Artemis: A Customizable Workload Generation Toolkit for Benchmarking Cardinality Estimation

***Abstract*—Cardinality Estimation (CardEst) is crucial for query optimization. Despite the remarkable achievement in DBMS, there is a pressing need to test or tune the work of CardEst. To satisfy the need, we introduce ARTEMIS, an open-source workload generator, which can be customized to generate various scenarios with the sensitive features for CardEst, including various data dependencies, complex SQL structures, and diverse cardinalities. It designs a PK-oriented deterministic data generation mechanism to plot various data characteristics; a genetic search-based workload generation is proposed for composing queries with various complexities; it takes a constraint optimization-guided way to achieve a cost-effective cardinality calculation. In this demonstration, users can explore the core features of ARTEMIS in generating queries.**

I. INTRODUCTION

Cardinality estimation (**CardEst**), renowned for the *Achilles Heel* of query optimization, is one of the critical tasks in query optimizer for estimating the intermediate result size of each operator, and deciding an optimal query plan and physical execution strategy. However, the intricate data dependencies and complicated selection/join operators still make CardEst an unsolved hard issue [1]. Traditionally, DBMSs use classical histograms to collect data statistics with low construction and maintenance costs. Due to high statistics storage cost, the histogram-based method is generally adopted based on some assumptions, e.g., uniformity within a histogram bucket and independence of attributes. To move beyond these assumptions, learning-based methods are leveraged to sketch the underlying data distribution and attribute correlations. Learning-based methods are mainly categorized into data-driven and query-driven. The data-driven method employs generative models for unsupervised learning of the proportion of rows in the joint domain represented as $P(D_M)$=$P(D_1, D_2, ..., D_n)$ [1], where D_M is the joint domain from the full outer join of all tables, and D_i is the sub-domain of the i^{th} attribute column. Supposing the size of table T as $|T|$, the new query as Q, the joint domain covered by Q as D_Q, the cardinality of query Q is $P(D_Q)*|T|$. The query-driven method uses discriminative models trained on annotated query-cardinality pairs $(Q, Card(Q))$ to predict the $Card(\bar{Q})$ for new query $\bar{Q}$. Though learning-based methods have demonstrated supreme capability on some classical benchmarks, e.g., TPC-H, IMDB-JOB, and STATS-CEB [2], it can not be generalized to obtain high accuracy across diversified scenarios, which makes it impossible for the practical industry use. However, providing diverse workloads for CardEst has always been a tough work. The core challenges come from the requirements of:

C1: Insufficient Data Dependency and Distribution. Both data-driven methods and traditional methods fundamentally try to identify joint domain with strong correlations and plot their joint distribution, which requires enormous datasets with various data dependencies and distributions. Traditional OLAP benchmarks, e.g. TPC-H, are meticulously designed for evaluating execution engines with uniform distribution and independence between attributes, which is too simple for the CardEst task. The other CardEst benchmarks [2] typically rely on a fixed schema using publicly available datasets, which have the defect of representativeness and generalizability; additionally, maintaining the established dependency relationships and data distributions when scaling data is still a great challenge.

C2: Demanding Training Label Generation. For query-driven methods, their effectiveness is fundamentally decided by the size of diverse annotated queries and cardinality label pairs for training, which can be generally obtained in two ways. One is to extract training data from real-life industry logs, but it is barely possible to collect such kinds of training data due to the concerns of privacy. The other one is to construct self-defined training data. However, it may consume a significant amount of CPU and memory resources, along with tedious and time-consuming manual effort to construct and run queries in a full-fledged system to gather labels. Therefore, the exhaustive resource consumption and burdensome manual effort for label generation challenge the model training.

C3: Limited Complexity of Workloads. The diversity of training workloads greatly influences the model robustness of learning-based methods, while the diversity of testing workloads verifies the effectiveness of methods across various scenarios. Workload diversity can be represented by three dimensions [2], i.e., 1) query templates composed of operators, 2) access distribution of the predicates, and 3) query cardinalities. On one hand, existing benchmarks have fundamental shortcomings in their template design. For instance, the state-of-the-art CardEst benchmark STATS-CEB loses support to some join types, including cycle/cyclic-joins, non-key joins, and outer/semi/anti-joins, and selection predicates like disjunctive logical predicates ($\vee$) and arithmetic predicates. On the other hand, existing benchmarks do not guarantee the generation diversity in predicate parameters or operator cardinalities, since they only take a randomization and manual inspection way to guarantee valid queries (cardinality > 0). Therefore, generating diverse query templates and instantiating parameters to fulfill arbitrary cardinality requirements remains a challenging and time-consuming task.

To tame these limitations and challenges, we propose a workload generation toolkit called ARTEMIS for CardEst with the following features:

- **Computation-driven Data Generation.** We design a PK-oriented deterministic data generation method by using a set of distribution functions to map from PKs to common attribute columns, with PKs playing as the seeds. By orchestrating the coefficients in the predefined map functions, we achieve plotting any type of data dependency and generating valid queries in a cost-effective way.
- **Constraint Optimization-guided Label Calculation.** We employ a lightweight automatic label generation method by modeling the cardinality calculation of predicates as a Constraint Optimization Problem (COP) with the predicates and the predefined data generation functions as constraints, which can be solved precisely by COP solver [3]. The computation-based approach does not burden the database for it abandons the iterative querying for cardinalities and can scale well with data sizes and query numbers.
- **Guided Search-based Workload Generation.** To accommodate diverse query generation, we first utilize a dynamic weighted random walk over a well-defined finite-state machine (FSM) to generate SQLs with various predicates; we then vectorize the generated query template, and compare it with the historical query template vectors to adapt the weight on FSM; finally, for each generated query template, we mutate or crossover each parameter with the dual objectives of minimizing both the distance from the specified cardinality requirement and the overlap in the accessed data ranges between the new query and historical queries.

II. DESIGN OVERVIEW

The architecture of ARTEMIS is shown in Fig. 1. Specifically, users configure their requirements about benchmarking scenarios, including schema (e.g., table number and table reference relationship); data characteristics (e.g., intra-column distribution and inter-column dependency); query type distribution (e.g., chain/cycle joins, the joined table numbers), and query cardinality. The generation has three phases: 1) it generates schema and data following the requirements of the data dependencies and distributions. 2) it constructs query templates having various query types and diverse predicates. 3) it instantiates predicate parameters for generating valid queries with different cardinalities across various access ranges.

A. Data Characteristic Sketch

For an effective testing, it is commonly required the generated query should access the data, called a valid query. However, scan-based value filling of parameters in the queries is too expensive for a large scale of query generation. Although we can deduce a valid parameter value in probability under a given data distribution $w.r.t$ a single column, the complex query semantics represented by the query templates, the predicates, and the intricate inter-column data dependencies make the valid parameter value deduction impossible [4].

To solve this problem, we employ a PK-oriented deterministic generation strategy, by which each non-key value can be well deduced from its corresponding PK value (abbr. PK)

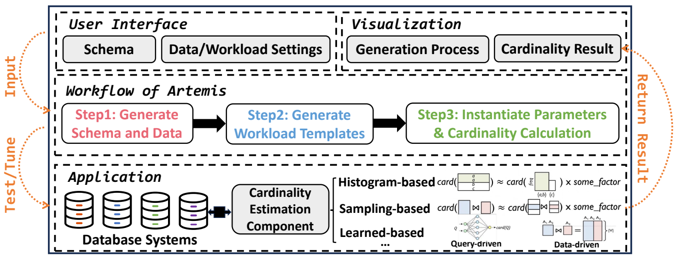


Fig. 1: ARTEMIS Architecture

under any intra-column distributions and inter-column data dependencies. Basically, each table T is associated with an automatic incremental PK column, and its non-key column, e.g., col_i, follows an intra-column data distribution ϕ_{col_i} and has the number of distinct values d_{col_i} from a specified domain ($d_{col_i} \leq |T|$). We propose to map PKs sequentially into d_{col_i} groups with the group id auto-increment and each group corresponds to a distinct value x in col_i. Let $g_k[PK]$ (abbr. g_k) represent the group id for an individual PK $w.r.t$ x_k in col_i and the number of PKs in each group follows the frequency of x_k calculated from ϕ_{col_i}. Since for a single column, a linear function can effectively guarantee the intra-data distribution by controlling the frequency of distinct value x_k, we then define the generation function from PK column to a non-key column col_i as $Func_{col_i}(g_k)$=$\alpha * g_k$+β, where coefficients α=$\frac{x_{max}\text{-}x_{min}}{d_{col_i}\text{-}1}$ and β=x_{min}-α with $x_{max/min}$ as the maximum/minimum values of col_i. Moreover, given a value x_k $w.r.t$ g_k, we can also derive its covered PK range. Specifically, according to ϕ_{col_i}, we have its cumulative distribution function (CDF) F_{col_i}. For the group g_k, its covered PK range is represented by the start PK (PK^s_k) and the end PK (PK^e_k), calculated as PK^s_k= $\lfloor |T| \times F_{col_i}(x_{k\text{-}1}) \rfloor$+1 and PK^e_k= $\lfloor |T| \times F_{col_i}(x_k) \rfloor$, among which $x_{k\text{-}1}$ and x_k are the values calculated by the generation function of col_i for $g_{k\text{-}1}$ and g_k, respectively. We thus can efficiently map g_k (i.e., x_k) back to its continuous PK range. Note that a column can be of any type, which can be achieved by adopting a commonly used type transformer from the generated integers, and data will be shuffled during the database population. Based on the generation functions, we introduce how to realize the four most typical inter-column data dependencies for CardEst [5], i.e., unique column combination (UCC), order dependency (OD), functional dependency (FD), and inclusion dependency (ID). For UCC, all records in the column must be unique, which can be well achieved by assigning each PK to a single group. For OD, it means the data of two columns follows the same order relationship, e.g., increasing or decreasing. Given that we have divided the PKs into groups ordered by group ids, so the dependency can be inherently satisfied by the above monotonic linear function based on group id g_k. For ID, it indicates that any value in column col_i will also be found in col_j ($i \neq j$). One example ID is between the referencing FK column (col_i with d_{col_i}) and the referenced PK column (col_j with d_{col_j}). Since $d_{col_i} \leq d_{col_j}$, if we take the same generation function for both col_i and col_j, the generated values for col_i must be covered by values in col_j. For FD, it means any records that have the same value in col_i will determinedly have the same value in col_j ($i \neq j$). We can achieve FD by taking the pair values in the two columns as the generation target. In such a way, we can guarantee FD. We can see that by organizing the PKs into groups corresponding to the distribution requirement of column col_i, the PK-oriented deterministic generation strategy can well guarantee the intra-column data distribution and inter-column data dependencies.

B. Workload Template Generation

For a thorough test, it requires to generate diverse query templates. To create a query template $\widehat{Q}$, we have two generation steps: 1) we determine its query type and the number of involved tables, then select a subset of table $\mathbb{T}_Q$ to construct the query template; 2) we compose the join predicates $\mathbb{J}$ and selection predicates $\mathbb{S}$ in the query. The basic form is like SELECT COUNT(*) FROM $\mathbb{T}_Q$ WHERE $\mathbb{J}$ AND $\mathbb{S}$. If there is a join, we first initialize an empty join graph and then iteratively update the graph by randomly selecting tables (nodes) to the graph. The nodes selected are decided by the join type and join number.

1) *Chain Query:* the new node can only connect with one of the nodes at either the start or end of the graph;
2) *Star Query:* the new node can only connect with the first node such that the first one becomes the center;
3) *Tree Query:* the new node can only connect with the node with neighbors no more than two;
4) *Cycle Query:* by searching cycles within the schema that satisfy the required number of tables, we put all nodes to the join graph all at once.
5) *Cyclic Query:* we first generate a cycle in the way of composing Cycle Query, and then the new node can connect with one node in the graph;

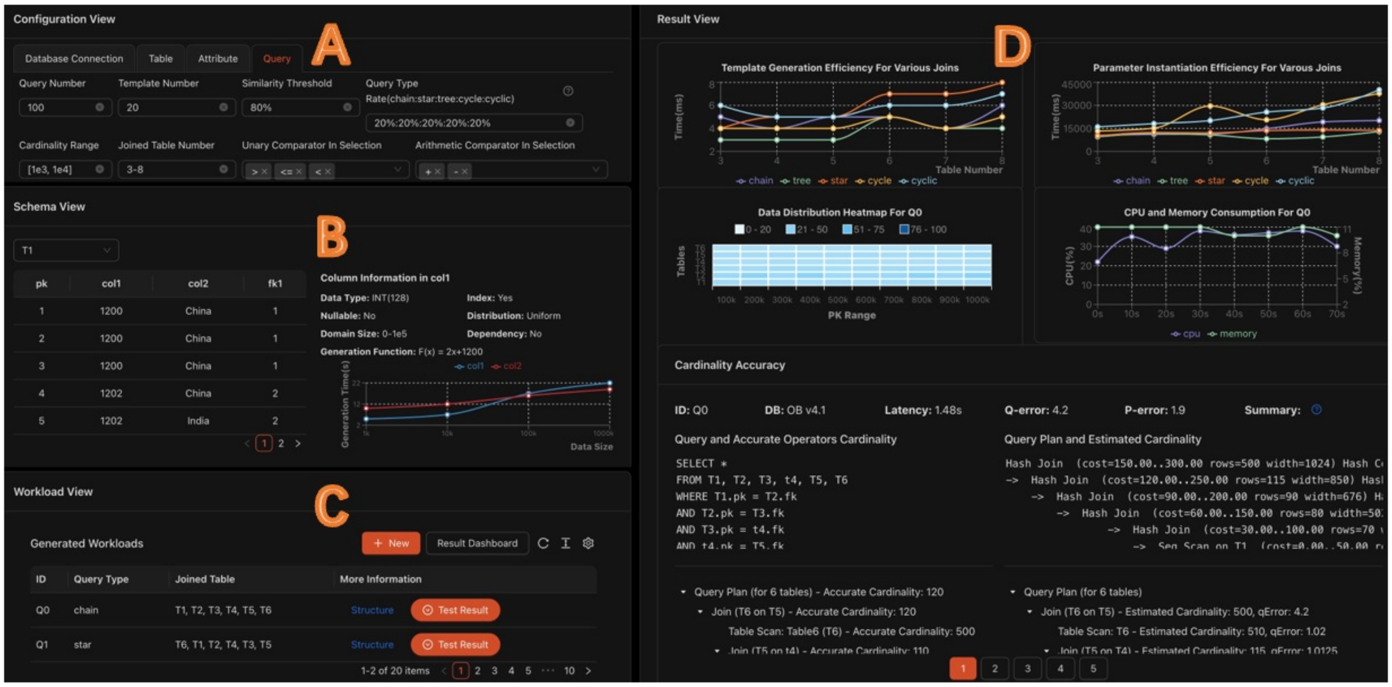
Next, we come to determine predicates for $\mathbb{J}$ and $\mathbb{S}$. $\mathbb{J}$ is formalized as $T_i.A_{i,p} \circ T_j.A_{j,q}$, where $A_{i,p}$ is the p^{th} attribute in T_i, and $\circ \in \{<, >, =, \neq, \leq, \geq\}$. For example, $T_i.PK$=$T_j.FK$ is the most common case of PK-FK equi-join. The $\mathbb{S}$ is composed of several clauses connected by the arbitrary logical predicate of $\wedge$ and $\vee$, each clause can be represented by any unary predicate or arithmetic predicates [6]. ARTEMIS constructs a complete SQL structure based on a well-defined finite-state machine (FSM) following SQL grammars, where the states include reserved keywords, table/attribute names, operations, etc. We conduct random walking over the FSM to generate syntax-complete SQL templates. The probability (weight) of each path for random walking is initially set to be uniform. To minimize the generation of similar templates, each generated query is embedded into a high-dimensional vector using the graph embedding strategy. For each new query, we calculate the cosine similarity between its embedding E^{cur} and historical embeddings $\mathbb{E}^{his}$, to identify the most similar historical query. We then adjust the weight of the generated path by $\hat{w}(=\frac{w}{1+similarity})$ and redistribute the weights of the other paths for normalization. In this way, ARTEMIS iteratively explores a broader query space. Meanwhile, when the number of historical queries reaches a threshold, we employ the classical outlier-insensitive K-Medoids clustering method [7] to select one query representing the class, thereby alleviating comparison costs.

C. Query Parameter Instantiation & Cardinality Calculation

For a customized test scenario, it is required to instantiate query templates to guarantee the expected cardinalities. So for a parameterized query template $\widehat{Q}$, ARTEMIS tryies to populate parameters in its predicates, i.e., $\hat{p_i}$ (i=$1 \cdots n$), targeting an expected output cardinality range r=$[L, R]$, where L and R are the range boundaries. If we can generate a population satisfying r, it is called a valid solution. Besides providing a valid solution, the parameter population aims to generate diverse access ranges and operator cardinalities for each $\widehat{Q}$, which is eagerly expected for learned-based methods to learn comprehensive data characteristics [8].

For this target, we design a multi-objective fitness function to evaluate the quality of the individual solution, measured by *cardinality gap* and *history data overlapping rate*. For an instantiated solution Q to $\widehat{Q}$, the *cardinality gap* (denoted as f_1) presents the gap between the real output from Q, i.e., $Card(Q)$, and the target r of $\widehat{Q}$, which is defined as f_1=$[max(0, L\text{-}Card(Q))$+$max(0, Card(Q)\text{-}R)] \cdot \frac{2}{L+R}$. The *history data overlapping rate* (denoted as f_2) quantifies the overlap of the selected keys from the current solution with those keys that have been covered by the historical solutions under the same template $\widehat{Q}$. Let $\mathbb{T}_Q$ represent the accessed tables in Q. Suppose that in the historical query set $\mathbb{Q}$ of $\widehat{Q}$, for any table T_i in $\mathbb{T}_Q$, keys that have been accessed are denoted by $\mathbb{D}_i^{his}$, and the current keys selected are denoted by $\mathbb{D}_i^{cur}$. We define f_2 as $\frac{1}{|\mathbb{T}_Q|} \cdot \sum_{i=1}^{|\mathbb{T}_Q|}(\frac{\mathbb{D}_i^{his} \cap \mathbb{D}_i^{cur}}{\mathbb{D}_i^{cur}})$, i.e., the average data overlapping rate across tables. So when instantiating parameters for diverse solutions to $\widehat{Q}$, we have the target to minimize both f_1 and f_2. Note that directly using a constraint programming solver to find feasible query parameters under specified query cardinality and templates is an NP-complete task [6]. To resolve this problem, ARTEMIS proposes to take the NSGA-II genetic algorithm, which has demonstrated supreme search efficiency, to perform a parameter search combined with the above multi-objective fitness function, and instantiate the parameter to p_i (i=$1 \cdots n$) on any expected data characteristics.

However, even if we can generate a valid query, checking whether the query meets the cardinality requirement of r and collecting the accurate cardinality label are still hard work. As we have mentioned, running generated queries $\mathbb{Q}$ on a full-fledged system to obtain cardinalities consumes significant resources, e.g., CPU and memory, which is not a feasible solution. Therefore, we design a lightweight computation method to expose cardinalities. Specifically, to calculate the cardinality of selection predicates, each table T has a selection predicate P. For ease of presentation, we mainly discuss the case in which P follows the disjunctive normal form (DNF), expressed as P=$\vee_{x=1}^{n} clause_x$ where $clause_x$=$\wedge_{y=1}^{m} literal_{xy}$. Note, any other form of predicate can be transformed to DNF [6].


A
B
C
D
Template Generation Efficiency For Various Joins
Time(ms)
Table Number
chain
tree
star
cycle
cyclic
Parameter Instantiation Efficiency For Varous Joins
Data Distribution Heatmap For Q0
0 - 20
21 - 50
51 - 75
76 - 100
Tables
PK Range
CPU and Memory Consumption For Q0
CPU(%)
Memory(%)
cpu
memory
Generation Time(s)
Data Size
col1
col2

Configuration View

Database Connection | Table | Attribute | Query

Query Number: 100
Template Number: 20
Similarity Threshold: 80%
Query Type Rate(chain:star:tree:cycle:cyclic): 20%:20%:20%:20%:20%
Cardinality Range: [1e3, 1e4]
Joined Table Number: 3-8
Unary Comparator In Selection: > <= <
Arithmetic Comparator In Selection: + -

Schema View

T1

pk	col1	col2	fk1
1	1200	China	1
2	1200	China	1
3	1200	China	1
4	1202	China	2
5	1202	India	2

Column Information in col1

Data Type: INT(128) **Index:** Yes
Nullable: No **Distribution:** Uniform
Domain Size: 0-1e5 **Dependency:** No
Generation Function: F(x) = 2x+1200

Workload View

Generated Workloads

+ New | Result Dashboard

ID	Query Type	Joined Table	More Information
Q0	chain	T1, T2, T3, T4, T5, T6	Structure Test Result
Q1	star	T6, T1, T2, T4, T3, T5	Structure Test Result

1-2 of 20 items

Result View

Cardinality Accuracy

ID: Q0 **DB:** OB v4.1 **Latency:** 1.48s **Q-error:** 4.2 **P-error:** 1.9 **Summary:**

Query and Accurate Operators Cardinality

```
SELECT *
FROM T1, T2, T3, t4, T5, T6
WHERE T1.pk = T2.fk
AND T2.pk = T3.fk
AND T3.pk = t4.fk
AND t4.pk = T5.fk
```

Query Plan and Estimated Cardinality

```
Hash Join  (cost=150.00..300.00 rows=500 width=1024) Hash C
->  Hash Join  (cost=120.00..250.00 rows=115 width=850) Hasl
    ->  Hash Join  (cost=90.00..200.00 rows=90 width=676) H
        ->  Hash Join  (cost=60.00..150.00 rows=80 width=50
                ->  Hash Join  (cost=30.00..100.00 rows=70
                    ->  Seq Scan on T1  (cost=0.00..50.00 r
```

- Query Plan (for 6 tables) - Accurate Cardinality: 120
  - Join (T6 on T5) - Accurate Cardinality: 120
    - Table Scan: Table6 (T6) - Accurate Cardinality: 500
    - Join (T5 on t4) - Accurate Cardinality: 110

- Query Plan (for 6 tables)
  - Join (T6 on T5) - Estimated Cardinality: 500, qError: 4.2
    - Table Scan: T6 - Estimated Cardinality: 510, qError: 1.02
    - Join (T5 on T4) - Estimated Cardinality: 115, qError: 1.0125

Fig. 2: The User Panel of ARTEMIS

The simplest case is when m=1 and n>0. In this condition, if the literal in $clause_x$ is a unary predicate over a single column col_i, it will filter out a continuous space represented by group ids, i.e., $g_{k_{max/min}}$. For instance, if $Func_{col_i}(g_k)$=α*g_k+β and the selection predicate is $col_i \leq p_1$, where p_1 is an instantiated value to parameter $\hat{p_1}$, then we have α*g_k+$\beta \leq p_1 \Rightarrow g_{k_{max}}$=$\frac{p_1\text{-}\beta}{\alpha}$. We can thus acquire the boundary value $PK_{max/min}$ for $g_{k_{max/min}}$ based on the F_{col_i} (see Step1). Finally, we have the selection cardinality PK_{max}-PK_{min}+1. If the literal in $clause_x$ is a non-unary predicate, such as col_i+col_j $> p_2$, we have $Func_{col_i}()$+$Func_{col_j}() > p_2$. We then use an off-the-shelf solver [3] to calculate a solution, i.e., group ids, from the two columns, which is used for calculating the corresponding PKs. Since n>0, we finally union the results from each clause for the final PK ranges that satisfy all predicates in selection.

A more complex case is m>1 and n>0. Similarly, in a single $clause_x$, we first batch all literals with unary predicates over individual attributes, i.e., $col_1 \ldots col_v$, into a group. Since each PK (row) corresponds to a group id of a column, PK can then be represented by a vector of group ids, i.e., $PK \rightarrow GV$ = $(g_1[PK], ..., g_v[PK])$. As mentioned before, for a unary predicate, a continuous space (represented by $PK_{max/min}$) is selected to satisfy the predicate. Since the literals in $clause_x$ are a conjunctive normal form (CNF), the result from each $clause_x$ is the conjunction of v continual space of PKs, which is still a continual space, represented by $PK_{min/max}$. Specifically, based on the GV, we can obtain the vectors satisfying all the unary literals of $clause_x$, which can be mapped back to the original PK domain. Since each group id is monotonic $w.r.t$ the PK, the sum of group ids of a row in GV is also monotonic. So, to find PK_{max} (resp. PK_{min}) is to find the row in GV having the maximum (resp. minimum) sum of group ids. Based on the observation, ARTEMIS models each literal including its involved generation function and selection predicate as a constraint (e.g., $Func_{col_i}() > p$), and the maximum/minimum sum of group id in GV as the optimization goal, which is a classic constraint optimization problem (COP). By employing the COP solver [3], it can precisely calculate $GV_{max/min}$. Note that, each g_i $(i \in [1, v])$ in GV_{max} can be mapped to its corresponding PKs_i. Since the literals in a clause are of pure CNF, PK_{max} is thus the smallest one from $PKs_1 \ldots PKs_v$. Similarly, we can get PK_{min} from GV_{min}. So the cardinality from the unary literals of a $clause_x$ is PK_{max}-PK_{min}+1. The solver can further handle the other literals in $clause_x$ to obtain complete solutions by intersecting results from all literals. The final result of predicate P is the union of results from all clauses. In such a way, we calculate to obtain the PKs from the leaf selection node on each table. With these PKs, we take a sort-merge join algorithm to calculate the matched rows (cardinality) of any type of joins. Given that data scanning is the most resource-intensive task in real-world applications [9], our COP-based cardinality calculation method significantly enhances the training label generation efficiency.

III. DEMONSTRATION SCENARIOS

We have developed a web-based user interface to illustrate the functionality of ARTEMIS. We deploy OceanBase v4.2.0 [10] to provide a demo scenario.

Scenario 1: Configuration View. ARTEMIS provides a panel to customize the requirement for the expected scenario as shown in Fig. 2(A). Database Setting connects a specific database; Table Setting configures the schema; Attribute Setting declares the metadata for data, e.g., data distributions, and dependency; Query Setting configures the workload, e.g., the scale of queries, join type ratio, the cardinality range, etc.

Scenario 2: Schema View. We display the metadata of the schema in Fig. 2(B). Users can access more detailed information by clicking on the table or attribute. For example, for 'col1', it displays its data distribution, the dependent attributes (if any), the generation function, etc. Besides, we can also view the data generation efficiency of each table.

Scenario 3: Workload View. It displays the generated query templates with instantiated parameters in Fig. 2(C). By clicking Test Result, it can expose the detailed query with true cardinalities and the query plan with estimated cardinalities as shown in the Cardinality Accuracy pannel of Fig. 2(D).

Scenario 4: Result View. Clicking the Result Dashboard in Fig. 2(C), the overall generation efficiency is shown in Fig. 2(D), i.e., the average query template generation time $w.r.t$ join types on different join table sizes and the corresponding parameter instantiation time. For each query, it also monitors the access data distribution heatmap scattered among different key ranges and the resource consumption(e.g., 'Q0'). After clicking the Test Result of 'Q0', the gap between the true cardinality and the estimated cardinality of the query plan is presented by estimation error, i.e., *Q-error* and *P-error* [2].

IV. Experiment

We conduct some representative experiments on three mainstream databases: OceanBase (v4.3), TiDB (v7.1), and PostgreSQL (v15) to demonstrate the functionality of ARTEMIS and compare the effectiveness of the CardEst components in typical commercial systems.

Evaluation Procedure. First, we used tools to generate a specified schema and data, importing them into each of the three databases. Then, we executed the workload generated by the tools on each database, using the command of 'EXPLAIN' to obtain the query plans and operator cardinality estimates.

Configurations. We create ten tables and 100 queries in total. For single-table queries, one table was generated with data distributed in Uniform, Zipfian1, and Zipfian3 patterns, with each distribution containing 100,000 rows. For multi-table join queries, 10 tables were generated with data distributed in Uniform, Zipfian1, and Zipfian3 patterns, with each distribution containing 1,000,000 rows. The flexibility of the ARTEMIS enabled the rapid generation of diverse data types.

In single-table queries, we have the following key observations. When estimating the cardinality of single-table selections (i.e., single-column predicates without arithmetic operations), all three databases—OceanBase, PostgreSQL, and TiDB—can accurately estimate cardinality when the attribute type and value type are the same, for both uniform and skewed distributions. However, the three databases handle cases where the attribute type and value type differ in distinct ways. OceanBase utilizes a Magic Number for comparisons between floating-point types, whereas PostgreSQL employs a Magic Number for comparisons between integer and floating-point types. TiDB's handling of type mismatches is not explicitly mentioned, but it is suggested that its estimation may be influenced by its specific algorithms.

TABLE I: Magic Number of Two-sided Predicates

Two-side Predicate	OceanBase	TiDB	PostgreSQL
Uniform	11970	80000	500
Zipfian 1	11757	80000	500
Zipfian 3	11769	80000	500

TABLE II: Magic Number of One-sided Predicates

Two-side Predicate	OceanBase	TiDB	PostgreSQL
Uniform	35910	80000	33333
Zipfian 1	35269	80000	33333
Zipfian 3	35306	80000	33333

For IN predicates where the user input contains duplicate values, the experimental results show notable differences in estimation accuracy. OceanBase and TiDB both perform deduplication, resulting in more accurate cardinality estimates, as shown in Fig. 3. In contrast, PostgreSQL does not perform deduplication, leading to an underestimation of cardinality in such cases. When estimating cardinality for equality predicates (single-column predicates without arithmetic operations), TiDB demonstrates higher accuracy than both OceanBase and PostgreSQL, as shown in Fig. 3. This is attributed to TiDB's use of the Count-Min Sketch algorithm, which provides a minimal upper bound for the estimation, thereby enhancing its accuracy compared to the other two databases.

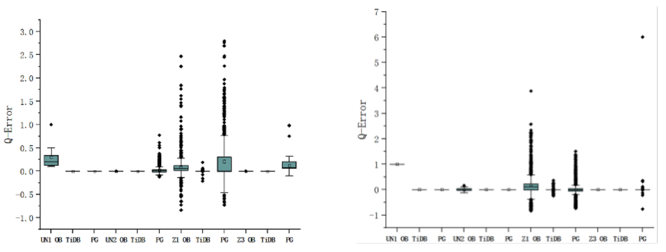


Fig. 3: Cardinality Estimation Error

When considering range predicates, OceanBase computes the Magic Number for two-sided predicates (e.g., 'between and') as the Magic Number for one-sided predicates divided by three, and the Magic Number for one-sided predicates (e.g., ¡) is calculated as approximately Table Size / 3. In TiDB, the Magic Number is determined as Table Size * (4/5), while PostgreSQL uses the same Magic Number for two-sided predicates as it does for equality predicates, and the Magic Number for one-sided predicates is Table Size / 3, as shown in Table I and Table II. Therefore, we do not demonstrate the error distribution, as it can exhibit significant randomness of result due to the fixed magic number.

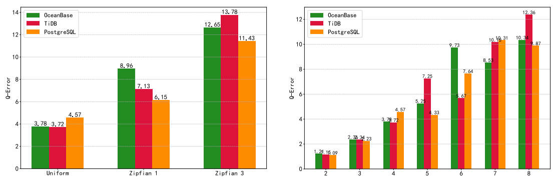
Regarding predicates with arithmetic operations, none of the three databases simplify the operations during cardinality estimation. The accuracy of cardinality estimation can be improved across all three databases when operations in predicates can be simplified (e.g., transforming col op val1 = val2 into col = val). In such cases, more accurate Magic Numbers are returned, resulting in better estimation outcomes.

In multi-table join queries, the cardinality estimation accuracy varies significantly across different query types and numbers of joins, as shown in Fig. 4. In the case of chain queries, when the number of joins is between 2 and 4, the estimation error ranges from 1 to 4 times. However, when the number of joins exceeds 5, the estimation error for all three databases increases to approximately 10 times. As the number of joined tables increases, the error for chain queries also grows progressively. For star queries, when the number of participating tables is less than or equal to 4, the estimation error for all three databases is approximately 3 to 4 times. However, when the number of joins reaches 5, there is a noticeable increase in the estimation error. In the case of cycle queries, both OceanBase and PostgreSQL return the cardinality estimate as a Magic Number (equal to 1). In contrast, TiDB provides relatively accurate cardinality estimates for circular queries. The results for cyclic queries are similar to those of cycle queries.

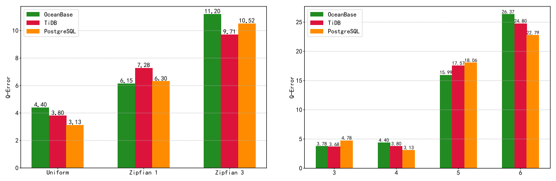
To conclude, the cardinality estimation accuracy for queries with different types shows significant variations across the databases. For common queries such as chain, tree, and star, the three databases are able to provide relatively accurate estimates when the number of joined tables is small. However, as the number of joined tables increases, the estimation errors also become larger. When dealing with queries involving cycles or cyclic joins, OceanBase and PostgreSQL tend to return a Magic Number, whereas TiDB is able to handle these cases more accurately. This discrepancy arises because handling cycle queries has long been a challenging problem. For cycle queries, the optimizer needs to consider the combined selectivity of multiple tables, which involves combining selection and join conditions across several tables and estimating the number of rows that satisfy these conditions.

V. Conclusion

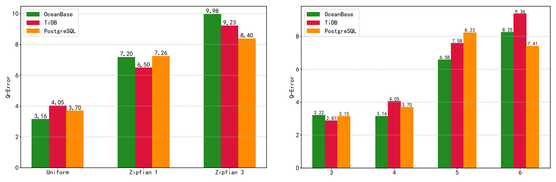
We introduce ARTEMIS to generate diverse Scenarios for testing or tuning CardEst, which is deployed on OceanBase database. The source codes and representative experiments on popular systems are all available on GitHub [11].



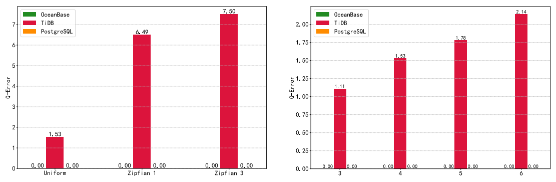
(a) Chain Query



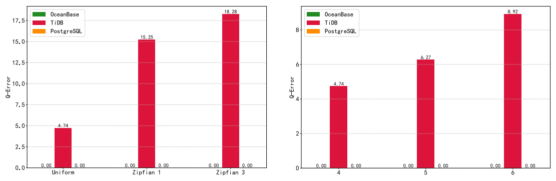
(b) Star Query



(c) Tree Query



(d) Cycle Query



(e) Cyclic Query

Fig. 4: Cardinality Estimation Error for Various Query Types

References

[1] Z. Wu and et al., “Factorjoin: a new cardinality estimation framework for join queries,” *SIGMOD*, vol. 1, no. 1, pp. 1–27, 2023.

[2] Y. Han and et al., “Cardinality estimation in DBMS: A comprehensive benchmark evaluation,” *VLDB*, vol. 15, no. 4, pp. 752–765, 2021.

[3] Gurobi, “Gurobi optimizer,” 2024. [Online]. Available: https://www.gurobi.com/solutions/gurobi-optimizer/

[4] K. Li and et al., “Dbstorm: Generating various effective workloads for testing isolation levels,” *ISSTA*, pp. 755–767, 2024.

[5] J. Kossmann and et al., “Data dependencies for query optimization: a survey,” *VLDBJ.*, vol. 31, no. 1, pp. 1–22, 2022.

[6] Q. Wang and et al., “Mirage: Generating enormous databases for complex workloads,” *ICDE*, pp. 3989–4001, 2024.

[7] P. Wu and et al., “Modeling shifting workloads for learned database systems,” *SIGMOD*, vol. 2, no. 1, pp. 1–27, 2024.

[8] K. Kim and et al., “Asm: Harmonizing autoregressive model, sampling, and multi-dimensional statistics merging for cardinality estimation,” *SIGMOD*, vol. 2, no. 1, pp. 1–27, 2024.

[9] A. van Renen and et al., “Cloud analytics benchmark,” *VLDB*, vol. 16, no. 6, pp. 1413–1425, 2023.

[10] Z. Yang and et al., “Oceanbase: a 707 million tpmc distributed relational database system,” *VLDB*, vol. 15, no. 12, pp. 3385–3397, 2022.

[11] Z. Hu, “Artemis supplementary materials,” DBHammer, 2024. [Online]. Available: https://github.com/DBHammer/Artemis-CardEst