Mirage: Generating Enormous Databases for Complex Workloads

Qingshuai Wang Hao Li Zirui Hu Rong Zhang* Chengcheng Yang* Peng Cai Xuan Zhou Aoying Zhou
East China Normal University
{qswang@stu, haoli@stu, zrhu@stu, rzhang@dase, ccyang@dase, pcai@dase, xzhou@dase, ayzhou@dase}.ecnu.edu.cn

Abstract—To optimize query parallelism techniques, substantial workloads are required with specific query plans and customized output size for each column’s domain, and further models the generation problem as a classic bin packing problem. Finally, requirements within each column's domain, and further models the generation errors. In this paper, we design a new generator Mirage supporting well for complex operators with low error bounds for cardinality constraints. First, Mirage leverages Query Rewriting and Set Transforming Rules to decouple dependencies between key and non-key columns, which could help generate each of them individually. Then, for the non-key columns, Mirage abstracts cardinality constraints of operators as placement requirements within each column’s domain, and further models the generation problem as a classic bin packing problem. Finally, for the key columns, Mirage proposes a uniform representation of join cardinality constraints for all types of PK-FK joins and partitions the data according to the matching status between PK and FK columns. Then, it formulates the key population as a Constraint Programming problem, which can be solved by an existing CP Solver. The experiments show that Mirage conquers all previous work in either operator support or generation error.

Index Terms—query-aware database generator, performance evaluation, benchmarking, query optimization

I. INTRODUCTION

Modern query execution engine takes the inter- and intra-query parallelisms [1] to schedule queries among cluster nodes and operator execution within a single query. And the overall performance of the query engine is highly sensitive to the inter-query scheduling policies, and intra-query parallelism optimizations [2]–[8], etc. To conduct comprehensive evaluation and optimization of these critical algorithms, developers would like to generate substantial workloads by customizing the query plans and the output size of each operator. Specifically,

1) Inter-query Scheduling Policies. When scheduling queries among multiple nodes, the optimizer would place queries requiring different types of resources in the same node such that the resource contentions can be effectively avoided and the resource utilization is maximized. Current work leverages machine learning to learn a generalized schedule solution [4], [8] which works well under various workloads. However, it requires a number of workloads with diverse query plans, operators, and output sizes of operators for training.

2) Intra-query Parallelism Optimization. The optimization task divides a query plan into multiple parts, which facilitate concurrent execution to improve the execution efficiency. Current work commonly uses heuristic rules to divide the query plans [3], [5]. However, if the rules are designed with only a single type of workload, it is difficult to achieve optimal division or parallelism for the other workloads [2]. To design and validate better rules, the developers need to increase the diversity of query plans and the output size of each operator.

3) Dynamic Memory Management. The optimization task aims to achieve efficient memory usage of each single node by dynamic memory allocation and assignment for multiple queries [6], [7]. However, it is rather hard to estimate the required memory of a query before execution [6], [7]. This is because the memory usage varies dynamically during different stages of execution. Moreover, even for the same operator, the memory usage differs if their child operators have different output sizes. Therefore, to evaluate and optimize the memory management algorithms, it is needed to construct workloads with various query plans and various output sizes of operators.

To meet the requirements described above, the Query-Aware Database Generator (QAG) [9]–[15] has been proposed. More specifically, given the user-specified database metadata, the query plans with anonymized parameters and the output size of each operator (denoted as cardinality constraint [9]), QAG generates a synthetic database and a set of synthetic queries. When executing synthetic queries on the database, QAG guarantees that the output size of each operator satisfies the cardinality constraints, which is helpful to evaluate mostly workload-aware optimization, e.g., adaptive index [16]–[18]. However, the QAG has been proven to be an NP-complete task [10]. This is because the cardinality constraints of various query operators in queries impose complex 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 relative errors (see Table I, operators in the common OLAP benchmark TPC-H), among which Mirage is our method. Specifically, we identify two key issues in existing methods. Insufficient Operator Support. Existing methods lack the support of: 1) Complex predicates connected by both arithmetic and logical operators. These operators usually introduce the requirement of joint data distributions among multiple non-key columns. To address this issue, existing...
studies [9], [10], [15] propose to leverage the logical predicates (e.g., $col_{<} < 1$ and $col_{=} > 2$) to divide the joint data domain of non-key columns into partitions, and then compute the data proportion for each partitioned domain according to the cardinality constraints of logical predicates. However, the arithmetic predicates (e.g., $col_{*} * col_{=} > 2$) might touch all the data in its involved columns, and they could not be used to partition the joint data domain. Thus, existing work cannot deal with the cardinality constraints of arithmetic predicates.

2) Outer/anti/semi join operators and projection on foreign keys. These operators often introduce the requirement of joint data distributions between primary and foreign key columns. Since each of these operators imposes different joint data distribution requirements compared to the traditional equi-join operator, they would introduce more matching rules between primary and foreign keys. One naive approach is to first enumerate all foreign key population results, and find a feasible result that satisfies all matching rules. However, the enumeration process is computationally prohibitive since the number of results increases exponentially with the table size.

Unbounded Generation Error. As QAG is an NP-complete problem, some work [12], [13] proposes to first generate one dedicated database for each query so as to reduce the computation complexity. Then, the databases are merged to produce fewer databases. However, they could not ensure a reduction to a single database due to conflict distributions among merged databases. Alternatively, some other work [14] adopts a randomized heuristic algorithm, such as the genetic algorithm, to provide an approximate solution. Specifically, they mutate a random database distribution in each algorithm iteration and stop once all cardinality constraints are satisfied or a predetermined number of iterations is reached.

In this paper, we propose a new generator Mirage to deal with the following joint data distribution requirements and guarantee errors of the cardinality constraints on the AQTs.

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 a relational algebra-based \textit{query rewriting} method to push down selection operators without breaking the cardinality constraints in query plans. Note that, the transformation is only used for designing a uniform generation method in Mirage, and users would finally execute the synthetic query with the original query plan. So, we reduce the bidirectional distribution dependencies to a unidirectional one, i.e., from key columns to non-key columns.

### Table I: Comparison of Current Query-Aware Data Generators

<table>
<thead>
<tr>
<th>Related Work</th>
<th>Selection</th>
<th>Join</th>
<th>Projection</th>
<th>Theoretical Error</th>
<th>Cardinality Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>QAGen</td>
<td>Arbitrary</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>Mirage</td>
<td>Arbitrary</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

*Based on the assumption that there is no data sampling.

Between Non-key Columns. It is caused by cardinality constraints of selections on multiple non-key columns connected by logical operators or arithmetic operators. Since QAG pursues the output size of each operator instead of that of each sub-predicate, we can simplify the multi-column distribution dependencies by trimming sub-predicates with the \textit{set transforming rules}. For an arithmetic predicate, we can evade the joint data distribution requirement by searching the parameters in the result space of arithmetic computations on its involved columns. Then we have only distribution requirements on each single non-key column, which can be modeled as data placement requirements, i.e., a classic bin packing problem, and solved with theoretical zero error bound.

Between the Primary and Foreign Key Columns. It is caused by cardinality constraints on $PK$-$FK$ joins and projections on $FK$ column ($FK$ projection cardinality constraint can be transformed to a specific join cardinality constraint [19]). They impose some requirements on the match status between primary and foreign key columns. We define a uniform representation (join cardinality&distinct cardinality) of cardinality constraints for various join operators. To avoid the common conflicts of the population solutions from different joins [12] and reduce computational complexity, a multi-dimensional join visibility-oriented row representation and organization method is proposed for tables, 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 \textit{Constraint Programming} (CP) problem [20] which can be solved by existing \textit{CP Solver} [21] with theoretical zero error. Note, to further control the memory usage, we propose to use a batch data generation strategy after deriving the distributions of non-key and key columns.

In summary, Mirage has the following contributions.

1) \textit{Mirage} provides the most powerful and strong support to complex query operators or predicates as listed in Table I.
2) \textit{Mirage} generates workloads with a theoretical zero error bound even having more supported operators.
3) \textit{Mirage} is able to generate a database of any data size in a linear way with a controllable memory consumption.
4) We run extensive experiments. \textit{Mirage} conquers the previous work by wider operator support and lower errors.

### II. Preliminaries

A. Database and Annotated Query Template

A database $D$ consists of multiple tables. Each table $R$ contains a single primary key column $PK$, a number of non-key columns $A_1, \ldots, A_m$, and zero or more foreign key columns $FK[R_1], \ldots, FK[R_m]$. Here, $FK[R_m]$ indicates that the foreign key of $R$ refers to the primary key of $R_m$.

Cardinality Constraints of Table and Non-Key Column. Given a table $R$ and a non-key column $A$ in $R$, their cardinality constraints refer to the distinct rows and values contained in $R$ and $A$, denoted as $|R|$ and $|R|_A$ respectively.

Annotated Query Template (AQT). A query template can be created by parameterizing the input value of each predicate in a
query plan [14]. An annotated query template AQT [10] labels the cardinality constraint on the output of each operator. For example, there are four AQTs in Fig. 1. 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, which indicates the output size of $\sigma_{s_1 < p_1}$.

### B. Cardinality Constraint of Query Operator

**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 table, and an internal view covers the execution results of a query operator on its input views. In Fig. 1, $V_1=S$ is a leaf view and $V_3=\sigma_{s_1 < p_1}(V_1)$ is an internal view. All view examples for our paper are from Fig. 1.

**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 [22]. That is,

$$P=\text{clause}_1 \land \ldots \land \text{clause}_n \quad \text{s.t.} \quad \text{clause}_i=\text{literal}_i \lor \ldots \lor \text{literal}_n$$

Here, $\text{literal}_i$ can be a unary or an arithmetic predicate, which operates on a single or multiple non-key columns with an arithmetic function. For example, $s_1 < p_1$ in $V_3$ and $t_1 \land t_2 > p_3$ in $V_7$ are unary (on column $s_1$) and arithmetic predicates on (columns $t_1$ and $t_2$), respectively.

**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 $P$. SCC can be classified 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 unary or arithmetic predicates by logical operators ($\land$ or $\lor$), e.g., the ones in $V_9$ and $V_{10}$.

**Join View** ($\Pi^{\text{type}}(V_I, V_r)$). A join view is generated by performing a join operator on two input views $V_I$ and $V_r$ with a specified join type. In general, there are eight types of joins, summarized in Table II. Following previous studies [9]–[15], we also focus on the PK-FK join in this paper.

**Join Cardinality/Disjoint 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 (JCC) and join distinct constraint (JDC). For ease of presentation, hereinafter we assume that $V_I$ contains the primary key ($pk$) of the referenced table, and $V_r$ contains the foreign key ($fk$) of the referencing table. If the $pk$ value of a row in $V_I$ equals the $fk$ value of a row in $V_r$, the two rows can be considered as matched. JCC requires that there should exactly exist $n_{jcc}$ matched pairs of rows in $V_I$ and $V_r$; while JDC requires that there should exactly exist $n_{jdc}$ distinct $pk/fk$ values in $n_{jcc}$ matched pairs. JCC and JDC are used together to determine the output size of any type of join in Table II. For example, suppose the view $V_I$ left outer joins the view $V_r$, and the cardinality constraints are $n_{jcc}$:$n_{jdc}$. We can infer that there exist $|V_I| - n_{jdc}$ rows that are not matched with the rows in $V_r$ and its output size is $|V_I| - n_{jdc} + n_{jcc}$.

**Projection View** ($\Pi_{PK/FK/A}(V)$). A projection view is generated by performing a projection operator on column of an input view $V$. It eliminates the duplicate rows in the result set.

**Projection Cardinality Constraint** (PCC). A projection cardinality constraint $|\Pi_{PK/FK/A}(V)|=m$ requires that the size of projection view 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 of interest in the PCC. Note that projections on non-key columns usually involve dimension columns of any data type but filled with category data [23]–[26], which have small cardinalities. Since the performance of projection has been observed to be only influenced by its input size if its output cardinality is not very huge [14], [27], the projection on non-key columns has a marginal effect on the performance and memory management. However, the cardinality of an $FK$ column is closely related to the cardinality of its referenced $PK$ column, which may be huge and have a great impact on performance. Based on the observations, we only focus PCC on $FK$ columns, which declares the unique number of foreign keys in the output, and can be converted to a JDC requirement on its child join view [19]. For example, the PCC of $|\Pi_{fj_k}(V_5)|=2$ 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 add a virtual right semi join view as its child. 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.

We are now ready to formulate the problem of query-aware database generation in Definition 1.

**Definition 1. Query-Aware Database Generation.** Given the database schema, the cardinality constraints of tables and non-key columns, and the annotated query templates (AQTs), the query-aware database generation (QAG) aims to (1) generate a synthetic database $D$, and (2) instantiate parameters in the AQTs, such that all the cardinality constraints are guaranteed if running the instantiated AQTs on the synthetic database $D$. 
Example 1. Consider 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 $t_{fk}$ referencing $s_{pk}$, and two non-key columns $t_{1}$ and $t_{2}$. The cardinality constraints of $D$ and operators are labelled in Fig. 1. Specifically, for $D$, it has $|S|=4$, $|T|=8$, $|S|_{s_{1}}=4$, $|T|_{t_{1}}=5$ and $|T|_{t_{2}}=4$. Then, QAG aims to construct a synthetic database $D$ and instantiate all parameters in AQTs such that all constraints are satisfied.

III. DESIGN OVERVIEW

For Mirage framework of Fig. 2, its inputs contain database schema, cardinality constraints of table and non-key columns, and annotated query templates (AQTs). For simplicity, we only discuss processing queries without subqueries, which can be eliminated from AQTs with unnesting policies [28]–[32].

Various cardinality constraints impose requirements of joint data distributions among different columns. In general, these joint 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 among different non-key columns. These requirements make the query-aware database generation (QAG) to be $NP$-complete [10]. To address this issue, we propose to decouple the joint distributions (i.e., dependencies) as much as possible to reduce complexity as shown in Fig. 2.

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, and vice versa. Generally, current query optimizer usually tries to use its rule based optimizer (RBO) module to push down select operators directly to each table, so as to reduce the volume of data transferred along the query tree. However, there exists some AQTs in which the selection executes after the join, e.g., TPC-H Q19. To address this issue, we propose to rewrite the annotated query tree based on algebraic transformations so as to push down the selection without breaking the cardinality constraints. Specifically, we propose to first classify the selections into three categories based on their predicates, which are logical predicates operating on multiple tables, arithmetic predicates operating on multiple tables and logical/arithmetic predicates operating on a single table. Then, we extend classic algebraic transformation rules [33], and also design new predicate decoupling rules [34] to facilitate pushing down the selections. With this mechanism, we avoid the bidirectional dependencies between key and non-key columns. This enables us to generate all non-key columns independent of key columns. Note, the rewritten query trees are only used in the generation phase and the original query plans with expected cardinalities are used for testing.

Generate Non-Key Data. Mirage utilizes its non-key generator to populate non-key columns and instantiate the selection related parameters in AQTs, such that all the selection cardinality constraints (SCCs) 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 [10], for the domain size of a joint distribution is the cumulative product of all involved columns’ domain sizes. It then causes expensive domain space searching cost when instantiating parameters in AQTs. To eliminate requirements of joint distributions between non-key columns, we propose the following four processing steps. Firstly, based on the observation that the QAG aims 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 LCCs without affecting the output size of each selection view. With this mechanism, we can decouple LCCs into individual unary cardinality constraints (UCCs) and ACCs (§IV-A). Considering that ACCs would also have the requirements of joint data 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 UCCs on each column (§IV-B and §IV-C); then we search a valid parameter value for each ACC based on the populated non-key columns and its specified cardinality constraints (§IV-D). In details, §IV-B derives each column distribution according to all the UCCs 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 UCCs are satisfied. Following the derived data distribution, §IV-C presents to instantiate parameters in UCCs and generate data for each non-key column. Lastly in §IV-D, 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 satisfying the ACC.

Generate Key Data. The key generator of Mirage is 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 [14] and generate them by an auto-incrementing integer generator. Then, our main aim is to populate FK columns according to the join constraints. For this purpose, we first formalize 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 $V_r$ (§V-A).

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, we propose to partition the table according to the overlaps between all the join input views $V_l$ and $V_r$. Specifically, given a 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 FK populating rules for each join view described above, and then the problem can be modeled as a classic Constraint Programming (CP) problem [20], which can be easily solved by existing CP Solver [21] (§V-B).

### IV. NON-KEY GENERATOR

Logical predicates in logical cardinality constraints (LCCs) may impose joint data dependencies among non-key columns, which makes it NP-complete to generate non-key data. Since our target is to ensure the output size of the whole logical predicate rather than that of its sub-predicate, it inspires us to eliminate some sub-predicates to reduce complexity, which may be achieved by set transforming rules. For example, if we set $p_5=$null and $p_6=\infty$ of $V_9$ in Fig. 3a, the LCC is simplified to a UCC as $\sigma_{t_1 \leq p_4}(T)$ by $\sigma_{(t_1 \leq p_4 \land t_2 = \text{null}) \land t_1 \leq t_2}(T)$ $\cap$ $\sigma_{t_1 \leq p_4}(T)$ $\cap$ $\sigma_{(t_1 < p_4 \lor \text{false}) \land t_1 \leq t_2}(T)$. Finally, LCC is represented by individual unary cardinality constraint (UCC) and arithmetic cardinality constraint (ACC) (§IV-A). Though ACC may still impose joint data dependencies between non-key columns involved in its arithmetic function, it can be simply converted to a parameter search problem in the one-dimensional result space calculated by the arithmetic function on its involved columns. So we propose to first generate data for each column based on its UCCs (§IV-B, §IV-C) and then to fill the parameters for ACCs (§IV-D).

#### A. Decouple Logical Dependencies in LCCs

The logical predicate is assumed to be a CNF formula (see §II-B), and a selection view $V$ with a logical predicate $\land_{k=1}^{n} \text{clause}_k$ can be converted into the intersection of sub-selection views $\cap_{k=1}^{n} V_k$, where $V_k$ is constructed by the sub-predicate $\text{clause}_k$:

$$V = \sigma_{\land_{k=1}^{n} \text{clause}_k}(R) = \cap_{k=1}^{n} \sigma_{\text{clause}_k}(R) = \cap_{k=1}^{n} V_k$$

Each $\text{clause}_k$ can be further represented by a disjunction of literals, and each sub-selection view $V_k$ is decomposed as:

$$V_k = \sigma_{\lor_{i=1}^{m} \text{literals}_i}(R) = \cup_{i=1}^{m} \sigma_{\text{literals}_i}(R) = \cup_{i=1}^{m} V_{k(i)}$$

$V_{k(i)}$ is a sub-sub-selection view constructed by $\text{literals}_i$ in $\text{clause}_k$. For example $V_9$ in Fig. 3a has a logical predicate $P = (t_1 \leq p_4 \lor t_2 = p_5) \land t_1 \leq t_2 < p_6$ and can be decomposed as:

$$V_9 = \sigma_{t_1 \leq p_4 \lor t_2 = p_5}(T) \cap \sigma_{t_1 \leq t_2 < p_6}(T) \cap \sigma_{t_1 \leq p_4}(T) \lor \sigma_{t_2 = p_5}(T)$$

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 sub-sub-selection view, we leverage the following two rules in the area of set theory to simplify the predicate and eliminate some sub-sub-selection views:

**rule1**: $V_l \cap V_j = V_j$ if $V_l \subseteq U$ \hspace{1cm} **rule2**: $V_l \cup V_j = V_j$ if $V_l \subseteq \emptyset$.

The **rule1** and **rule2** indicate that we can eliminate any sub/sub-sub-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 in related to the comparators in each sub/sub-selection view’s predicate, as listed in Table III. 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 $\text{clause}_k$ 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 sub-sub-selection view $V_{k(i)}$ constructed by $\text{literals}_i$ in $\text{clause}_k$ as $\emptyset$, then we eliminate the sub-sub-selection views in a similar way to that of the first step.

**Example 2.** Since $V_9$ is decomposed into $V_9^1 \cap V_9^2$, **rule1** presents that if we can set one item to $U$, then the sub-selection view is eliminated. For instance, we can assign $p_4=\infty$, then $V_9^1 = \sigma_{t_1 \leq \infty \lor t_2 = p_5}(T) = \sigma_{\text{true}}(T) = U$. We can also assign $p_6=\infty$, then $V_9^2 = \sigma_{t_1 \leq t_2 < \infty}(T) = \sigma_{\text{true}}(T) = U$. Since both $V_9^1$ and $V_9^2$ can be $U$, we keep one sub-selection...
view and eliminate the other one to represent the cardinality size requirement, i.e., \(|V_0|=1\). Suppose \(V_0^{(2)}\) is eliminated by setting \(p_{n}=-\infty\). Then we find the boundary values to set the subsub-selection views of \(V_0^{(2)}\) as \(\emptyset\). Specifically, both \(V_0^{(1)}\) and \(V_0^{(2)}\) can be \(\emptyset\) if we set \(p_{1}=-\infty\) and \(p_{5}=null\). Finally, we keep one sub-selection view to represent the cardinality constraint of \(|V_0|=1\). If it is \(V_0^{(1)}\), we eliminate the distribution dependency requirement between \(t_1\) and \(t_2\) in \(|V_0|=1\) to a single column distribution requirement of \(t_1\) in \(|V_0^{(1)}|=1\).

Note, if all the predicates in a clause only contain comparators of \(\not in\), \(\not like\) and \(\neq\), then none of the subsub-selection view can be set as \(\emptyset\) (see Table III). To address this issue, we make use of the De Morgan’s law [35] shown in rule3 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. 3a with the universal set size \(|T|=8\). Then, we can convert the cardinality constraint of \(|V_{10}|=5\) as \(|\sigma_{t_1=p_{7}}(T)\cap\sigma_{t_2=p_{8}}(T)|=3\).

\[
\text{rule3: } |V^{(1)}_1 \cup \ldots \cup V^{(1)}_n| = n \Leftrightarrow |V^{(1)}_1 \cap \ldots \cap V^{(1)}_n| = |U_V| - n
\]

Theorem 1 shows that our elimination procedure can reduce a selection view \(V\) to its sub-selection view \(V^{(1)}(\text{Case1})\) or the conjunction of \(\omega\) unary views \(\bigwedge_{j=1}^{\omega} V_j^{(2)}\) (Case2) whose comparators are \(\not in\), \(\not like\) or \(\neq\). Note that Case2 requires that some values of different columns must coexist in the same row. For example, the above conversion of \(V_{10}\) requires that there must exactly exist three rows whose \(t_1=p_{7}\) and \(t_2=p_{8}\). It can be easily resolved by a post-processing step (§IV-C) after all values generated for each non-key column following its distribution requirement from UCCs (§IV-B). Fig. 3b shows the elimination result of selection views \(V_1, V_2, V_5, V_6\) and \(V_{10}\).

**Theorem 1.** Given a selection view \(V\) from LCC, our elimination procedure can reduce it to one of the following two cases: \(V^{(1)}\) or \(\bigwedge_{j=1}^{\omega} V_j^{(2)}\), where \(V^{(1)}\) is a sub-selection view of \(V\), \(V_j^{(2)}\) is an unary view whose comparator is \(\not in\), \(\not like\) or \(\neq\), and \(\omega\) is the number of unary views.

**Proof.** For elimination, the 1st step is to find boundary values to make sub-selection view \(V^{(1)}\) and keep one sub-selection view to represent LCC requirement. If literal\(_{i}\) in clause\(_{k}\) only contains comparators of \(\not in\), \(\not like\) and \(\neq\), then \(V^{(1)}\) cannot be \(U\). Suppose \(q\) sub-selection views cannot be set as \(U\).

If \(q>0\), it reduces \(V\) to \(\bigcap_{j=1}^{q} V_j^{(2)}\) and clause\(_{k}\) has comparator of \(\not in\), \(\not like\) or \(\neq\). In the 2nd step, each sub-sub-selection view in \(V_j^{(2)}\) can be \(\emptyset\) referring Table III. Finally, it reduces \(V\) to \(\bigcap_{j=1}^{q} V_j^{(2)}\) by keeping one sub-selection view for each \(V_j^{(2)}\).

If \(q=0\), \(V\) is reduced to a single sub-selection view \(V^{(1)}\). Suppose \(s\) sub-sub-selection views of \(V^{(1)}\) (having comparator of \(\not in\), \(\not like\) or \(\neq\)) cannot be set as \(\emptyset\). If \(s=0\), in the 2nd step, Mirage can reduce it to \(V^{(1)}\) with literal\(_{i}\); otherwise, it reduces \(V^{(1)}\) to \(\bigcup_{j=1}^{s} V^{(1)k}\) by keeping all \(s\) literals. By rule3, \(\bigcup_{j=1}^{s} V^{(1)k}\) can be converted to \(\bigwedge_{j=1}^{s} V^{(k)\neq}\) where \(V^{(k)\neq}\) is a unary view with comparator of \(\not in\), \(\not like\) or \(\neq\).

**B. Solve Unary Selection Operators**

Each UCC specifies a distribution requirement for its cardinality constraint on the column’s data distribution. Then, to derive a valid data distribution for the column satisfying all its associated UCCs is equivalent to solving a bin packing problem in its domain size. 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 non-key column to its cardinality space of an integer type such that all the UCCs 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 \(A\)). All the parameters in UCCs are normalized as integers in \([0, |R_A|]\). In this way, deriving \(CDF\) in the original domain space is converted to deriving \(CDF\) in the cardinality space. Note that, Mirage transfers the generated data into the expected data type by a data transformer as defined in [14].

**Definition 2. 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 \(p\) plus \(p\). Moreover, \(F_A(p_1, p_2)\) is defined as the probability of a row whose attribute \(A\) lies in the interval \([p_1, p_2]\), where \(p_1 < p_2\), 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 UCCs (see Table III) into two kinds of comparators, which are \(=\) and \(\leq\). Specifically, the UCC with a comparator \(\not in\) can be transformed into the union of multiple UCCs with the \(\neq\) comparator. Similarly, a UCC with a comparator \(\not like\) can be converted into an in comparator with a pre-defined number of distinct matching values. In addition, based on commutativity property from De Morgan’s Law [35], the comparators \(>, \geq\) and \(\neq\) can be easily converted to \(\leq\) and \(=\). For example, \(|\sigma_{A=p}(R)|-k\) is equal to \(|\sigma_{A=p}(R)|-k\). Then, we use the two kinds of UCCs to derive \(F_A(p)\) and \(f_A(p)\) respectively. If \(|\sigma_{A=p}(R)|=k\), we have \(F_A(p)=k/|R|\) and \(f_A(p)=k/|R|\). As example, consider the three UCCs on column \(t_1\) of Table T in Fig. 3b. From \(|\sigma_{t_1=p}(T)|=6\), \(|\sigma_{t_1=p_4}(T)|=1\), \(|\sigma_{t_1=p_5}(T)|=3\) and \(|T|=8\), we can infer \(F_1(p_2)={8-6}/8=25\%, F_1(p_4)=1/8=12.5\%\) and \(f_1(p_7)=3/8=37.5\%\). Next, given a non-key column \(A\) and its associated \(CDFs\) \(F_A(p)\) and \(f_A(p)\), we take three steps to instantiate all the parameters of UCCs on column \(A\).

**I** Determine the Partial Order for Each Parameter \(p\) in \(F_A(p)\). As \(F_A(p)\) monotonically increases with \(p\), \(p_j\) must be greater than \(p_i\) if \(F_A(p_i)<F_A(p_j)\). Then, for each parameter \(p\) and its corresponding cumulative distribution function \(F_A(p)\), we propose to sort the pair \((F_A(p), p)\) in ascending order of \(F_A(p)\). Suppose we have \(n\) parameters in column \(A\)’s UCCs

<table>
<thead>
<tr>
<th>Comparator</th>
<th>&gt;</th>
<th>≥</th>
<th>&lt;</th>
<th>≤</th>
<th>(;)</th>
<th>not in</th>
<th>not like</th>
<th>≠</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>+∞</td>
<td>−∞</td>
<td>NUL</td>
<td>NUL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE III**

**ASSIGNED BOUNDARY VALUES FOR VIEW ELIMINATION**

<table>
<thead>
<tr>
<th>Comparator</th>
<th>&gt;</th>
<th>≥</th>
<th>&lt;</th>
<th>≤</th>
<th>(,)</th>
<th>not in, not like, −∞</th>
<th>+∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>+∞</td>
<td>−∞</td>
<td>NUL</td>
<td>NUL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE III**

**ASSIGNED BOUNDARY VALUES FOR VIEW ELIMINATION**

<table>
<thead>
<tr>
<th>Comparator</th>
<th>&gt;</th>
<th>≥</th>
<th>&lt;</th>
<th>≤</th>
<th>(,)</th>
<th>not in, not like, −∞</th>
<th>+∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>+∞</td>
<td>−∞</td>
<td>NUL</td>
<td>NUL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
with comparators as \( \leq \). 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. They divide the cardinality space \((0, |R_A|)\) into \( n+1 \) ranges which satisfy
\[
F_A(0, p_{h_i}) + \sum_{j=1}^{n-1} F_A(p_{h_j}, p_{h_{j+1}}) + F_A(p_{h_n}, |R_A|) = 1
\]

(2) Determine the Partial Order For Each Parameter \( p \) in \( f_A(p) \). In this step, we try to put each parameter \( p \) in \( f_A(p) \) into one of the \( n+1 \) ranges properly. Specifically, we suppose we put \( k \) parameters \( p_{e_1}, \cdots, p_{e_k} \) into the range \((p_1, p_j)\) from the first step, 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 \((p_1, p_j)\), i.e.,
\[
\sum_{j=1}^{k} f_A(p_{e_j}) \leq F_A(p_1, p_j)
\]
Then, the problem can be regarded as a bin packing problem, which is \( NP \)-complete [36]. We adopt the state-of-the-art greedy method [37] to address it, whose approximate ratio is \( 2/3 \). It means the total probabilities of parameters placed into ranges is at least \( 2/3 \) of the total probabilities of all parameters. Specifically, we first sort both the parameters and the ranges in descending order of their corresponding probabilities. Then, for each parameter \( p \) in \( f_A(p) \) in descending order, we always find the first range \((p_i, p_j)\) which can accommodate \( p \) validly, i.e.,
\[
F_A(p_i, p_j) - \sum_{j=1}^{k} f_A(p_{e_j}) > F_A(p) \quad \text{with} \quad p_{e_j} \quad \text{as the parameter already put in this range.}
\]
After putting \( p \) into \((p_i, p_j)\), we repeat the above process until all parameters are in the \( n+1 \) ranges. However, 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 \( f_A(p') = f_A(p) \). If so, we specify \( p = p' \). Otherwise, we split \( f_A(p) \) into \( f_A(p_1) = x_1, \cdots, f_A(p_k) = x_k \), with \( x_1 + \cdots + x_k = x \). This can be implemented by converting the UCC from \( |\mathcal{A} \cap p(R)| = x \) to \( |\mathcal{A} \cap (p_1, \cdots, p_k)(R)| = x \). Obviously, the UCC with a smaller granularity of cardinality requirement is more likely to be satisfied. After putting \( q \) parameters \( p_{e_1}, \cdots, p_{e_q} \) into a range \((p_1, p_j)\), we should specify the data existence probability of each newly split range. As we only need to guarantee \( F_A(p_{e_1}, p_{e_2}) \geq f_A(p_{e_1}) \), there are various ways to allocate the rest probability \( F_A(p_{e_1}, p_j) - \sum_{j=1}^{k} f_A(p_{e_j}) \) to each range, e.g., random allocation.

(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 \((p_1, p_j)\) (i.e., \( F_A(p_1, p_j) \)) 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, which is in a uniform way of our case. Note that, when assigning unique values to a range \((p_1, p_j)\), the data existence probability \( F_A(p_1, p_j) \) should not be violated. For example, consider a range \((p_1, p_j)\), 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 \(|R| \cdot (F_A(p_1, p_j) - f_A(p_j))\). 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 locates at or before \( p_i \).

**Example 3.** Consider the column \( t_1 \) of Table \( T \) in Example 1, and its domain size is \(|T| = 5\), 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 UCCs and \(|T| = 8\). An example distribution of \( t_1 \) is shown in Fig. 4. 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 their partial order 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 \( F_{t_1}(7) \) and only the third range has a data existence probability that is greater than \( F_{t_1}(7) \), 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 \( F_{t_1}(p_7, 5) = 62.5\% \) 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 remains in the domain space. Since \(|T| \cdot F(0, p_4) = |T| \cdot F(p_4, p_2) = |T| \cdot F(p_7, 5) = 1\), 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. At last, we have \( p_{t_1} = 1, p_{t_2} = 2 \) and \( p_{t_7} = 7 \). Fig. 3c shows the CDFs for columns \( t_1 \) and \( t_2 \).

C. Generate Data Based on CDF

We generate data for non-key columns following the CDFs, and then arrange values of different columns in the 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 §IV-B). 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. 5, column \( t_2 \) is populated with \(|T| \cdot f_{t_2}(2)\) rows valued 2 based on the CDFs in Fig. 3c. 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 \( \mu \) 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 \( \mu \) unique values based on any given distribution, e.g., uniform distribution. For example in Fig. 5, we assign \( 4 \cdot |T| \cdot f_{t_2}(2, 4) \) rows with two unique values (i.e., 3, 4) uniformly into column \( t_2 \).
Arrange Values from Different Non-Key Columns. After generating data for non-key columns, we start to arrange them in each table. Recall that our decoupling procedure might reduce a selection view $V$ into $\cap_{j=1}^{n} V_{j}^{l}$, where $V_{j}^{l}$ is a unary view whose comparator is $in$, like or $=$. Thus, the $\cap$ operator and $=$ comparator in the cardinality constraint $|\cap_{j=1}^{n} V_{j}^{l}|=n$ requires that the values associated with each view $V_{j}^{l}$ must be bound to the same $n$ rows. To this end, we propose to first populate rows with values meeting the selection views that follow the format of $\cap_{j=1}^{n} V_{j}^{l}$. For example in Fig. 5, based on the requirement $|\sigma_{t_{1}=p_{5}=4(4)} \cap \sigma_{t_{2}=p_{6}=2(4)}|=3$ from $V_{10}$, we bind 3 rows valued (4,2) for $t_{1}$ and $t_{2}$. Then, we populate the rest generated data into rows randomly to satisfy the other selection constraints on the non-key columns.

D. Solver of Arithmetic Selection Operator

The arithmetic cardinality constraint (ACC) may impose joint data dependencies between non-key columns involved in its arithmetic function. After generating the non-key data, to satisfy ACC can be converted to search a parameter in the space generated by the arithmetic function on its involved columns. Specifically, given an ACC $|sg(A_{1},...,A_{j})_{op}(R)|=n$, where $g()$ is an arithmetic function, $A_{1},...,A_{j}$ are the non-key columns operated by $g()$, and $\circ$ is a comparator. We first calculate the result view $g(A_{1},...,A_{j})(R)$ based on the generated non-key columns of table $R$. Then, our aim is to find a parameter $p$ such that there exactly exist $n$ items in the result view which satisfies the predicate. 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 $|V_{7}|=|\sigma_{t_{1} \leq t_{2} \geq p_{5}}(T)|=5$ in Fig. 3b. We first calculate the result view $V=g(t_{1},t_{2})=t_{1} - t_{2}$ based on the populated data in $t_{1}$ and $t_{2}$ as shown in Fig. 6. As the comparator is $>_{\circ}$, we then set $p_{5}$ as the $3^{rd}$ largest value in the result (i.e., $p_{3}=0$). To control the memory usage for calculation, we further 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. Based on Hoeffding’s Inequality [38], the number of sampled rows can be calculated by $\frac{\ln(\delta^{-1} - n(1-\alpha))}{2\alpha}$ according to the given error bound $\delta$ and confidence level $\alpha$.

Until now, we have generated all non-key column data and instantiated all related parameters. For our example, the instantiated parameters in $Q_{1}$~$Q_{4}$ as following and the generated data for non-key columns are shown in Fig. 6, which are then used by key generator to populate foreign key $t_{fk}$.

V. KEY GENERATOR

For the complex joint data distribution requirement between primary keys and foreign keys, most work only deals with equi-join, which cares just about the size of matched pairs between tables. To make our solution general for any join type, we generalize the constraint representation by join cardinality and distinct constraints (JCC&JDC). In §V-A, we propose

Q1: $V_{5}$: $\sigma_{t_{f}}(\sigma_{t_{2}}=30 \cap \sigma_{t_{3}}=2(2\rightarrow 4))$

Q2: $V_{6}$: $\sigma_{t_{f}}(\sigma_{t_{2}}=2 \cap \sigma_{t_{3}}=2(2\rightarrow 8))$

Fig. 6. Non-key Column Data for Table $S$ and $T$ and Instantiated Queries to transform the join constraints of a single join into the populating rules, i.e., the valid way to populate primary keys to the foreign keys of the referencing table. Since we can only populate one foreign key for one row, we have to identify a population solution that satisfies all populating rules. Note that enumerating all population solutions and selecting the valid one is computationally prohibitive. Therefore, we propose to fuse the populating rules from multiple joins to reduce the computation complexity. Specifically, we present a join visibility-oriented row representation for tables, based on which we partition the tables. Then the input key candidates of each join is the union of its visible partitions, and we further represent populating rules by partitions. Finally, the problem of fusing the populating rules is a classic Constraint Programming (CP) problem [20] which can be accurately solved by existing CP Solver [21](in §V-B).

A. Single Join Constraint on Two Tables

As defined in §II-B, a join view $\mathcal{Join}(V_{l}, V_{r})$ has two input child views $V_{l}$ and $V_{r}$. For a single join view, its JCC requires that there exist $n_{jcc}$ matched pairs of rows in two input views (i.e. the $pk$ of a row in $V_{l}$ equals the $fk$ of another row in $V_{r}$). Its JDC requires that there exactly exist $n_{jdc}$ distinct $pk/fk$ values in all matched pairs of rows. 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 $PF_{V_{l} \rightarrow V_{r}}=n_{jcc}$). Meanwhile, we can also infer that the rest of $|V_{l}| - n_{jcc}$ 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 that do not exist in $V_{l}$ (denoted as $PF_{\neg V_{l} \rightarrow V_{r}}=n_{jcc}$). In addition, we must exactly select $n_{jdc}$ primary keys in $V_{l}$ when populating foreign keys in $V_{r}$ (denoted as $PF_{V_{l} \rightarrow V_{r}}=n_{jdc}$). To summarize, we derive three foreign key populating rules from join cardinality and distinct constraints in Eqn. 1.

$PF_{V_{l} \rightarrow V_{r}}=n_{jcc}$ $PF_{\neg V_{l} \rightarrow V_{r}}=|V_{r}| - n_{jcc}$ $PF_{V_{l} \rightarrow V_{r}}=n_{jdc}$

Recall that we always firstly push down selection operators (see §III) 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 Eqn. 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 $\mathcal{Join}(V_{l}, V_{r})$.

Example 4. Consider the join view $V_{5}$: $\mathcal{Join}_{t_{f}}(V_{3}, V_{4})$ in Fig. 6. From its join cardinality constraints $n_{jcc}=3$, there exist

\[ \frac{\ln(\delta^{-1} - n(1-\alpha))}{2\alpha} \]
3 matched pairs of rows for \( V_3 \) and \( V_4 \). Additionally, from \( |V_6|=|H_{jcc}(V_3)|=2 \), we can infer that 2 distinct \( s_{pk}/tfk \) in matched rows, i.e., \( n_{jdc}=2 \). Based on Eqn. 1, the three foreign key population rules for \( V_4 \) are listed as follows.

\[
PF_{V_3 \rightarrow V_4} = 3 \quad PF_{V_5 \rightarrow V_4} = |V_4| - n_{jcc}=3 \quad PF_{V_3 \rightarrow V_4} = 2
\]

Fig. 7a 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 \( PF_{V_3 \rightarrow V_4} = 3 \) and \( PF_{V_5 \rightarrow V_4} = 2 \), we select 2 primary keys in \( V_3 \) to populate 3 foreign keys in \( V_4 \). Then we can populate the first three foreign keys in \( V_4 \) as 1, 2 and 2. According to the rule \( PF_{V_5 \rightarrow V_4} = 3 \), we should populate the rest foreign keys in \( V_4 \) by any primary key in \( V_5 \), e.g. 3 in \( V_5 \). Finally, since \( V_4 \) does not affect the output of \( bo_{tfk}^m (V_3, V_4) \), the foreign keys in it can be populated with any primary key in table \( S \), e.g., the primary key 1.

**B. Multiple Join Constraints on Two Tables**

Suppose there exist \( m \) join views on two tables \( S \) and \( T \), where \( T \) references \( S \). If we populate the foreign keys according to the constraint 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, the row \( tfk=1 \) in \( T \), its foreign key is populated as \( tfk=1 \) in Fig. 7a by \( V_5 \) and \( tfk=2 \) Fig. 7b by \( V_6 \). Though we can enumerate all foreign key population results and find a feasible one, it is computationally prohibitive with the complexity of \( O(|S|^T) \), for each row of \( T \) can be populated by any one of the primary keys in \( S \).

Note that, there exist some overlaps of primary keys between the input left child views. For example, the left child views \( V_1 \) and \( V_3 \) in Fig. 7 contain the primary keys \( 1, 2, 3, 4 \) in \( V_3 \). 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 [20]. Finally, we take advantage of the existing solver 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 easier 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 \( V_k^l \) and \( V_k^r \), respectively. If a primary key in the referenced table \( S \) is contained in \( V_k^l \), 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, we use a status value 0/1 to indicate whether a row in \( S \) is a join candidate of a given join view. Suppose there exist \( m \) join views, then each row in \( S \) is associated with an \( m \) dimensional status vector. Similarly, each row in the referencing table \( T \) also has an \( m \) dimensional status vector, which indicates whether it is an input join candidate in each of the \( m \) right child views, i.e., whether \( V_k^r \) contains the row’s foreign key.

**Example 5.** Fig. 8 shows the 2 dimensional status vectors of tables \( S \) and \( T \) regarding two join views \( V_5 = \langle b_{tfk}^m (V_3, V_4) \rangle \) and \( V_6 = b_{tfk}^l (V_1, V_7) \) in Fig. 6. 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 \( V_7 \). Consider the referenced table \( S \), 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 rows have the same value, we put them into the same partition. For example, consider table \( S \) in Fig. 8. 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 1~2 and \( S_2 \) consists of rows 3~4. Similarly, table \( T \) can be partitioned into \( T_1, T_2 \) and \( T_3 \). Since the dimension of status vectors is \( m \), the number of partitions is at most \( 2^m \). However, it is much smaller than \( 2^m \) in practice because dependencies among statuses can greatly reduce the enumeration space of status vectors, and we have put analysis of the theoretical number of partitions in our technical report [34].

**(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, it means that all the primary keys in \( S_i \) are contained in \( V_k^r \). For a referencing partition \( T_j \), setting its \( k^{th} \) bit as 1 indicates that all the foreign keys in \( T_j \) are contained in \( V_k^l \). For ease of presentation, hereinafter we use \( S_i/T_j \) to represent a partition of primary/foreign key column. Then, for any join view \( V_k \), foreign key population from \( V_k^r \) to \( V_k^l \) is

\[
V_k^r \rightarrow V_k^l = \bigcup_{S_i \subseteq V_k^l} \bigcup_{T_j \subseteq V_k^r} S_i \rightarrow T_j = \bigcup_{S_i \subseteq V_k^l} S_i \rightarrow T_j \tag{2}
\]

From Eqn. 2, we can convert three populating rules on a join view in Eqn. 1 into the ones on their associated partitions. Here, \( n_{jcc} \) and \( n_{jdc} \) denote the join cardinality and distinct constraints on the \( k^{th} \) join view.

\[
PF_{V_k^r \rightarrow V_k^l} = \bigcup_{S_i \subseteq V_k^l} \bigcup_{T_j \subseteq V_k^r} PF_{S_i \rightarrow T_j} = n_{jcc}^k \quad PF_{V_k^l \rightarrow V_k^r} = \bigcup_{S_i \subseteq V_k^l} \bigcup_{T_j \subseteq V_k^r} PF_{S_i \rightarrow T_j} = n_{jdc}^k \quad PF_{V_k^r \rightarrow V_k^l} = \bigcup_{S_i \subseteq V_k^l} \bigcup_{T_j \subseteq V_k^r} PF_{S_i \rightarrow T_j} = |V_k^l| - n_{jcc}^k \tag{3}
\]

Further, 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 number of foreign keys populated by partitions in table $S$ must equal to its number of rows.

$$\sum_{S_i \subseteq S} PF_{S_i \rightarrow T_j} = |T_j|$$  \hspace{1cm} (4)

**Example 6.** Consider the join view $V_3 = \prod_{T_4} (V_3, V_4)$ in Example 4. From Fig. 8, we have $V_3 \rightarrow S_1, V_3 \rightarrow S_2$ and $V_4 = T_1 \cup T_2$. Based on Eqn. 3, we have the populating rules as

$$PF_{V_3 \rightarrow V_4} = PF_{S_1 \rightarrow T_1} + PF_{S_1 \rightarrow T_2} = 3$$

$$PF_{V_3 \rightarrow V_4} = PF_{S_2 \rightarrow T_1} + PF_{S_2 \rightarrow T_2} = 3$$

$$PF_{V_3 \rightarrow V_4} = PF_{S_1 \rightarrow T_1} + PF_{S_1 \rightarrow T_2} = 2$$

Based on Eqn. 4, the two following equations make sure that the foreign keys in $T_1$ and $T_2$ are exactly covered.

$$PF_{S_1 \rightarrow T_1} + PF_{S_2 \rightarrow T_1} = |T_1| = 5$$

$$PF_{S_1 \rightarrow T_2} + PF_{S_2 \rightarrow T_2} = |T_2| = 1$$

**3) Construct Constraint Programming Problem.** By combining all of the populating rules in Eqs. 1 \~ 4, we can formalize the problem as a Constraint Programming (CP) problem [20]. Then, after finding a feasible solution for a set of variables stated in the constraint equations, we can follow it to populate foreign keys in each partition $T_j$. However, decomposing populating rules of join views into partitions would enlarge the solution space and lead to contradictory solutions. To address this issue, we further introduce three constraints to ensure the validity of populating results.

1. The number of populated $fks$ must not be less than the number of populated distinct $fks$, i.e., $PF_{S_i \rightarrow T_j} \geq PF_{S_i \rightarrow T_j}^d$.

2. If a partition $S_i$ uses its $pks$ to populate $fks$ in partition $T_j$, then at least one $pks$ is used, i.e., $PF_{S_i \rightarrow T_j} > 0 \Rightarrow PF_{S_i \rightarrow T_j}^d > 0$.

3. The size of selected $pks$ from $S_i$ cannot exceed the size of its $pk$ candidates, i.e., $|S_i| \geq \sum_{T_j \subseteq V_k} PF_{S_i \rightarrow T_j}^d$.

After integrating all the three constraints into the CP problem, we propose to use the Or-Tools [21] as our CP solver. It improves its efficiency by utilizing the constraint propagation method to prune the search space. Then we can use the solution to populate foreign keys.

**Example 7.** Consider the join constraints from two join views $V_5$ and $V_6$. After we include the above three constraints, we can find a feasible solution without contradictions as follows.

$$PF_{S_1 \rightarrow T_1} = 2$$

$$PF_{S_2 \rightarrow T_1} = 1$$

$$PF_{S_2 \rightarrow T_2} = 3$$

$$PF_{S_2 \rightarrow T_2} = 0$$

$$PF_{S_1 \rightarrow T_1} = 1$$

$$PF_{S_1 \rightarrow T_2} = 1$$

$$PF_{S_2 \rightarrow T_2} = 2$$

$$PF_{S_2 \rightarrow T_2} = 0$$

**Fig. 9** presents the result of populating foreign keys for table $T$ based on this solution. Specifically, from $PF_{S_1 \rightarrow T_1} = 2$ and $PF_{S_1 \rightarrow T_1} = 1$, we should select one primary key from $S_1$ to populate two foreign keys in $T_1$, i.e., 1. In addition, recall that the foreign keys in $T_3$ can be populated by arbitrary primary keys, we populate $T_3$ with 4.

Note that CP Problem may be unsolvable due to the constraints exactly conflict with each other. 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. To satisfy the join constraints on multiple tables, we populate the foreign keys following the topological order among tables based on their PK-FK references [39].

**VI. DISCUSSION**

**Supported Query Types and Operators.** Mirage still could not cover all query types [40]–[42] as in Table IV, where $T$ means supported by Mirage and $F$ means no support from any work. Mirage is only dominated by PiGen [43] for the non-key projection in the single table query. However, PiGen is mainly designed for non-key projection operators considering no joins. Thus, it has a weaker operator support compared to Mirage. Note that Mirage deals with join constraints on multiple tables following the topological order of reference relationship, so it can not process cyclic query types.

**Error Bound.** For non-key generator, we follow strict rules to decouple and simplify data dependencies among columns. We generate data based on CDFs that satisfy all UCCs, i.e., no error occurs (as in Theorem 2). For ACCs, we accurately calculate parameters from the generated data, introducing no error. However, to avoid memory overflows caused by large volumes of data, sampling-based method is used to calculate the parameters of ACC, which may introduce errors. Even so, we still guarantee the error bound $\delta$ in a confidence level $\alpha$ if the sampling data size is no less than $\frac{ln2-1n(1-\alpha)}{2\delta^2}$ [38].

**Theorem 2.** Non-key generator can solve unary cardinality constraints (UCCs) on non-key column without error.

**Proof.** Revisit our algorithm in §IV-B. We represent UCC with the distribution requirement on a non-key column $A$ by $f_A(p)$ or $f_A(p, q)$, where $p$ is the parameter of its predicate. We first divide the whole domain space of column $A$ into multiple sub-ranges by parameters following the partial order of $F_A(p)$. Then, for the distribution requirement $f_A(p, q)$, it is to put $p$ in a valid sub-range, i.e., $(p_1, p_2)$, with a distribution probability no less than $f_A(p)$. If we can not find such a range for $f_A(p)$, $f_A(p)$ can be divided into smaller probability requirements (see §IV-B). So $f_A(p)$ can be exactly guaranteed. And inserting an $f_A(p)$ into a range does not introduce any distribution probability error for the original range $(p_1, p_2)$. So our distribution generation algorithm does not introduce any error for distribution requirements from UCCs. By exactly generating $|R| * F(p_1, p_2)$ values for each range $(p_1, p_2)$ (see §IV-C), the non-key generator satisfy UCCs with no error. □

For key generator, fusing join constraints from all join views are formulated as a classic Constraint Programming (CP) problem. It can be accurately solved by a CP Solver. But join constraints for a single join view may not be guaranteed if taking the sampling-based parameter instantiation for ACC with a given error bound $\delta$ and confidence $\alpha$ (see §IV-D). Specifically, running an arithmetic predicate on our generated

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Selection</th>
<th>PK-FK join</th>
<th>Non-PK-FK join</th>
<th>Projection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single table</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Union</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Star</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Clique</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
### VII. Related Work

**Data-aware generators** [44]–[49] aim to generate a database instance closely related to the given data characteristics which is independent of workload. For example, *Alexander* [46] designs pseudo-random number generators and *Torlak* [44] uses kinds of multi-dimensional models. Although the generated database satisfies the given data distribution, the cardinality of each operator cannot be guaranteed [14].

**Query-aware generators** [9]–[15], [50] aim to generate a database that satisfies the cardinality constraints of required query plans. *QAGen* [9] is the first work for query-aware database generation. It has powerful support in operators, but can only generate one database for one query. Subsequent work focuses on the generation based on multiple queries, which has been proved *NP-complete* [10]. These work makes a great effort to reduce the computation complexity rather than improve the operator support-ability. Specifically, *My-Benchmark* [12] 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, it can not guarantee to generate one single database. *Touchstone* [14] 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 error. However, it cannot sample for the cardinality constraints from arbitrary logical predicates or support semi-anti-join [49].

*DCGen* [10] and *Hydra* [15] transfer the generation task into multiple Linear Programming (LP) problems. However, the LP model cannot be adapted for the arithmetic filter operator and outer-/-semi-join. *Loki* [50] works well only for solving the selection cardinality constraints. Compared to these work, *Mirage* can give the most powerful support to complex operators and guarantee error to be *zero* theoretically.

**Transaction-aware generators** [51]–[53] aim to synthetic a scenario for transaction processing. *Lauca* [51], [52] focuses on the simulation of access patterns. *Dike* [53] emphasizes the distribution control of distributed transactions.

### VIII. Experiment

**Database:** Since ensuring the cardinality constraints of operators is independent of the database, PostgreSQL (v.14.2) is used to verify the design of *Mirage*. It is equipped with 2×Intel(R) Xeon(R) Gold 6240R CPU, 390GB memory, and 2.5TB disk.

**Workloads:** We compare *Mirage* with recent work *Touchstone* [14] and *Hydra* [15]. Classic benchmark scenarios from SSB [54], TPC-H [55] and TPC-DS [56] are used to provide query plans and cardinality constraints. To test the generation scalability, *Hydra* provides large-scale workloads from TPC-DS, but removes its unsupported complex operators and finally we take all the 100 distinct queries. For the slow data import speed of the database, when comparing database performance, we take a small scale factor $SF=1$: for evaluating generation efficiency, a large default $SF=200$ is used.

**Metrics:** We adopt relative error, i.e., $\frac{|\hat{V}_i - V_i|}{V_i}$ in [14] to measure the simulation cardinality deviation for query $Q$. $|\hat{V}_i|$ represents the required cardinality of the $i^{th}$ view $V_i$ in query $Q$. $|V_i|$ represents the cardinality of $V_i$ in instantiated query $Q$ on the generated database.

**Setting:** We take the sampling-based parameter instantiation for *ACC*. Its default sampling size is 4 million (4M) rows with a theoretical error bound 0.1% in a confidence level 99.9%. To control memory usage, we use a batch generation strategy. For each batch, we first generate non-key columns with their CDFs, and then populate joins based on the join constraints whose sizes are scaled down by the ratio between batch size and table size. The default batch is set as 7M rows. Note 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.

### A. Comparing with the State-of-the-art Work

**Comparison of Workload Support Ability and Fidelity**

We compare the workload support ability and the generation fidelity with *Touchstone* and *Hydra*. We plot the relative errors for all queries in Fig. 10. Note, if a query is not supported by a specific method, we mark its relative error as 100%. To plot the errors clearly for TPC-DS, we divide every 5 queries of TPC-DS into a group and show the result of each group.

*Touchstone* gives full support to generate the simple SSB application as in Fig. 10a and a TPC-H application with its first 16 queries as in Fig. 10b. But it cannot scale well to generate a TPC-DS application and only 25 queries are supported as in Fig. 10c. For the lack of operator support, even for SSB, *Hydra* can not process $Q_2$ and for TPC-H, it can only generate an application with 6 out of all 22 queries. Even though *Hydra* can support all queries in its TPC-DS workload, it has a higher error than *Mirage* which has errors (about 1%) for its sampling-based parameter filling of *ACC* in TPC-H. So *Mirage* conquers the related work for its strongest support to complex operators with the best generation fidelity.

| Database, its output size $|\hat{V}_i|$ may be less than the cardinality requirement $|V_i|$ of its *ACC*, and its error is $1 - \frac{|\hat{V}_i|}{V_i} \leq \delta$, see §IV-D). Suppose $V_i$ is the right child view of a join. If $|\hat{V}_i| < n_{jcc}$, the output size of the join is at most $|V_i|$. The relative error of *JCC* for current join is $1 - \frac{|\hat{V}_i|}{n_{jcc}} \leq 1 - \frac{|\hat{V}_i|}{|V_i|} \leq \delta$. Similarly, we have the same conclusion for the relative error of *JDC*. So, the relative error of each join is bound by $\delta$, if taking the sampling-based processing for *ACC* or else no error occurs.

**Fig. 10.** Comparison for Relative Errors of Various Workloads including SSB, TPC-H, and TPC-DS (100% error means no support)
Comparison of Generation Efficiency The OLAP database usually has big volume of data, which requires the generation tool to have a high generation speed. We compare their generation time in Fig. 11 by changing SF from 200 to 1000. Note that each tool only generates the application scenarios with its supported queries (size labeled on figures).

For fair comparisons, we only analyze the results with the same workload size. Specifically, for SSB, Mirage is as fast as Touchstone; for TPC-DS, Mirage is $2 \times$ slower than Hydra. For TPC-H, though Mirage is slower, it supports more queries in generation. For TPC-DS in Fig. 11c, 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 as Hydra. Take TPC-DS for example. Mirage is about $2 \times$ faster than Hydra when applying the duplicating strategy, because Mirage only puts the join constraints into a $CP$ problem, but Hydra models all the constraints into an LP problem. So Mirage is more practical and guarantees to generate data instead of duplicating from a small-sized data.

B. Evaluating Proposed Techniques of Mirage

Memory Consumption of Mirage Maintaining too much data dependency in memory has always bottlenecked the data generation efficiency [14], [15]. To balance generation performance and memory usage, Mirage can generate data in batch. Given a table, a large batch usually occupies more memory, but it decreases the total rounds of generations.

![Fig. 11. Comparing with Existing Methods by Varying Data Sizes](image1)

![Fig. 12. Evaluating the Effect of Batch Sizes](image2)

Here, we illustrate the impact of batch size for generation by changing batch size in Fig. 12. Given a workload, the generation time for $GN$, $CS$, and $PF$ is only relevant to table size, which is stable. However, a larger batch size reduces the number of rounds to solve $CP$, which decreases generation time for $CP$. But the gain from $CP$ decreases as the batch size increases. For example, the turning point is $4M$ rows for TPC-H, after which we obtain only a slower performance improvement from $CP$. Fig. 12 also shows the memory required for the generation is linearly related to the batch size. Since TPC-DS has wider tables, it consumes memory for data generation about twice as much as that of TPC-H or SSB. To trade off the overall generation efficiency against memory usage, taking the batch sized $7M$ rows (the default setting) almost uniformly reaches the peak performance for all workloads and the memory used is no more than 28GB. So Mirage can reach its peak performance with conservative memory usages decided by the batch size.

Generation Scalability of Mirage 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. 13a). 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. 13b). 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 accessing relationship of join tables (details in technical report [34]). In the $i^{th}$ group, we randomly sample 100 query templates involving joins with at most $i$ tables and the result is shown in Fig. 13c. We observe the total generation time of non-key columns is relatively small; $GN$, $CS$, and $PF$ scale well with the number of query templates and query instances per template. As the $CP$ problem is $NP$-complete [57], the time of resolving $CP$ increases super-linearly. When a join involves more than 5 tables, the time for $CP$ increases suddenly. Because more query templates introduce an exponential growth of status vectors of $l_{orderkey}$, i.e., a sharp increase of parameters in the $CP$ solver. However, the memory consumption is quite stable because of the batch-based data generation strategy. So Mirage is linearly scalable with $GN$, $CS$ and $PF$; $CP$ is $NP$-complete, which is super-linearly scalable.

IX. Conclusion

To optimize query parallelism techniques, we propose Mirage, a query-aware data generator with the most powerful support to complex OLAP workloads and achieve data generation with zero theoretical error bound. Extensive experiments have verified the effectiveness of Mirage.

ACKNOWLEDGMENT

The work is supported by NSFC (92270202, 62072179, 62202171, U22B2020), Fundamental Research Funds for the Central Universities (YBNTS2022-013), Shanghai Pujiang Program (21PJ1403200) and Ant Group Research Fund.