Fig. 12a, all cardinality constraints in AQTs hold, so there is a solution for this QAGen problem.

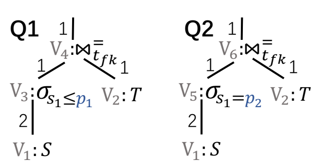


Figure 11: Annotated Query Templates for Table S and T

s_{pk}	s_1
1	3
2	7

t_{pk}	t_{fk}
1	1

(a) Original Data

s_{pk}	s_1
1	1
2	2

t_{pk}	t_{fk}
1	?

(b) Generated Data Without t_{fk}

Figure 12: Data of Table S and T

However, considering the way that we deal with the two UCCs of the non-key column s_1, i.e., $|\sigma_{s_1<p_1}(S)| = 1$ (for V_3) and $|\sigma_{s_1=p_2}(S)| = 1$ (for V_5), we infer their CDF requests are $F(p_1) = 50\%$ and $f(p_2) = 50\%$. Suppose the generated data for s_1 is 1 and 2 as in Fig. 12b. First, since the cardinality of s_1 is 2, we initialize the domain range of s_1 as $(0,2]$, and divide the range into $(0, p_1]$ and $(p_1, 2]$ by $F(p_1) = 50\%$. Then we have two CDFs for s_1 divided by p_1 as $F(0, p_1) = 50\%$ and $F(p_1, 2) = 50\%$. Note that p_2 can be put into either of the two ranges. However, if we put p_2 into range $(p_1, 2]$, we have $p_1 = 1$ and $p_2 = 2$, which cause a conflict when populating t_{fk}. In the case when p_1=1 and p_2=2, V_3 and V_5 output the primary keys valued 1 and 2 respectively. Correspondingly, to satisfy the two $\bowtie_{t_{fk}=}$ for V_4 and V_6, we must have two foreign keys valued 1 and 2 respectively.

But we can only populate one t_{fk} in table T. Then we can not find a solution to satisfy the cardinality constraints in the two *AQT*s.

In such a case, the generated table T splits the column t_{fk} into two foreign key columns, i.e., t_{fk1} and t_{fk2} as in Fig. 14. We instantiate each *AQT* based on the data from the individual foreign key column as in Fig 13.

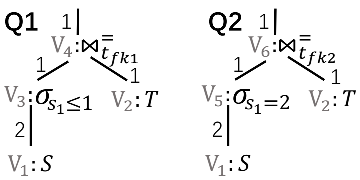


Figure 13: Populated Query with Different Foreign Key Columns

s_{pk}	s_1
1	1
2	2

t_{pk}	t_{fk1}	t_{fk2}
1	1	2

Figure 14: Generated Data

Notice that these conflicts are highly related to the associations between multiple cardinality constraints. For the test workloads, our experiments show that 1) all the cardinality requests from $UCC_=$ are well satisfied without dividing any $UCC_=$ into smaller size $UCC'_=$; 2) all the populations of foreign key columns accomplish the merge, i.e., no foreign key column splitting introduced. Though we have solutions for the conflicts introduced by the two assumptions, we have not met the conflicts in our experimental benchmark scenarios.

7 EXTENSION

7.1 Extension to Residual Predicate

There are three general cases where the query optimizer would not push down the residual predicate filter entirely to the base table:

- The residual predicate contains a logical predicate that operates on multiple tables, e.g., the residual predicate in TPC-H q19.
- The residual predicate contains an arithmetic predicate that operates on multiple tables.
- The residual predicate operates on one table and the query optimizer can push down it entirely. However, it is more cost-effective to execute the join before selection.

Logical predicate crosses tables. For simplicity, we mainly discuss the case in which the residual predicate follows the disjunctive normal form (DNF). Note that, a predicate with any other form can be transformed to DNF [28]. That is,

$$(LeftTableClause_1 \wedge RightTableClause_1) \vee ...$$
$$\vee(LeftTableClause_m \wedge RightTableClause_m).$$

More specifically, the query optimizer would first push down the union of predicate filters $LeftTableClause_1 \vee ... \vee LeftTableClause_m$ and $RightTableClause_1 \vee ... \vee RightTableClause_m$ to the left and right tables, respectively. Next, our proposed decoupling dependency procedure would try to eliminate $m-1$ sub-predicates and keep only one sub-predicate in both of the pushed-down predicate filters. As described above, if we guarantee that the two kept sub-predicates belong to the same sub-predicate of the residual predicate (i.e., $LeftTableClause_i \wedge RightTableClause_i$), then the sub-predicate $LeftTableClause_i \wedge RightTableClause_i$ would always hold. Moreover, as the residual predicate is a union of multiple sub-predicates, then it would also always hold.

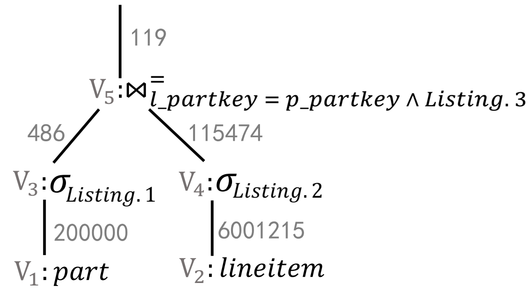


Figure 15: The annotated query template of TPC-H q19 in PostgreSQL, whose residual predicate *listing* 3 is usually applied alongside the join operator during the execution.

Below, we will specifically outline the process. Consider the union of predicates pushed down to the left table, i.e., $LeftTableClause_1 \vee ... \vee LeftTableClause_m$. Its corresponding view is as follows.

$$\sigma_{LeftTableClause_1 \vee ... \vee LeftTableClause_m}(LeftTable) =$$
$$\sigma_{LeftTableClause_1}(LeftTable) \cup ... \cup \sigma_{LeftTableClause_m}(LeftTable)$$

In the first decoupling phase, we find all sub-selection views that can be instantiated to be $\emptyset$. If all the selection views can be set as $\emptyset$, we keep all the sub-selection views; otherwise, we only keep the sub-selection views that can not be set as $\emptyset$. Suppose we finally keep $n^l(\le m)$ sub-selection views based on the predicates on the left table, i.e., V^l. We process the predicates on the right table similarly and keep $n^r(\le m)$ sub-selection views, i.e., V^r.

If there are two sub-selection views in V^l and V^r generated from the same i-th pair of clauses in the residual predicate, we keep only these two clauses in the decoupling clauses and then the $LeftTableClause_i \wedge RightTableClause_i$ always holds, then reset the other sub-selection views to $\emptyset$. Finally, the problem can be simplified to be processed by Mirage.

If there are no sub-selection views in both V^l and V^r generated from the same i-th pair of predicates, the residual predicate does not always hold. Then Mirage cannot deal with this situation. However, this situation may only happen when $n^l + n^r \le m$ according to the pigeonhole principle, then we further infer that $n^l < m$ and $n^r < m$. In other words, each kept clause cannot be set to $\emptyset$. Based on the Table. 3 in § 4.1, the comparators of all literals in all n^l (resp. n^r) clauses must strictly belong to $\neq$, *not like*, *not in*. As far as we know, the situation does not exist in current OLAP benchmarks, e.g., TPC-H, and public workloads, e.g., Public BI benchmark.

In summary, Mirage always supports residual predicates as long as either of the decoupling result does not contain $\neq$, *not like*, *not in*.

Example 7.1. As shown in Fig. 15, the annotated query template for TPC-H q19 contains a residual predicate *listing* 3 which operates on both tables *part* and *lineitem*. Thus, *listing* 3 could not be directly pushed down to the two base tables. Note that the residual predicate *listing* 3 is usually applied alongside the join operator during the execution. To reduce the join cost, the query optimizer would apply the union of predicate filters in *listing* 3 to each table, which are *listing* 1 and 2. For this case, as *listing* 3 imposes dependencies between multiple non-key columns from two tables, our query rewriting method also could not push down the selection without breaking the cardinality constraints in the *AQT*. Fortunately, our proposed sub-selection view elimination method (see

Listing 1: the union of predicates on the table *part*

```
( p_brand = p1 AND p_size >= p2 AND p_size <= p3
    AND
  p_container IN (p4, p5, p6, p7) )    --PartClause1
OR
( p_brand = p8 AND p_size >= p9 AND p_size <= p10
    AND
  p_container IN (p11, p12, p13, p14) ) --PartClause2
OR
( p_brand = p15 AND p_size >= p16 AND p_size <= p17
    AND
  p_container IN (p18, p19, p20, p21) ) --PartClause3
```

Listing 2: the union of predicates on the table *lineitem*

```
( l_shipmode IN (p22, p23) AND l_shipinstruct = p24
    AND
  l_quantity >= p25 AND l_quantity <= p26 )
      --LineitemClause1
OR
( l_shipmode IN (p27, p28) AND l_shipinstruct = p29
  AND l_quantity >= p30 AND l_quantity <= p31 )
      --LineitemClause2
OR
( l_shipmode IN (p32,p33) AND l_shipinstruct = p34
  AND l_quantity >= p35 AND l_quantity <= p36 )
      --LineitemClause3
```

Listing 3: the residual logical predicate on *part* and *lineitem*

```
(PartClause1 AND LineitemClause1) OR
(PartClause2 AND LineitemClause2) OR
(PartClause3 AND LineitemClause3)
```

Section 4.1) could effectively decouple the dependencies between non-key columns. This further helps us simplify the complexity of dealing with the residual predicate filter.

More specifically, the logical predicate in Fig. 1 follows the disjunctive normal form (*DNF*), which is $PartClause1 \vee PartClause2 \vee PartClause3$. As shown in Table 3 in our paper, we can assign proper boundary values to the parameters of some sub-predicates such that we can keep only one sub-predicate and eliminate the others. For example, by setting $p_{10} = -\infty$ and $p_{17} = -\infty$, we can make $\sigma_{PartClause2}(part) = \sigma_{PartClause3}(part) = \emptyset$. Then, we can eliminate *PartClause*2 and *PartClause*3 and reduce *listing* 1 to *PartClause*1. Similarly, we can also reduce *listing* 2 to *LineitemClause*1 by setting $p_{31} = -\infty$ and $p_{36} = -\infty$. That is, after finishing the procedure of decoupling dependencies, all the left (*resp.* right) child input rows of V_5 in Fig. 15 would satisfy *PartClause*1 in the table *part* (*resp. LineitemClause*1 in the table *lineitem*). Then, the sub-predicate *PartClause*1 *AND LineitemClause*1 of the residual predicate *listing* 3 (see Figs. 15 and 3) would always hold. With this mechanism, we do not need to take into account the residual predicate when generating the foreign keys of table *lineitem* to guarantee the output size of the join operator (see V_5). Note that, the foreign key index could not be leveraged to improve the performance of join as its inputs are intermediate execution results, and the join condition computing cost is determined by the number of input primary keys and foreign keys. Fortunately, our *Mirage* could guarantee the input size of the join operator.

Arithmetic predicate crosses tables. For the arithmetic predicate, Mirage cannot push down to any single table. Therefore, Mirage tries to eliminate the predicate like dealing with logical predicate across multiple tables. Based on Table. 3 in § **4.1**, Mirage can eliminate the arithmetic predicate. For example, the residual predicate is an arithmetic predicate in the following *AQT*. We can set $p_2 = -\infty$ to eliminate $s_1 + t_1 > p_2$.

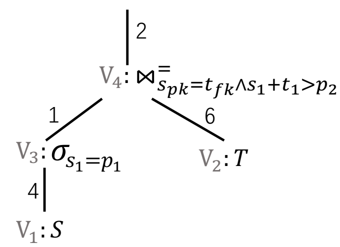


Figure 16: The *AQT* with a residual arithmetic predicate

Select After Join Because of Lower Cost. Consider the logical query plan $\sigma_{P_T}(\sigma_{P_S}(S) \bowtie^{=}_{t_{fk}} T)$, where predicate P_S (resp. P_T) only involves table S (resp. T) and P_T is executed after join. It commonly occurs in nested loop join using foreign key index. Specifically, if P_S filters a larger number of rows of S, it is cheaper to execute the nested loop join using the foreign key index over t_{fk} firstly and then apply P_T to the join result. In such a query plan, the size of foreign key index scan is equal to the size of matched rows, so we need to deal with three cardinality constraints as follows.

$$|\sigma_{P_S}(S)| = n_1 \quad |\sigma_{P_S}(S) \bowtie^{=}_{t_{fk}} T| = n_2 \quad |\sigma_{P_T}(\sigma_{P_S}(S) \bowtie^{=}_{t_{fk}} T)| = n_3$$

Mirage supports the first two cardinality constraints. For supporting the last one, *Mirage* replaces it with an equivalent transformation, i.e., $|\sigma_{P_S}(S) \bowtie^{=}_{t_{fk}} \sigma_{P_T}(T)| = n_3$. However, the transformation introduces an extra cardinality constraint for $\sigma_{P_T}(T)$, i.e., $|\sigma_{P_T}(T)| = n_4$. If customers allow, *Mirage* obtains n_4 by executing $\sigma_P(T)$ in the original database. Otherwise, we are inspired by the join predicate independence assumption from traditional query optimizer [30] and assume that P_T filters the same proportion of rows on the table T as on the join result. Then *Mirage* calculates n_4 by the following equation.

$$\frac{|\sigma_{P_T}(T)|}{|T|} = \frac{|\sigma_{P_T}(\sigma_{P_S}(S) \bowtie^{=}_{t_{fk}} T)|}{|\sigma_{P_S}(S) \bowtie^{=}_{t_{fk}} T|} = \frac{n_3}{n_2} \Rightarrow n_4 = |\sigma_{P_T}(T)| = \frac{n_3}{n_2}|T|$$

7.2 Extension to Subquery

Mirage can be extended to support the nested subqueries. Usually, DBMS provides ways to eliminate subqueries for efficiency [3] and then in most of the *AQT*s, we see no subqueries. Here we illustrate the solution for the subqueries that cannot be eliminated by running TPC-H on PostgreSQL in Example [3].

Example 7.2. There are 13 queries with subqueries in TPC-H, i.e., Q_2, Q_4, Q_7, Q_8, Q_9, Q_{11}, Q_{13}, Q_{16}, Q_{17}, Q_{18}, Q_{20}, Q_{21}, Q_{22}. PostgreSQL eliminates subqueries for 7 of them, i.e., Q_4, Q_7, Q_8, Q_9, Q_{13},

Q_{18}, Q_{21}. But there are still 6 queries having subqueries in their query plans, i.e., Q_2, Q_{11}, Q_{16}, Q_{17}, Q_{20} and Q_{22}.

We classify the subqueries that cannot be eliminated into the following two cases.

(1) An uncorrelated subquery returning a single row, e.g., Listing 4 simplified from Q_{11}. For this case, database first executes the sub-query independently and saves its result as a temporary variable. Then the database replaces the sub-query in its parent query with the temporary variable and executes its parent query, e.g., Q_{11} and Q_{22}.

Listing 4: An uncorrelated subquery returning a single row

```
select * from T where t1 >
        (select avg(t1) as t_avg from T)
```

(2) There is no unnesting policy that can be applied to eliminate the subquery in current database, e.g., *line 2* of Listing [7] simplified from Q_{16}. Note that in this case, MySQL can eliminate the subquery with antijoin, and the result is then an unnested query plan, e.g., *line 4* of Listing [7]. However, PostgreSQL cannot eliminate it due to the lack of left anti join operator. Given the lack of unnesting policies in PostgreSQL, Q_2, Q_{16}, Q_{17} and Q_{20} have subqueries in their *AQTs*.

Listing 5: An uncorrelated subquery can be unnested

```
1 -- An uncorrelated subquery
2 select * from T where t_fk not in
3         (select s_pk from S where s1 > 1)
4 -- The unnested query by antijoin
5 select * from T antijoin S on t_fk=s_pk where s1> 1
```

For the first case, we parameterize the result of the subquery as a parameter p in the parameter query, and an example is provided in Example 7.3. Then we can deal with the subquery and the parent query as two separate *AQTs*. But after dealing with the two *AQTs*, the instantiated value p of the parent query may be different from the execution result r of the subquery on the generated data. Then we take an adjustment value $\delta(=p-r)$ in the subquery to make the result of the subquery consistent with the instantiated parameter value p of the parent query. The policy is suitable to Q_{11} and Q_{22} in TPC-H.

Example 7.3. Consider a table T with only a non-key column t_1. There are three rows in the table, i.e., 1, 5, 6. The query in Listing 4 contains an uncorrelated subquery returning a single row (an aggregated value). Usually, the database system executes *select avg*(t_1) *as* t_{avg} *from t* firstly to get t_{avg}, i.e., 4, and then executes *select* $*$ *from t where* $t_1 > 4$ $(=t_{avg})$ to get the results, i.e., 5, 6.

After parameterizing the returned result of the subquery as p_1 in its parent query, the original query can be divided into two independent queries, which are 'select avg(t_1) as t_{avg} from T' and 'select * from T where $t_1 > p_1$'. There are only one cardinality constraint in the *AQT* of the parent query, i.e, $|\sigma_{t_1>p_1}(T)| = 2$. After dealing with the constraint, if *Mirage* populates data 1, 2, 3 for table T, it instantiates parameter p_1=1 to guarantee $|\sigma_{t_1>p_1}(T)| = 2$ on the generated data. However, the execution result of the subquery on the generated data is t_{avg}=2. We now have $t_{avg} \neq p_1$. Then, we initiate an adjustment value δ=$p_1 - t_{avg}$ = 1-2 = -1 for the subquery by revising it as '*select avg*$(t_1) + \delta$ *as* t_{avg} *from t*'. Finally, we can combine the two instantiated *AQTs* to generate an equivalent new revised query as following.

```
select * from T where t1 >
        (select avg(t1) -1 as t_avg from T)
```

For the second case, the subquery can not be eliminated by its unnesting policy, which may be uncorrelated or correlated with the parent query. We present the solutions for them separately as follows.

Uncorrelated subquery. The example is shown in *line 2* of Listing 5 which is simplified from Q_{16} in TPC-H. We can unnest its subquery with an antijoin operator (query rewriting in *line 4* of Listing 5), which is taken by MySQL [7]. In *Mirage*, before processing the *AQT* containing uncorrelated subquery, it merges the plan of the uncorrelated subquery into its parent query plan and then deals with the merged *AQT* without a subquery. *Mirage* can use the existed query rewriting rules [8] to launch the merge. For example, any subquery of the form '*IN (select * from * where *)*' or '*EXISTS (select * from * where *)*' is transformed into a left semi join; any negation subquery of the form '*NOT IN (select * from * where *)*' or '*NOT EXISTS (select * from * where *)*' is transformed into a left anti join, e.g., the *antijoin* in MySQL [7]. The policy can be applied to Q_{16} in TPC-H.

Correlated subquery. An example is shown in Listing 6, which is simplified from Q_2 in TPC-H. It is related to three tables, i.e., *part*, *lineitem*, *supplier*.

Its physical plan from PostgreSQL is visualized in Fig. 17. In the plan, It is divided into three execution stages. Firstly, PostgreSQL executes *Hash Join* between *part* and *lineitem*, which is the *plan before subquery*. Secondly, PostgreSQL invokes the correlated subquery as an extra filter for each output *p_partkey* from the first step hash join, which is the *plan of subquery*. Finally, PostgreSQL executes another Hash Join between the result of the subquery and *supplier*, which is *plan after subquery*. Obviously, *Mirage* can guarantee the cardinality constraints in the first stage. The difficulty is to ensure that the execution cost of the second stage is similar to the original query, and the cardinality from the second stage is the same with the original query so as to further guarantee the cost of the third stage.

Listing 6: A simplified query from Q_2 in TPC-H

```
select *
from part join lineitem on p_partkey = l_partkey
          join supplier on s_suppkey = l_suppkey
where p_size = 2
  and l_quantity <= (select min(l_quantity) as a
                     from lineitem
                     where l_partkey = p_partkey)
  and s_acctbal < 2000;
```

To guarantee similar execution cost in the second stage of the subquery. Note that the scan in the correlated subquery is always the index-scan because of that its input from the first stage usually involves only a few rows. Therefore, each invocation to process the subquery is similar in cost. Since Mirage can guarantee the cardinality constraints in the first stage, the number of invocations to process the correlated subquery on the generated database

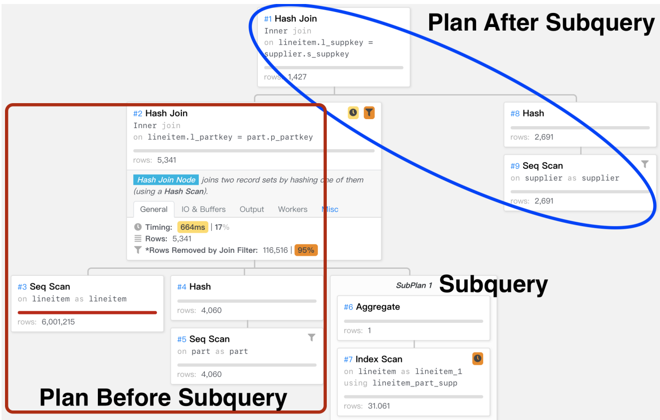


Figure 17: Query plan of Listing 6

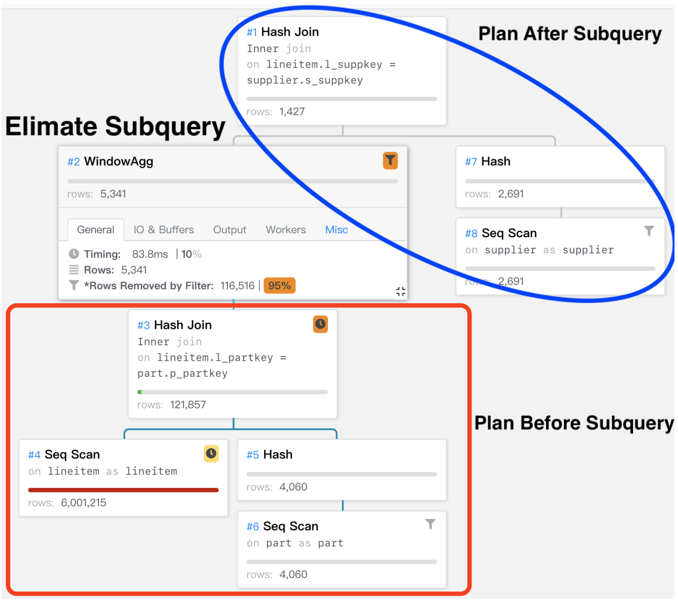


Figure 18: Query plan without subquery of Listing 3

can always be the same with the original one. Finally, the total execution cost of the second stage is similar to the original execution on the real database, which is only related to the cardinality from the first stage processing.

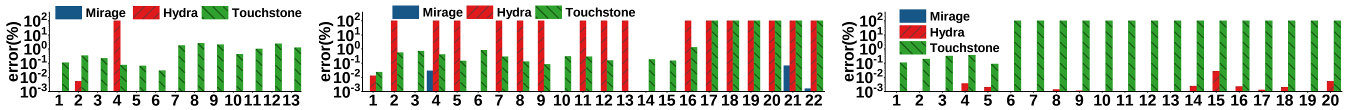
To guarantee the same output cardinality from the second stage. Since the framework of *Mirage* cannot support the subquery, *Mirage* then cannot control the cardinality from the second stage (i.e., from the subquery). Therefore, before dealing with the *AQT* with the correlated subquery, *Mirage* takes the way to replace the plan of the subquery with a semantically equivalent plan (as well as the equivalent cardinality constraint) without subquery. Note that the parameters in the re-written equivalent plan (in Fig. 18) for the subquery in the original real plan (in Fig. 17) always have the one-to-one correspondence. Notice that, *Mirage* can support the data generation and parameter instantiation on the re-written query plan. By taking the semantically equivalent plan replacement, *Mirage* guarantees the same cardinality from the second stage. This kind of subquery rewriting has been widely studied [9, 40]. *Mirage* achieves the replacement with the help of the well-studied query rewriting technique. For example, we replace the plan of the subquery in Fig. 17 with a WindowAgg operator [40], which is presented in Fig. 18. *Mirage* supports WindowAgg because it is a physical operator of projection. Note that all the cardinality constraints of the *plan after subquery* can be well supported by *Mirage*. Since its input cardinality from the subquery is the same with the original one, we then achieve an accurate simulation for the query with correlated subquery. In TPC-H, we take this way to support Q_2, Q_{17} (by rule in work [40]) and Q_{20} (by rule in work [9]).

In summary, *Mirage* can support the generation with subqueries based on some well-studied rewriting policies. We can then transform and eliminate the subquery into an equivalent query plan without subquery, which can be successfully processed by *Mirage* but still guarantees the cardinality constraints in the original query plan.

8 RELATED WORK

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

Data-aware generators [1, 5, 13, 31, 34] 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* [34] 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 [15]. **Query-aware generators** [2, 4, 11, 15–17, 21, 29] 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* [4] is the first work for query aware database generation. It has powerful support in 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* [17] 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* [29] 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. However, Touchstone cannot sample for the cardinality constraints from arbitrary logical predicates or support semi-join, anti-join. *DCGen* [2] and *Hydra* [29] 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. *Loki* [21] works well only for solving the selection cardinality constraints. Compared to these works, *Mirage* can give the most powerful support to complex operators and guarantee simulation error to be *zero* theoretically.



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

Figure 19: Relative Errors for Various Workload including SSB, TPC-H, and TPC-DS (100% error means no support)

9 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 three most recent work, i.e., *Touchstone* [15] *Hydra* [29] and *Loki* [21] with their provided codes. We select one real-world benchmark, i.e., Public BI Benchmark [37], which has all queries on single tables, and three classic benchmark scenarios, i.e., SSB [25], TPC-H [36] and TPC-DS [35], to play as our simulation applications. SSB is a benchmark with 13 simple structured queries. TPC-H is the most popular OLAP benchmark with 22 queries having complex operators, e.g., various joins. In order to test the scalability of workload, *Hydra* produces a scenario with large-scale workloads from TPC-DS, but removes its unsupported complex operators, e.g., *arithmetic* selections. For a fair comparison, we take all 100 distinct queries. Since *Loki* aims to solve the selection cardinality constraints from a single table, Public BI benchmark is selected for comparison instead of the other three benchmarks with table joins. Public BI provides to download all the data and workload [37], and we take the official generation tools designed for the other three benchmarks [23, 35, 36] to compose the input applications. 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, a large default SF=200 is used.

Metrics: We adopt *relative error* $= \frac{\sum ||V_i|-|\hat{V}_i||}{\sum |V_i|}$ [15] 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: We take the sampling-based parameter instantiation for ACC. Its default sampling size is 4 million (4M) rows with theoretical error bound 0.1% in a confidence level 99.9% according to Hoeffding's Inequality. In order to control memory usage, 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. Note that we use GN, CS, CP, and PF to represent the methods of generating non-key columns, computing status vectors, constraint programming solving, and populating foreign keys.

9.1 Comparing with Existing Methods

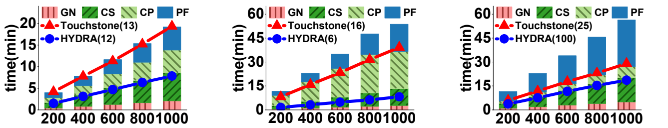
We first compare the complex workload support ability as well as the simulation fidelity with the state-of-the-art work, i.e., *Touchstone* and *Hydra* in §9.1.1; we then compare their generation efficiency in §9.1.2; finally, we introduce and compare the performance of *Loki* with all other work on the real application Public BI in §9.1.3.

9.1.1 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. 19. 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. 19a). *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. 19b). *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 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. 19c). *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



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

Figure 20: Comparison of Generation Efficiency

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.

9.1.2 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. 20 by changing *SF* from 200 to 1000 for all scenarios. Note that each tool only generates the application with its supported queries (size labeled on figures).

For a fair comparison, we only analyze the results with the same workload size. Specifically, for SSB, *Mirage* is as fast as *Touchstone*; for TPC-DS, *Mirage* is 2× slower than *Hydra*. For the other situation, though *Mirage* is slower, it supports more queries in generation. So let us study the reason why *Mirage* is slower than *Hydra* on TPC-DS (in Fig. 20c). *Hydra* resolves all cardinality constraints as a linear programming problem (*LP*) and generates a small-sized initial dataset, which is simply duplicated for a large database. Therefore, it does not introduce new computation cost in duplicating, but its generated dataset has an extremely high duplication ratio. However, in order to ensure the diversity of dataset, *Mirage* calculates *CS*, *CP*, and *PF* for each round of data population, which causes the slower generation speed. To reduce the time of population, *Mirage* can take a similar way to generate a small dataset, and then expand it to a large *SF* by duplication. Take TPC-DS for example. *Mirage* is even faster (14s) than *Hydra* (32s) when generating the data of *SF*=1, because *Mirage* only puts the join constraints into a *CP* problem, but *Hydra* models all the constraints into an *LP* problem.

In summary, *Mirage* **guarantees to generate data instead of duplicating** from a small-sized data. It is more practical.

9.1.3 Comparison on Real Application Public BI. Since *Loki* is designed to resolve selection constraints, Public BI is a suitable application with all queries on single tables. Even for the single table query in Public BI, since *Touchstone* cannot support complex logical predicates and *Hydra* cannot support the comparisons on the string column, we finally select 150 queries supported by all the tools. To plot the result in Fig. 22a clearly, we randomly divide the queries into 5 groups, each of which covers 30 queries. Since *Mirage* and *Hydra* have been proved to have no simulation errors theoretically, their relative errors are ≤ 0.01%. *Touchstone* samples randomly to select the data distribution with the least simulation deviation, which cannot guarantee accuracy, and its relative error is ≥ 50% in the 4^{th} and 5^{th} groups. *Loki* also constructs the cardinality constraint as a *CP* problem as in *Mirage* by computing the value for each row, which has an exponential computation cost for a large table. To reduce the computation cost, *Loki* shrinks both the table size and each cardinality constraint related to the table following the same shrinkage factor to generate a small table and then replicates it for the original big table. It then causes the errors, i.e., 1.05% for *Loki*. Since Public BI has the most tables with various data distributions, we also plot the generation efficiency of *Mirage* for the related 47 tables in Fig. 22b, with each point in the figure representing a table and the largest table sized 20*GB*. The generation efficiency of *Mirage* is almost linearly related to the table size and the total time to generate all the tables (410*GB*) is 25.6 min.

In summary, *Mirage* performs well to simulate a real application scenario **in a linear way** even for a large number of tables.

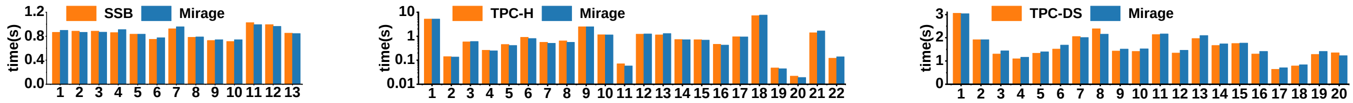
9.2 Technical Design of *Mirage*

We now run experiments to expose the effectiveness of technical designs in *Mirage* by the complex benchmark scenarios.

9.2.1 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. 21. The mean deviations of latencies for SSB, TPC-H, and TPC-DS are all less than 6%.

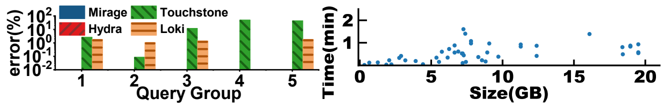
In summary, *Mirage* guarantees **a high fidelity** on simulating complex application scenarios.

9.2.2 Generation Scalability. We demonstrate the generation scalability of *Mirage* by varying the numbers of query instances, query templates, and involved join tables. We use TPC-H as the test workload and generate data with *SF*=200. To evaluate the efficiency of *Mirage* under different numbers of query templates, we add query templates in TPC-H from 4 to 22 gradually and use 5 query instances for each query template (in Fig. 23a). To evaluate the efficiency of *Mirage* under different numbers of query instances per template, we vary the number of query instances per template in TPC-H from 1 to 5 (in Fig. 23b). To evaluate the efficiency of *Mirage* with different numbers of join tables, we divide the 22 Queries in TPC-H into 7 groups based on the join table accessing relationship. Specifically, queries in i^{th} group can only join with the tables involved in $1^{st} \sim (i-1)^{th}$ groups. We use TPC-H as the test workload and generate data with *SF*=200. We present the tables involved in query templates in Table 4. The query templates from left column to right column in Table 4 have a partial order relationship based on the



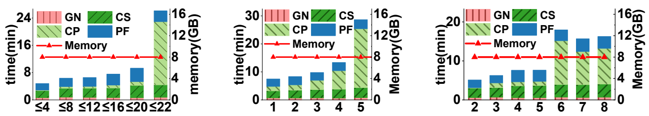
(a) Query of SSB (b) Query of TPC-H (c) Query Group of TPC-DS

Figure 21: Comparison of Query Latency for Various Workload



(a) Relative Errors (b) Generation Efficiency

Figure 22: **Performance on Public BI Benchmark**



(a) Varying Query Templates (b) Varying Query Instances (c) Varying the Number of Join Tables

Figure 23: **Generation Scalability of *Mirage* Based on TPC-H**

involved tables. That is any query template of the current column cannot involve any table right to it. For example, Q_{14} involves table *Part* and *Lineitem*.

Table 4: The queries and involved tables

Tables	Orders, Lineitem	Customer	Part	Nation	Supplier	Partsupp	Region
Queries	$Q_{1,4,6,12}$	$Q_{3,13,18,22}$	$Q_{14,17,19}$	Q_{10}	$Q_{7,15,21}$	$Q_{9,11,16,20}$	$Q_{2,5,8}$

Based on Table 4, we construct seven groups of workloads. In the i-th group, we randomly sample 100 query templates in the first i columns and then instantiate them, parameters of which are populated by the official tool [1] and different with each other. Therefore, the queries in the (i+1)-th group involve more join tables than those in the i-th group. In the i^{th} group, we randomly sample 100 query templates, which involve joins with at most i tables and the result is shown in Fig. 23c. Note, as we observed that the total time of generating non-key columns is relatively small, we do not show the running time of each step in the non-key generator. We also observe that *GN*, *CS*, and *PF* scale well with the increasing number of query templates and query instances per template. As the *CP* problem is NP-Hard, we observed that the time of *CP* increases super-linearly. In Fig. 23c, when the join involves more than 5 tables, the time of *CP* increases suddenly. Because more query templates will introduce exponential growth of the status vectors of *l_orderkey*, i.e., the sharp increase of parameters in the *CP* solver. Furthermore, we can see that the memory consumption is quite stable, because of the batch-based data generation strategy to control the memory usage.

In summary, *Mirage* is **linearly scalable** with *GN*, *CS* and *PF*; *CP* is NP-Hard, which is **super-linearly scalable**.

10 CONCLUSION AND FUTURE WORK

In this paper, we propose *Mirage*, a query-aware data generator (QAG) that provides the most powerful support to complex OLAP workloads. 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 to the related work, we can totally dominate them in operator support ability and simulation fidelity. However, *Mirage* has not covered all operators, e.g., non-PK-FK join, which will be left as our future work.

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

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] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zait, and Chun-Chieh Lin. 2009. Enhanced subquery optimizations in oracle. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1366–1377.

[4] 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

[5] 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.

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

[7] MySQL Documentation. 2023. *MySQL 8.0 Anti-Join*. https://dev.mysql.com/blog-archive/antijoin-in-mysql-8/

[8] MySQL Documentation. 2023. *MySQL 8.0 Reference Manual: Semi-Joins*. https://dev.mysql.com/doc/refman/8.0/en/semijoins.html

[9] Philipp Fent, Altan Birler, and Thomas Neumann. 2022. Practical planning and execution of groupjoin and nested aggregates. *The VLDB Journal* (2022), 1–26.

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

[11] 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

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

[13] 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.

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

[15] 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

[16] 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

[17] 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

[18] Stephen Macke, Maryam Aliakbarpour, Ilias Diakonikolas, Aditya Parameswaran, and Ronitt Rubinfeld. 2021. Rapid approximate aggregation with distribution-sensitive interval guarantees. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1703–1714.

[19] MariaDB. 2014. *MariaDB performance regression due to unnecessary index in HashJoin*. https://jira.mariadb.org/browse/MDEV-6888?jql=labels%20%3D%20hash-join

[20] 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

[21] Parimarjan Negi, Laurent Bindschaedler, Mohammad Alizadeh, Tim Kraska, Jyoti Leeka, Anja Gruenheid, and Matteo Interlandi. 2023. Unshackling Database Benchmarking from Synthetic Workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3659–3662.

[22] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

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

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

[25] 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.

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

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

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

[29] 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

[30] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[31] 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.

[32] 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

[33] Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. Lightsaber: Efficient window aggregation on multi-core processors. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2505–2521.

[34] 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.

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

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

[37] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get real: How benchmarks fail to represent the real world. In *Proceedings of the Workshop on Testing Database Systems*. 1–6.

[38] Fang Wang, Xiao Yan, Man Lung Yiu, Shuai LI, Zunyao Mao, and Bo Tang. 2023. Speeding Up End-to-end Query Execution via Learning-based Progressive Cardinality Estimation. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[39] Jintao Zhang, Chao Zhang, Guoliang Li, and Chengliang Chai. 2023. AutoCE: An Accurate and Efficient Model Advisor for Learned Cardinality Estimation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2621–2633.

[40] Calisto Zuzarte, Hamid Pirahesh, Wenbin Ma, Qi Cheng, Linqi Liu, and Kwai Wong. 2003. Winmagic: Subquery elimination using window aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 652–656.