

# 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, hao.li@stu, zrhu@stu, rzhang@dase, ccyang@dase, pcai@dase, xzhou@dase, ayzhou@dase}.ecnu.edu.cn

**Abstract**—To optimize the query parallelism techniques, database developers require substantial workloads with specific query plans and customized output size for each operator (denoted as cardinality constraint). To this end, a rich body of query-aware database generators (QAG) are proposed. Given user specified database schema, column domain and query templates annotated with cardinality constraints, the QAG aims to generate a synthetic database and a set of synthetic queries such that all the cardinality constraints are guaranteed when executing synthetic queries on the database. However, the complex data dependencies hidden behind queries make previous QAGs suffer from deficiencies in supporting complex operators and controlling the generation errors. In this paper, we design a new generator *Mirage* with good properties of supporting complex operators and providing low error bounds for cardinality constraints. First, *Mirage* leverages the *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. For the key columns, *Mirage* proposes a uniform representation of join cardinality constraints for all types of PK-FK joins and partition 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*. We have run extensive experiments and the results show that *Mirage* has a good scalability in data generation, and it greatly conquers all the previous work in either operator support or generation error.

## I. INTRODUCTION

Modern query execution engine supports high-performance query processing with the help of inter- and intra-query parallelisms. The inter-query parallelism [1] schedules different queries among multiple cluster nodes, while the intra-query parallelism [1] enables concurrent execution of multiple operators within a single query. Due to the complex orchestration of query and query operators during execution, the overall performance of the query engine is highly sensitive to the inter-query scheduling policies, intra-query parallelism optimization, dynamic memory management [2]–[7], etc. To conduct comprehensive evaluation and optimization of these performance critical algorithms, database developers would like to generate substantial workloads by customizing the query plans and the output size of each operator. We illustrate the scenarios desiring for these workloads as follows.

**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. The state-of-the-art work leverages machine learning to learn a generalized schedule solution [4], [8] which works well under various workloads. However, it requires a number of diverse workloads for training. Specifically, the diversity of workloads is commonly represented by the diversity of query plans, query operators and the output size of each operator [4], [8].

**2) Intra-query Parallelism Optimization.** The optimization task divides a query plan into multiple parts, and then database can concurrently execute them with an appropriate level of parallelism to improve the execution efficiency. The state-of-the-practice work commonly uses heuristic rules to divide the query plan and decides the number of concurrent threads [9]–[12]. 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. For example, a substantial amount of memory is required for sorting data during the merge join, which is later released after the completion of join. Moreover, even for the same operator, the memory usage differs if their child operators have different output sizes. Therefore, when evaluating and optimizing the dynamic memory management algorithms, the developers need 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) [13]–[19] has been proposed. More specifically, given the user specified database metadata (including schema, table size and column domain), the query plans and the output size of each operator (denoted as cardinality constraint [13]), 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. Note that, the parameter variables in these query plans are anonymized, and database developers just need to design the query plan and annotate the cardinality constraint of each operator.

TABLE I  
COMPARISON OF CURRENT QUERY AWARE DATA GENERATORS WITH *Mirage* IN OPERATOR SUPPORTABILITY

Related Work	Selection			Join				Projection	Theoretical Relative Error	Terabyte- Generation
	Predicate	Arithmetic	Logical	Equi	Anti	Outer	Semi	Foreign Key		
QAGen	Arbitrary	F	Arbitrary	T	F	F	F	T	Zero	F
MyBenchmark	Arbitrary	F	Arbitrary	T	F	F	F	T	No Guarantee	F
DCGen	$>, \geq, <, \leq, =$	F	DNF	T	F	F	F	F	Low	T
Hydra	$>, \geq, <, \leq, =$	F	DNF	T	F	F	F	F	Zero	T
Touchstone	Arbitrary	T	Simple	T	F	F	F	F	No Guarantee	T
<b>Mirage</b>	<b>Arbitrary</b>	<b>T</b>	<b>Arbitrary</b>	<b>T</b>	<b>T</b>	<b>T</b>	<b>T</b>	<b>T</b>	<b>Zero</b>	<b>T</b>

However, the QAG has been proven to be an *NP-complete* task [14]. This is because the cardinality constraints of various query operators in queries impose complexing 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), among which *Mirage* is our method. Note, all operators in Table I exist in the common OLAP benchmark TPC-H. 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 [14]; **2) Outer/anti/semi join operators and projection on foreign keys in enormous database.** It introduces joint data distribution requirements between primary and foreign key columns. Moreover, existing work for this issue usually relies seriously on memory consumption [13], [16], [17] and are not scalable.

In this paper, we propose a new generator *Mirage*, which can not only successfully solve the complex joint data distribution requirements from different queries/operators, but also strictly guarantee the relative errors of the cardinality constraints on the AQTs. Specifically, it utilizes different techniques to deal with joint data 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 a relational algebra-based *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.

**Between Non-key Columns.** It is introduced by cardinality constraints of selections on multiple non-key columns which are connected by logical operators (e.g.,  $\wedge$ ) or arithmetic operators (e.g.,  $+$ ). Since the purpose of QAG is to guarantee the output size of each operator instead of that of each sub-predicate on the generated testing scenario, it inspires us to simplify the multi-column distribution dependencies by eliminating some sub-predicates 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 result

space of arithmetic computations on 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 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 [16] 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 *Constraint Programming* (CP) problem [20] which can be simply solved by existing *CP Solver* [21].

Based on above solutions, we implement *Mirage*, a novel and generic QAG, which has the following contributions.

- 1) *Mirage* provides the most powerful and strong support to complex query operators or predicates as listed in Table I.
- 2) *Mirage* generates workloads 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 a controllable memory consumption.
- 4) We run extensive experiments. *Mirage* conquers the previous work by a wider operator support and lower errors.

## II. PRELIMINARIES

### A. 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 column  $PK_i$ , a certain number of non-key columns  $A_{i_1}, \dots, A_{i_q}$ , and zero or more foreign key columns  $FK[R_{i_1}], \dots, FK[R_{i_m}]$ . 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  $A$ .  $|R|$  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 [18]. An annotated query template AQT [14] 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  $\sigma_{s_1 < p_1}$ .

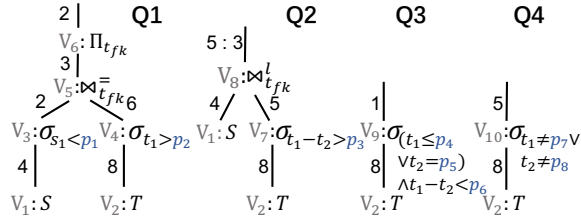


Fig. 1. Annotated Query Templates on Tables  $S$  and  $T$

### B. Cardinality Constraint of Query Operator

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

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

**Example 1.** The AQT  $Q_1$  in Fig. 1 can be represented by query operator views from the bottom up as follows.

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

**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 [22]. That is,

$$P = \text{clause}_1 \wedge \dots \wedge \text{clause}_n \quad \text{s.t.} \quad \text{clause}_i = \text{literal}_i \vee \dots \vee \text{literal}_n$$

Here,  $\text{literal}_i$  can be a unary or an arithmetic predicate. More specifically, a unary predicate follows the form of  $A_i \bullet p_k$  with  $\bullet \in \{=, \neq, <, >, \leq, \geq, (\text{not}) \text{ in}, (\text{not}) \text{ like}\}$ , e.g.,  $s_1 < p_1$  in  $V_3$ . An arithmetic predicate might operate on multiple non-key columns  $A_i, \dots, A_k$  through an arithmetic function  $g()$ , which usually follows the form of  $g(A_i, \dots, A_k) \circ p_k$ ,  $\circ \in \{<, >, \leq, \geq\}$ , 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 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 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*

TABLE II  
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  - n_{jdc} + n_{jcc}$	$ V_l  - n_{jdc} +  V_r $	$n_{jdc}$	$n_{jcc}$	$ V_l  - n_{jdc}$	$ V_r  - n_{jcc}$

*join* ( $la/ra$ ). Following previous studies [13]–[19], 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 ( $JCC$ ) and join distinct constraint ( $JDC$ ).  $JCC$  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.  $JDC$  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 II). For example, suppose the view  $V_l$  left outer joins the 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  $|V_l| - n_{jdc} + n_{jcc}$ .

**Projection View ( $\Pi_{PK/FK/A}(V)$ ).** A projection view is generated by performing a projection operator on 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 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 of interest in the  $PCC$ . 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 [18], [23]. 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* [24], *textual data* [25], its cardinality is relatively small as it is filled with category data. For example, the max cardinality of all the dimension columns in these datasets [24]–[27] is 2112, which is far less than their table sizes. Thus, the output size of projections on non-key columns is relatively small and then 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 has a great impact on projection performance. Based on the observations, we only focus  $PCC$  on  $FK$  columns to simplify the complexity of QAG. Note, 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, *Mirage* automatically adds 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. 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 *PCCs* to *JDCs* 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. For example, we add a child view  $V_{13}$  for the projection view  $V_{12}$  as shown in Fig. 2

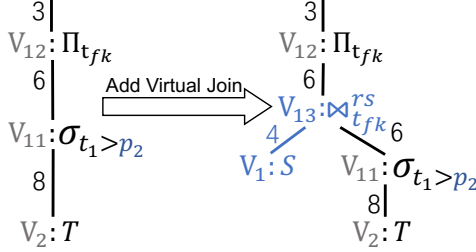


Fig. 2. Add a Virtual Join View for the Projection View

### C. Problem Definition

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 2.** Consider 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 AQTs are listed in Fig. 1. Then, the query aware database generation (QAG) aims to construct a synthetic database  $D$  and instantiate all parameters in AQTs. The results are shown as in Fig. 3.

## III. DESIGN OVERVIEW

Figure 4 shows an overview of our *Mirage* framework. Its input configurations contain the database schema, the cardinality constraints of table and non-key columns, and the AQTs. For simplicity, we only discuss processing queries without subqueries. Note, *Mirage* can also be extended to process subqueries, which is discussed in §??.

Various cardinality constraints have 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

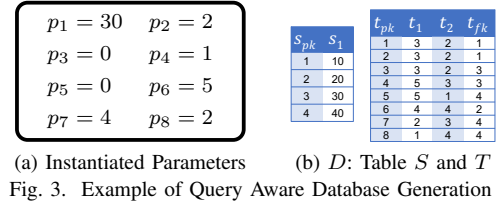


Fig. 3. Example of Query Aware Database Generation

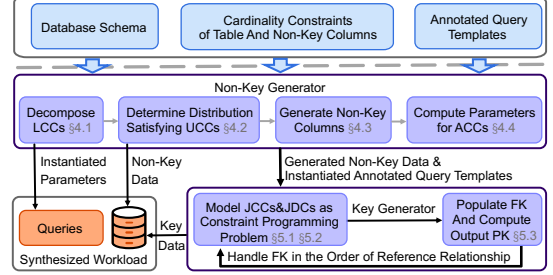


Fig. 4. Design Overview of *Mirage*

and non-key columns, and among different non-key columns. These requirements make the query aware database generation to be *NP-complete* [14]. 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 (see Fig. 4).

**Decouple Dependencies Between Key and Non-key Columns.** The requirement of the joint distributions between key and non-key columns is from the uncertain execution order between selection on non-key columns and join on key columns. If the selection executes after the join, it indicates that the filter result relies on the join result, i.e., the distribution of non-key columns relies on that of key columns. Otherwise, the distribution of key columns relies on that of non-key columns. Generally, 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, there exists some AQTs in which the selection executes after the join. For example, the residual predicate in TPC-H Q19 always executes after the join operator. 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 (more details in §??). 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 original plan are guaranteed when performing tests on the generated database.

**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* [14]. 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 *AQTs*. To address the 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 *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 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 *UCCs* on each column (§IV-B and §IV-C). 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 (§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 such that the *ACC* is satisfied.

**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 [18] and generate them by an auto-incrementing integer generator. Then, our main aim is to populate *FK* columns according to the join constraints. For this purpose, we first formalize three *FK* populating rules for each *JCC/JDC* on two joined tables. Specifically, for any join view, the rules specify how to select primary keys in its left input view  $V_l$  and how to use them to populate foreign keys in its right input view  $V_r$  (§V-A). However, we observe that there might exist some conflicts if we populate foreign keys according to the *JCC/JDC* of

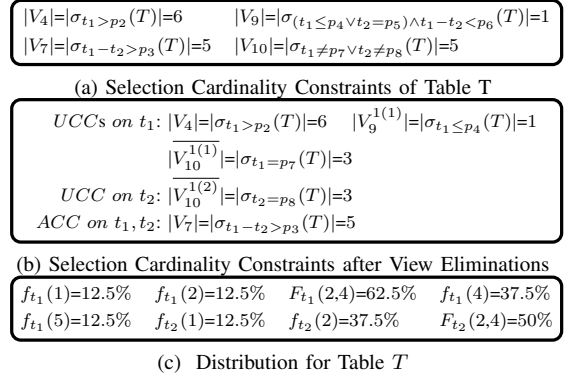


Fig. 5. Selection Cardinality Constraints on Table T

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, 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 three *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). Finally, to support the case of multi-table joins, we propose to build a directed graph according to the references between tables and perform a topological sorting on the graph. Then, we populate the *FK* column of each table in the topological order (§V-C).

#### IV. NON-KEY GENERATOR

In this section, we first introduce how to decouple *LCCs* into *UCCs* and *ACCs* (§IV-A). Then, we discuss how to use the bin packing method to deal with *UCCs*, including instantiating parameters in each *UCC* and generating non-key column data (§IV-B and §IV-C). Finally, we present how to instantiate parameters in each *ACC* (§IV-D).

##### A. Decouple Logical Dependencies

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



TABLE III  
ASSIGNED BOUNDARY VALUES FOR VIEW ELIMINATION

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

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

Each  $clause_k$  can be further represented by a disjunction of *literals*, and each sub-selection view  $V^k$  is decomposed as:

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

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. 5a, it has a logical predicate  $P=(t_1 \leq p_4 \vee t_2=p_5) \wedge t_1 - t_2 < p_6$ . Then, we can decompose it as follows.

$$\begin{aligned} V_9 &= \sigma_{t_1 \leq p_4 \vee t_2=p_5}(T) \cap \sigma_{t_1-t_2 < p_6}(T) &= V_9^1 \cap V_9^2 \\ V_9^1 &= \sigma_{t_1 \leq p_4}(T) \cup \sigma_{t_2=p_5}(T) &= V_9^{1(1)} \cup V_9^{1(2)} \end{aligned}$$

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:

$$\text{rule}_1 : V_i \cap V_j = V_j \text{ if } V_i \leftarrow U \quad \text{rule}_2 : V_i \cup V_j = V_j \text{ if } V_i \leftarrow \emptyset.$$

The **rule**<sub>1</sub> and **rule**<sub>2</sub> indicate that we can eliminate any sub/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 in related to the comparators in each sub/subsub-selection view's predicate. Table III lists the boundary values. 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 3.** Consider the cardinality constraint  $|V_9| = 1$  in Fig. 5a. Firstly, we observe that both  $V_9^1$  and  $V_9^2$  can be  $U$  if we set  $p_4 = +\infty$  and  $p_6 = +\infty$ . Then, we keep one sub-selection view and eliminate the other one. Suppose  $V_9^2$  is eliminated by setting  $p_6 = +\infty$ . Secondly, we try to set subsub-selection views of  $V_9^1$  as  $\emptyset$ . Specifically, both  $V_9^{1(1)}$  and  $V_9^{1(2)}$  can be  $\emptyset$  if we set  $p_4 = -\infty$  and  $p_5 = null$ . Then, we eliminate one subsub-selection view. Suppose  $V_9^{1(2)}$  is eliminated, then we simplify the cardinality constraint from  $|V_9| = 1$  to  $|V_9^{1(1)}| = 1$ .

Note, if all the predicates in a *clause* only contain comparators of *not in*, *not like* and  $\neq$ , then none of the subsub-selection view can be set as  $\emptyset$  (see Table III). To address this issue, we make use of the *De Morgan's law* [28] shown in **rule**<sub>3</sub> 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. 5a. Its universal set size is  $|T| = 8$ . Then, we can convert the cardinality constraint of  $|V_{10}| = 5$  as  $|\sigma_{t_1=p_7}(T) \cap \sigma_{t_2=p_8}(T)| = 3$ .

$$\text{rule}_3 : |V^{k(1)} \cup \dots \cup V^{k(m)}| = n \Leftrightarrow |\overline{V^{k(1)}} \cap \dots \cap \overline{V^{k(m)}}| = |U_V| - n$$

Theorem 1 shows that our elimination procedure can reduce a selection view  $V$  to its subsub-selection view  $V^{k(i)}$  (Case1) or the conjunction of  $\omega$  unary views  $\bigcap_{j=1}^{\omega} V_e^j$  (Case2) whose comparators are *in*, *like* or  $=$ . Note that Case2 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 distribution for each column based on its cardinality constraints (§IV-B). Then, after all the non-key columns are generated, we add a post-processing step to bound these values into same rows (§IV-C). Fig. 5b shows the elimination result of selection views  $V_4$ ,  $V_7$ ,  $V_9$  and  $V_{10}$ .

**Theorem 1.** Given a selection view  $V$ , our elimination procedure can reduce it to one of the following two cases:  $V^{k(i)}$  or  $\bigcap_{j=1}^{\omega} V_e^j$ , 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  $\omega$  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 III, 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  $\bigcap_{j=1}^q V^{i_j}$ , where all the literals in the  $i_j^{th}$  clause (i.e.,  $clause_{i_j}$ ) only contain comparators of *in*, *like* and  $=$ . As a result, in the second step, each subsub-selection view of  $V^{i_j}$  can be as  $\emptyset$ . Next, *Mirage* only keeps one subsub-selection view for each  $V^{i_j}$ , and finally reduces  $V$  to  $\bigcap_{j=1}^q V_e^j$ .

If  $q = 0$ , the first step would reduce  $V$  to a single sub-selection view  $V^k$ . Suppose there exist  $s$  subsub-selection views of  $V^k$  which cannot be set as  $\emptyset$ . If  $s = 0$ , the second step of *Mirage* would reduce it to  $V^{k(i)}$ . Otherwise, it reduces  $V^k$  to  $\bigcup_{j=1}^s V^{k(i_j)}$ , where the  $i_j^{th}$  literal (i.e.,  $literal_{i_j}$ ) in  $clause_k$  only contains the comparator of *not in*, *not like* and  $\neq$ . From **rule**<sub>3</sub>, we can see that  $\bigcup_{j=1}^s V^{k(i_j)}$  can be converted to  $\bigcap_{j=1}^s \overline{V^{k(i_j)}}$ , where  $\overline{V^{k(i_j)}}$  is a unary view whose comparator is *in*, *like* or  $=$ .

The theorem is proved.  $\square$

## B. Solve Unary Selection Operators

As each *UCC* involves a single column, we propose to use *UCCs* 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 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  $A$ ). We assume that all the parameters in *UCCs* are integers in  $(0, |R|_A]$ . In this way, deriving *CDF* in the original domain space is converted to deriving *CDF* in the cardinality space. Note that, *Mirage* transfers the generated data into the expected data type by a data transformer as defined in [18].

**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  $p$ . 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 *UCCs* (see Table III) into two kinds of comparators, which are  $=$  and  $\leq$ . Specifically, the *UCC* with a comparator *in* can be transformed into the union of multiple *UCCs* with the  $=$  comparator. Similarly, a *UCC* with a comparator *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* [28], the comparators  $>$ ,  $\geq$  and  $\neq$  can be easily converted to  $\leq$  and  $=$ . For example,  $|\sigma_{A>p}(R)|=k$  is equal to  $|\sigma_{A\leq p}(R)|=|R|-k$ . Then, we use the two kinds of *UCCs* 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 *UCCs* on column  $t_1$  in Fig. 5b. 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 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  $A$ .

**(1) 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 *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 in column  $A$ 's *UCCs* whose comparators can be converted to  $\leq$ . After ordering, the partial order of these  $n$  parameters is represented by  $p_{h_1}, \dots, p_{h_n}$ , 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  $f_A(p)$ .** In the step, we put each parameter  $p$  in  $f_A(p)$  into one of the  $n+1$  ranges properly. Specifically, suppose we would put  $k$  parameters  $p_{e_1}, \dots, p_{e_k}$  into the range  $(p_i, p_j]$ , we should guarantee that the sum of data existence probabilities of  $p_{e_1}, \dots, p_{e_k}$  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 a bin packing problem, which is *NP-complete* [29]. We propose a greedy method to address it. 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,  $p_{e_1}, \dots, p_{e_k}$  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. 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 its  $f_A(p')=f_A(p)$ . If so, we specify  $p=p'$ . Otherwise, we transform  $f_A(p)=x$  into  $f_A(p_1)=x_1, \dots, f_A(p_q)=x_q$ , with  $x_1 + \dots + x_q = x$ . This can be implemented by converting the *UCC* from  $|\sigma_{A=p}(R)|=x \cdot |R|$  to  $|\sigma_{A \text{ in } (p_1, \dots, p_q)}(R)|=x \cdot |R|$ . Obviously, the *UCC* with a smaller granularity of cardinality requirement is more likely to be satisfied. After putting  $k$  parameters  $p_{e_1}, \dots, p_{e_k}$  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, 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  $(p_i, p_j]$  (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 locates at or before  $p_i$ .

**Example 4.** Consider the column  $t_1$  of table  $T$  in Example 2, 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 *UCCs* and  $|T| = 8$ .

An example is shown in Fig. 6. The first step is to sort the pair  $(F_{t_1}(p), p)$  for each parameter  $p$  in  $F_{t_1}(p)$ . As  $F_{t_1}(p_4) < F_{t_1}(p_2)$ , then we have the partial order of parameters as  $p_4 < p_2$  and the cardinality space can be divided into three ranges  $(0, p_4]$ ,  $(p_4, p_2]$  and  $(p_2, 5]$ , where  $F_{t_1}(0, p_4) = 12.5\%$ ,  $F_{t_1}(p_4, p_2) = 12.5\%$  and  $F_{t_1}(p_2, 5) = 75\%$ . Next, our second step is to find a proper range for each parameter  $p$  in  $f_{t_1}(p)$ . In our case, there exists only one  $f_{t_1}(p_7)$  and only the third range has a data existence probability that is greater than

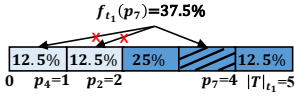


Fig. 6. CDFs of Column  $t_1$

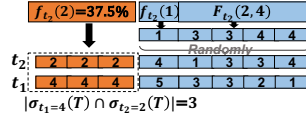


Fig. 7. Generate Table  $T$

$f_{t_1}(p_7)$ , 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  $|T| \cdot F(0, p_4) = |T| \cdot F(p_4, p_2) = |T| \cdot F(p_7, 5) = 1$ , 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  $p_4 = 1$ ,  $p_2 = 2$  and  $p_7 = 4$ . Fig. 5c shows the CDFs for non-key columns  $t_1$  and  $t_2$  after processing  $t_2$ .

### C. 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 §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. 7, column  $t_2$  is populated with  $3 = |T| \cdot f_{t_2}(2)$  rows valued 2 based on the CDFs in Fig. 5c. 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. 7, we assign  $4 = |T| \cdot F_{t_2}(2, 4)$  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, our next step is to arrange them in each table. Recall that our decoupling procedure might reduce a selection view  $V$  into  $\bigcap_{j=1}^{\omega} V_e^j$ , where  $V_e^j$  is a unary view whose comparator is  $in$ ,  $like$  or  $=$ . Thus, the  $\cap$  operator and  $=$  comparator in the cardinality constraint  $|\bigcap_{j=1}^{\omega} V_e^j| = n$  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  $\bigcap_{j=1}^{\omega} V_e^j$ . For example in Fig. 7, based on the requirement  $|\sigma_{t_1=p_7=4}(T) \cap \sigma_{t_2=p_8=2}(T)| = 3$  from  $V_{10}$ , we bind 3 rows valued (4, 2). Then, we populate the rest generated data into rows randomly to satisfy the other selection constraints on the

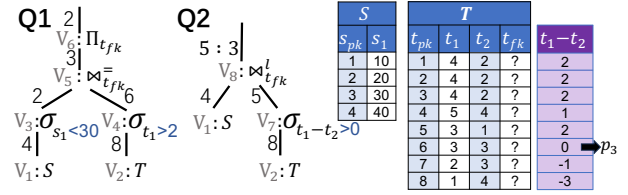


Fig. 8. Non-key Column Data and Instantiated Queries

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.

### D. 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  $ACC |\sigma_{g(A_i \dots A_j) \bullet p}(R)| = n$ , where  $g()$  is an arithmetic function,  $A_i, \dots, A_k$  are the non-key columns operated by  $g()$ , and  $\bullet$  is a comparator. We first calculate the result view  $g(A_i \dots A_j)(R)$  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. 5b. We first calculate the result view  $V_{rs} = g(t_1, t_2) = t_1 - t_2$  based on the populated data in  $t_1$  and  $t_2$ . As the comparator is  $>$ , we compute the result as shown in Fig. 8 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 [30], the number of sampled rows can be calculated as  $\frac{\ln 2 - \ln(1-\alpha)}{2\delta^2}$  according to the given error bound  $\delta$  and confidence level  $\alpha$ .

After solving all the  $ACC$ s, we have finished generating non-key column data and instantiating all their parameters. Next, we transfer the generated data and the instantiated query templates containing join operators to *key generator* for populating foreign keys. For example, the parameters in  $Q_1 \sim Q_4$  are instantiated as following; then we transfer the generated table  $S$  and  $T$ , and the instantiated  $Q_1$  and  $Q_2$  in Fig. 8 to *key generator* for populating  $t_{fk}$ .

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

## V. KEY GENERATOR

In this section, we first present how to deal with join cardinality and distinct constraints ( $JCC$ & $JDC$ ) on a single join in §V-A. Then, we discuss how to fuse multiple  $JCC$ s& $JDC$ s on the same two tables without conflicts in §V-B. Finally, we extend the solution to multiple tables in §V-C.



### A. Single Join Constraint on Two Tables

As defined in §II-B, a join view  $\bowtie^{type}(V_l, V_r)$  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 primary key of a row in  $V_l$  equals the foreign key of another row in  $V_r$ ). Its *JDC* requires that there exactly exist  $n_{jdc}$  distinct  $pk/fk$  values in all matched 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  $|V_r| - n_{jcc}$  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  $PF_{\bar{V}_l \rightarrow V_r} = |V_r| - n_{jcc}$ ). In addition, *JDC* imposes an additional constraint which requires that there exist  $n_{jdc}$  distinct foreign keys in all matched pairs 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  $PF_{V_l \rightarrow V_r}^d = n_{jdc}$ ). To summarize, we derive three foreign key populating rules from join cardinality and distinct constraints in Equation 1.

$$PF_{V_l \rightarrow V_r} = n_{jcc} \quad PF_{\bar{V}_l \rightarrow V_r} = |V_r| - n_{jcc} \quad PF_{V_l \rightarrow V_r}^d = n_{jdc} \quad (1)$$

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 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.** Consider the join view  $V_5 = \bowtie_{t_{fk}}^-(V_3, V_4)$  in Fig. 8. From its join cardinality constraints  $n_{jcc}=3$ , there exist 3 matched pairs of rows from  $V_3$  and  $V_4$ . Additionally, from  $|V_6| = |\Pi_{t_{fk}}(V_5)| = 2$ , we can infer that 2 distinct  $s_{pk}/t_{fk}$  in matched rows, i.e.,  $n_{jdc}=2$ . Based on Eqn. 1, the three foreign key populating rules for  $V_4$  are listed as follows.

$$PF_{V_3 \rightarrow V_4} = 3 \quad PF_{\bar{V}_3 \rightarrow V_4} = |V_4| - n_{jcc} = 3 \quad PF_{V_3 \rightarrow V_4}^d = 2$$

Fig. 9a 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  $PF_{\bar{V}_3 \rightarrow V_4}^d = 2$ , we select 2 primary keys in  $V_3$  to populate 3 foreign keys in  $V_4$ . As shown in Fig. 9a, we can populate the first three foreign keys in  $V_4$  as 1, 2 and 2. According to the rule  $PF_{\bar{V}_3 \rightarrow V_4} = 3$ , we should populate the rest foreign keys in  $V_4$  by any primary key in  $\bar{V}_3$ , e.g. 3 in  $\bar{V}_3$ . Finally, since  $\bar{V}_4$  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.

### B. 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, consider the row in  $T$  with primary key 1, its foreign key is populated as 1 in Fig. 9a and 2 in Fig. 9b. To address this issue, one naive method is to firstly enumerate all possible foreign key population results, and then find a feasible result which does not lead to any conflict. However, it's computationally prohibitive to enumerate population results for two joined tables. Specifically, for each foreign key in the referencing table  $T$ , it can be populated by any one of the primary keys in the referenced table  $S$ . Thus, there exist  $|S|$  population results for each foreign key. Considering that the referencing table has  $|T|$  foreign keys, then the total number of population results is  $|S|^{|T|}$ .

Note that, there exist some overlaps of primary keys between the input left child views. For example, consider the left child views  $V_1$  and  $V_3$  in Fig. 9, 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 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  $V_l^k$  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, 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 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 6.** Fig. 10 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  $V_8 = \bowtie_{t_{fk}}^l(V_1, V_7)$  in Fig. 8. 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 two rows have the same value, we put them into the same partition. For example, consider table  $S$  in Fig. 10. The status vectors of its 4 rows are (1, 1), (1, 1), (1, 0) and (1, 0), respectively.

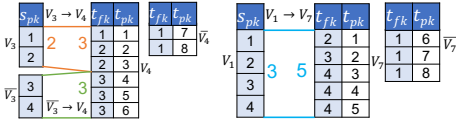


Fig. 9. Populate  $t_{fk}$  Based on Populating Rules

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

Fig. 10. Partitions of  $S$  and  $T$  According to Status Vectors

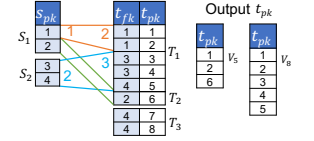


Fig. 11. Unified Population for  $t_{fk}$

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. Specifically, 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 §III), 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 §IV-A, 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  $\alpha$  columns in the partitioned table, and each of these columns is associated with  $x_1, \dots, x_\alpha$  unary selection views, s.t.  $\sum_{i=1}^{\alpha} x_i = j$ . Then, according to our cumulative distribution function based analysis described in §IV-B, 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  $\prod_{i=1}^{\alpha} (x_i + 1)$  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  $\prod_{i=1}^{\alpha} (x_i + 1) \cdot 2^{m-j}$ . 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.,  $\alpha$  is small). For example, the unary selection views cover at most 4 columns in any given table in the TPC-H workload. Thus, the actual number of partitions is much less than  $2^m$ .

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

view  $V^k$ , key population from  $V_l^k$  to  $V_r^k$  is

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

Based on Equation 2, we 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  $T_j$ .

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

In addition, from Equation 2, we can convert three populating rules on a join view in Equation 1 into the ones on their associated partitions. Here,  $n_{jcc}^k$  and  $n_{jdc}^k$  denote the join cardinality and distinct constraints on the  $k^{th}$  join view.

$$\begin{aligned} PF_{V_l^k \rightarrow V_r^k} &= \sum_{S_i \subseteq V_l^k, T_j \subseteq V_r^k} PF_{S_i \rightarrow T_j} = n_{jcc}^k \\ PF_{V_l^k \rightarrow V_r^k}^d &= \sum_{S_i \subseteq V_l^k, T_j \subseteq V_r^k} PF_{S_i \rightarrow T_j}^d = n_{jdc}^k \\ PF_{V_l^k \rightarrow V_r^k} &= \sum_{S_i \subseteq V_l^k, T_j \subseteq V_r^k} PF_{S_i \rightarrow T_j} = |V_r^k| - n_{jcc}^k \end{aligned} \quad (4)$$

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  $V_r^k$ , *Mirage* avoids selecting the primary key repeatedly to populate different partitions in  $V_r^k$ . If not, we would have  $\sum_{S_i \subseteq V_l^k, T_j \subseteq V_r^k} PF_{S_i \rightarrow T_j}^d < n_{jdc}^k$ .

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.

$$\sum_{S_i \subseteq S} PF_{S_i \rightarrow T_j} = |T_j| \quad (5)$$

**Example 7.** Consider the join view  $V_5 = \bowtie_{t_{fk}} (V_3, V_4)$  in Example 5 again. From Fig. 10, we can see that  $V_3 = S_1$ ,  $V_4 = T_1 \cup T_2$ . Based on Equation 4, the three populating rules in Example 5 can be converted as

$$\begin{aligned} PF_{V_3 \rightarrow V_4} &= PF_{S_1 \rightarrow T_1} + PF_{S_1 \rightarrow T_2} = 3 \\ PF_{V_3 \rightarrow V_4} &= PF_{S_2 \rightarrow T_1} + PF_{S_2 \rightarrow T_2} = 3 \\ PF_{V_3 \rightarrow V_4}^d &= PF_{S_1 \rightarrow T_1}^d + PF_{S_1 \rightarrow T_2}^d = 2 \end{aligned}$$

Based on Equation 5, the two following equations makes sure that the foreign keys in  $T_1$  and  $T_2$  are exactly covered.

$$PF_{S_1 \rightarrow T_1} + PF_{S_2 \rightarrow T_1} = |T_1| = 5 \quad PF_{S_1 \rightarrow T_2} + PF_{S_2 \rightarrow T_2} = |T_2| = 1$$

**(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 [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.

- **Composability.** The number of populated foreign keys must not be less than the number of populated distinct foreign keys, i.e.,  $PF_{S_i \rightarrow T_j} \geq PF_{S_i \rightarrow T_j}^d$
- **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.,  $PF_{S_i \rightarrow T_j} > 0 \Rightarrow PF_{S_i \rightarrow T_j}^d > 0$
- **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.,  $|S_i| \geq \sum_{T_j \subseteq V_r^k} PF_{S_i \rightarrow T_j}^d$

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 8.** Consider the join constraints from two join views  $V_5$  and  $V_8$ . After we include the above three constraints, we can find a feasible solution without contradictions as follows.

$$\begin{array}{cccc} PF_{S_1 \rightarrow T_1} = 2 & PF_{S_1 \rightarrow T_2} = 1 & PF_{S_2 \rightarrow T_1} = 3 & PF_{S_2 \rightarrow T_2} = 0 \\ PF_{S_1 \rightarrow T_1}^d = 1 & PF_{S_1 \rightarrow T_2}^d = 1 & PF_{S_2 \rightarrow T_1}^d = 2 & PF_{S_2 \rightarrow T_2}^d = 0 \end{array}$$

Fig. 11 presents the result of populating foreign keys based on this solution. Specifically, from  $PF_{S_1 \rightarrow T_1} = 2$  and  $PF_{S_1 \rightarrow T_1}^d = 1$ , 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.

Note that *CP* Problem may be unsolvable due to the constraints are 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.

### C. 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 [31], 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 [31]. 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 regarding  $T$  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 §V-B, we use these primary keys to populate the foreign keys in another table referencing  $T$ .

## VI. DISCUSSION

### A. 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 §IV-A, 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 2 proves that our data generation and parameter instantiation methods based on *CDF*s (see §IV-B and §IV-C) 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 §IV-D), 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  $\delta$  in a confidence level  $\alpha$  if the sampling data size is no less than  $\frac{\ln 2 - \ln(1-\alpha)}{2\delta^2}$ .

**Theorem 2.** 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 §IV-B. First, we transfer *UCC* into distribution requirements  $F_A(p)$  or  $f_A(p)$  in §IV-B, 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 probabilities from preceding sub-ranges, i.e., the following equation. Then, the step dose not introduce any error for  $F(p)$ .

$$\begin{aligned} \sum [F_A(p_i) - F_A(p_j)] &= [F_A(p) - F_A(0)] = F_A(p) \\ \text{s.t., } 0 &\leq p_i < p_j \leq p \ \& \ \nexists p \in (p_i, p_j) \end{aligned}$$

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.,  $F(p_i, p_k - 1) + F(p_k, p_j) =$

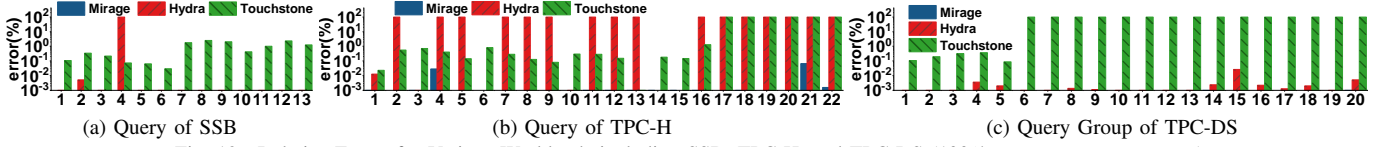


Fig. 12. Relative Errors for Various Workloads including SSB, TPC-H, and TPC-DS (100% error means no support)

$F(p_i, p_j) - f_A(p_k)$ . 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 §IV-C, 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.,  $|UCC| = |\hat{UCC}|$ .  $\square$

**Error Bound of the Key Column Generator.** As all the equations in §V-B 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 §V-A, 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.,

$$|V_l^i| \geq n_{jdc_i} \quad |V_r^i| \geq n_{jcc_i} \quad s.t. \quad n_{jcc_i} \geq n_{jdc_i} \quad (6)$$

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  $|V_r^i| \geq n_{jcc_i}$  in Equation 6, we can derive that relative error introduced by such a resizing is also bounded by  $\delta$ :

$$\frac{|(n_{jcc_i} - |\hat{V}_r^i|)|}{n_{jcc_i}} = |1 - \frac{|\hat{V}_r^i|}{n_{jcc_i}}| \leq |1 - \frac{|\hat{V}_r^i|}{|V_r^i|}| \leq \delta \quad (7)$$

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  $\delta$  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  $\delta$ , which can be adjusted considering the memory limitation (in §IV-D). In our experiment, it is verified to be an especially small number.

## VII. RELATED WORK

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

**Data-aware generators** [32]–[36] aim to generate a database closely related to the given data characteristics and the generation of a database instance is independent of workload. For example, *Alexander* [34] designs pseudo-random number generators and *Torlak* [32] uses kinds of multi-dimensional models. Although the generated database satisfies the given data distribution, the cardinality of each operator cannot be guaranteed [18]. **Query-aware generators** [13]–[19], [37] aim to generate a database that satisfies the cardinality constraints of required query plans. *QAGen* [13] 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* [14]. Then these works make a great effort to reduce the computational complexity rather than improve the support capability to operators. Following *QAGen*, *MyBenchmark* [16] 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* [19] 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 error. However, *Touchstone* cannot sample for the cardinality constraints from arbitrary logical predicates or support semi-/anti-join. *DCGen* [14] and *Hydra* [19] 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* [37] 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 error to be *zero* theoretically.



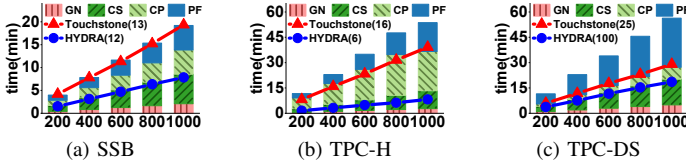


Fig. 13. Data Size vs. Generation Efficiency

## VIII. EXPERIMENT

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

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

**Workloads:** We compare *Mirage* with the two most recent work, i.e., *Touchstone* [18] and *Hydra* [19] with their provided codes. We select three classic benchmark scenarios, i.e., SSB [38], TPC-H [39] and TPC-DS [40], to provide the query plans and the cardinality constraints. SSB is a benchmark with 13 simple queries. TPC-H is the most popular OLAP benchmark with 22 queries having complex operators. In order to test the generation scalability, *Hydra* produces a scenario with large-scale workloads from TPC-DS, but removes its unsupported complex operators, e.g., *arithmetic* selections. In this workload, we take all the 100 distinct queries. Their official generation tools are used to compose the inputs of *Mirage* [38]–[40]. 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, i.e.,  $\frac{\sum ||V_i| - |\hat{V}_i||}{\sum |V_i|}$  in [18] to measure the accuracy for query  $Q$ .  $|V_i|$  represents the required cardinality of the  $i^{th}$  view  $V_i$  in query  $Q$ .  $|\hat{V}_i|$  represents the cardinality of  $\hat{V}_i$  in instantiated query  $\hat{Q}$  on the generated database. *Relative error* represents the cardinality deviation between  $Q$  and  $\hat{Q}$ , and the smaller is better.

**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%. In order to control memory usage, we use a batch generation strategy. Specifically, for each batch, we first generate non-key columns according to their CDF, and then populate foreign keys based on the join constraints whose values are scaled down according to the ratio between batch size and table 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.

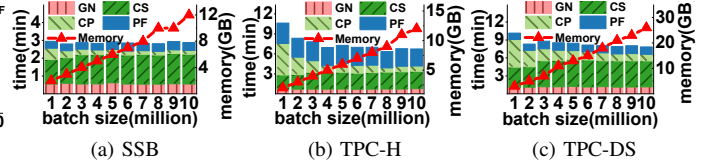


Fig. 14. Batch Size vs. Generation Efficiency

### A. Comparison of Workload Support Ability and Fidelity

We first compare the complex workload support ability as well as the generation fidelity with the state-of-the-art work, i.e., *Touchstone* and *Hydra*. We plot the relative errors for the benchmark queries in Fig. 12. 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* can give full support to generate the simple SSB application as in Fig. 12a and a TPC-H application with its first 16 queries as in Fig. 12b. But it can not scale well with the number of queries, which can generate a TPC-DS application with only 25 queries as in Fig. 12c. For the lack of operator support, even for SSB, *Hydra* can not process  $Q_4$  for generation and for TPC-H application, 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 relative error than *Mirage*. *Mirage* gives a strong support to almost all queries in these three applications. For the generation fidelity, it performs the best. Only for TPC-H, sampling-based parameter instantiation of ACC introduces errors, which are around 1%.

In summary, *Mirage* has the **widest** support of operators with **lower** errors and totally conquers its related work.

### B. 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 generation time of *Mirage* with *Touchstone* and *Hydra* in Fig. 13 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. So let us study the reason why *Mirage* is slower than *Hydra* on TPC-DS (in Fig. 13c). *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*, 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.

### C. Memory Consumption

Maintaining too much data dependency in memory, e.g., PK-FK dependency, has always bottlenecked the data generation efficiency [18], [19]. 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. Here, we illustrate the impact of batch size for generation by changing batch size from 1M to 10M rows in Fig. 14.

For a given 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 slender performance improvement from *CP*. Fig. 14 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 benchmarks. In such a setting, the maximum memory used is no more than 28GB.

In summary, *Mirage* can reach its peak performance with **conservative memory** usages decided by the batch size.

### D. 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. 15a). 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. 15b). 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 IV. The query templates from left column to right column in Table IV have a partial order relationship based on the 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 IV  
THE QUERIES AND INVOLVED TABLES

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

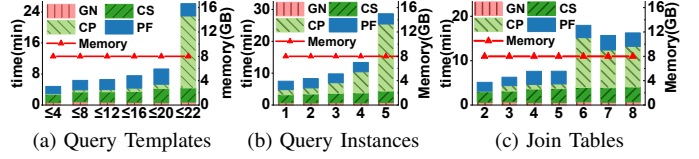


Fig. 15. Generation Scalability of *Mirage* Based on TPC-H

Based on Table IV, 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 <sup>1</sup> 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. 15c. 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. 15c, 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**.

## IX. CONCLUSION AND FUTURE WORK

To optimize query parallelism techniques, we propose *Mirage*, a query-aware data generator that provides the most powerful support to complex OLAP workloads. It achieves data generation in a linear way with *zero* error bound theoretically. Though *Mirage* performs the best among all the related work, it still could not cover all operators, e.g., non-PK-FK join, which will be left as our future work.

<sup>1</sup>TPC-H Benchmark: <http://www.tpc.org/tpch/>

## REFERENCES

- [1] C. Chekuri, W. Hasan, and R. Motwani, “Scheduling problems in parallel query optimization,” in *PODS*, 1995, pp. 255–265.
- [2] K. Goda, Y. Hayamizu, H. Yamada, and M. Kitsuregawa, “Out-of-order execution of database queries,” in *VLDB*, vol. 13, no. 12, 2020, pp. 3489–3501.
- [3] L. Landgraf, F. Wolf, A. Boehm, and W. Lehner, “Memory efficient scheduling of query pipeline execution,” in *CIDR*, 2022, pp. 9–12.
- [4] I. Sabek, T. S. Ukyab, and T. Kraska, “Lsched: A workload-aware learned query scheduler for analytical database systems,” in *SIGMOD*, 2022, pp. 1228–1242.
- [5] B. Wagner, A. Kohn, and T. Neumann, “Self-tuning query scheduling for analytical workloads,” in *SIGMOD*, 2021, pp. 1879–1891.
- [6] J. Wang and M. Balazinska, “Toward elastic memory management for cloud data analytics,” in *SIGMOD*, 2016, pp. 1–4.
- [7] Z. Yu, Z. Bei, and X. Qian, “Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing,” in *ASPLOS*, 2018, pp. 564–577.
- [8] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, “Learning scheduling algorithms for data processing clusters,” in *SIGCOMM*, 2019, pp. 270–288.
- [9] J. Cieslewicz, J. Berry, B. Hendrickson, and K. A. Ross, “Realizing parallelism in database operations: insights from a massively multithreaded architecture,” in *DaMoN*, 2006, pp. 4–es.
- [10] S. Manegold, M. L. Kersten, and P. Boncz, “Database architecture evolution: Mammals flourished long before dinosaurs became extinct,” in *VLDB*, vol. 2, no. 2, 2009, pp. 1648–1653.
- [11] D. J. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, “Gamma-a high performance dataflow database machine,” University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1986.
- [12] G. Graefe, “Volcano an extensible and parallel query evaluation system,” in *TKDE*, vol. 6, no. 1, 1994, p. 120–135.
- [13] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, “QAGen: generating query-aware test databases,” in *SIGMOD*, 2007, pp. 341–352.
- [14] A. Arasu, R. Kaushik, and J. Li, “Data generation using declarative constraints,” in *SIGMOD*, 2011, p. 685–696.
- [15] A. Gilad, S. Patwa, and A. Machanavajjhala, “Synthesizing Linked Data Under Cardinality and Integrity Constraints,” in *SIGMOD*, 2021, pp. 619–631.
- [16] E. Lo, N. Cheng, W. W. Lin, W. K. Hon, and B. Choi, “MyBenchmark: generating databases for query workloads,” in *VLDB*, vol. 23, no. 6, 2014, pp. 895–913.
- [17] E. Lo, N. Cheng, and W.-K. Hon, “Generating databases for query workloads,” in *VLDB*, vol. 3, no. 1–2, 2010, p. 848–859.
- [18] Y. Li, R. Zhang, X. Yang, Z. Zhang, and A. Zhou, “Touchstone: Generating enormous query-aware test databases,” in *USENIX ATC*, 2018, pp. 575–586.
- [19] A. Sanghi, R. Sood, J. R. Haritsa, and S. Tirthapura, “Scalable and dynamic regeneration of big data volumes,” in *EDBT*, 2018, pp. 301–312.
- [20] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.
- [21] L. Perron and V. Furnon, “Or-tools,” Google, 2021. [Online]. Available: <https://developers.google.com/optimization/>
- [22] S. J. Russell and P. Norvig, *Artificial intelligence a modern approach*. London, 2010.
- [23] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber, “Cache-efficient aggregation: Hashing is sorting,” in *SIGMOD*, 2015, p. 1123–1136.
- [24] S. Macke, M. Aliakbarpour, I. Diakonikolas, A. Parameswaran, and R. Rubinfeld, “Rapid approximate aggregation with distribution-sensitive interval guarantees,” in *ICDE*, 2021, pp. 1703–1714.
- [25] Z. Miao, A. Lee, and S. Roy, “Lensxplain: Visualizing and explaining contributing subsets for aggregate query answers,” in *VLDB*, vol. 12, no. 12, 2019, pp. 1898–1901.
- [26] Y. Diao, P. Guzewicz, I. Manolescu, and M. Mazuran, “Efficient exploration of interesting aggregates in rdf graphs,” in *SIGMOD*, 2021, pp. 392–404.
- [27] X. Liang, S. Sintos, Z. Shang, and S. Krishnan, “Combining aggregation and sampling (nearly) optimally for approximate query processing,” in *SIGMOD*, 2021, pp. 1129–1141.
- [28] I. M. Copi, C. Cohen, and K. McMahon, *Introduction to logic*. Routledge, 2016.
- [29] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.
- [30] W. Hoeffding, “Probability inequalities for sums of bounded random variables,” in *The collected works of Wassily Hoeffding*, 1994, pp. 409–426.
- [31] A. B. Kahn, “Topological sorting of large networks,” in *Commun. ACM*, vol. 5, no. 11, 1962, p. 558–562.
- [32] E. Torlak, “Scalable test data generation from multidimensional models,” in *SIGSOFT*, 2012, pp. 1–11.
- [33] E. Shen and L. Antova, “Reversing statistics for scalable test databases generation,” in *Proceedings of the Sixth International Workshop on Testing Database Systems*, 2013, pp. 1–6.
- [34] A. Alexandrov, K. Tzoumas, and V. Markl, “Myriad: scalable and expressive data generation,” in *VLDB*, vol. 5, no. 12, 2012, pp. 1890–1893.
- [35] N. Bruno and S. Chaudhuri, “Flexible database generators,” in *VLDB*, 2005, pp. 1097–1107.
- [36] K. Houkjaer, K. Torp, and R. Wind, “Simple and realistic data generation,” in *VLDB*, 2006, pp. 1243–1246.
- [37] P. Negi, L. Bindschaedler, M. Alizadeh, T. Kraska, J. Leeka, A. Gruenheid, and M. Interlandi, “Unshackling database benchmarking from synthetic workloads,” in *ICDE*, 2023, pp. 3659–3662.
- [38] P. O’Neil, E. O’Neil, X. Chen, and S. Revilak, “The star schema benchmark and augmented fact table indexing,” in *TPCTC*, 2009, pp. 237–252.
- [39] TPC-H, “TPC-H benchmark,” <http://www.tpc.org/tpch/>, 1999.
- [40] TPC-DS, “TPC-DS benchmark,” <http://www.tpc.org/tpcds/>, 1999.