Beyond Row Counts: Enhancing Workload-Aware Data Synthesis

Anupam Sanghi

Technische Universität Darmstadt, Germany

Abstract

Synthetic database generation is critical for testing and benchmarking database systems and applications. Current approaches focus on workload-aware data synthesis that ensures volumetric similarity, where the output row cardinalities of query operators closely match those of customer workloads. However, they often neglect critical features like data duplication and value ordering, which influence the performance of fundamental database operations like hashing and sorting. This work addresses this lacuna by incorporating two additional data characteristics: Duplication Distribution and Presortedness. We present (a) mathematical models for these characteristics, (b) techniques to extract them from query execution, and (c) strategies to mimic them in synthetic data generation. These enhancements aim to better simulate real-world database performance.

Keywords

Synthetic Data Generation, Workload-Aware Data Synthesis, Database Testing and Benchmarking, Data Duplication, Presortedness

1. Introduction

Workload-Aware Data Synthesis is Essential. In industrial practice, database vendors often perform tasks such as testing and benchmarking database systems and applications, data masking, and assessing the performance impacts of planned engine upgrades. These tasks require data that mirrors customer environments [1, 2]. However, transferring original client data is often impractical due to privacy concerns, making the use of workload-aware data generators essential [3].

Current Focus on Volumetric Similarity. Contemporary workload-aware data generators [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] utilize query execution plans derived from customer workloads to provide *volumetric similarity* [10], i.e., ensuring that the intermediate row cardinalities produced by query plans on synthetic data closely match those observed on the original data. This preserves data layout and flow during query execution.

Overlooked Characteristics. Despite its significance, volumetric similarity does not capture other crucial data characteristics, such as *data duplication*, *value ordering*, *data skew*, and *correlations*, which significantly affect query performance. SQL constructs like `JOIN`, `GROUP BY`, `DISTINCT`, and `UNION` rely heavily on hash-based computations and sorting operations, which are sensitive to factors like the *duplication* of values and the *presortedness* (i.e. the extent to which the data is already ordered). Excessive duplication can cause inefficient hash bucket usage, leading to spills and longer probe times, while partially sorted data reduces sorting complexity, improving execution speed by minimizing tuple movement and comparison costs.

Our Contributions. In this paper, we include additional data characteristics, namely *Duplication Distribution* and *Presortedness*, within the ambit of workload-aware data synthesis. Specifically, we contribute the following:

1. Case studies demonstrating the impact of these characteristics on query performance,
2. Mathematical modeling of these characteristics,
3. Techniques for extracting them from query execution, and
4. Initial strategies to mimic them in data synthesis.

By addressing these aspects, our work enhances the fidelity of synthetic data, enabling more accurate simulations of real-world database performance scenarios.

Organization. The paper is organized as follows: Section 2 presents case studies on the impact of Duplication Distribution and Presortedness on query performance. Sections 3 and 4 present the formal characterization, extraction methods, and integration strategies for Duplication Distribution and Presortedness, respectively. Section 5 concludes the paper and outlines future research directions.

2. Case Studies

2.1. Case Study 1: Data Duplication

Data duplication significantly impacts operations like hashing, commonly used in SQL constructs such as hash joins, group by, distinct, and union. To illustrate this, we created two datasets, D_1 and D_2, each containing two tables, *Student*(S) and *Register*(R), with identical row counts ($|R|$ = 655 million, $|S|$ = 82 million rows) across the corresponding tables. For simplicity, both tables have only one column, *SNo*, and *R.SNo* references *S.SNo* as a foreign key. In D_1, *R.SNo* has a uniform distribution on all values in *S.SNo*, while in D_2, *R.SNo* contains the same value for all rows. We executed the following SQL query on both datasets using identical hardware, database platform (a popular commercial engine), and system configuration.

```
Select * From R, S where R.SNo = S.SNo;
```

Although the query optimizer chose identical physical plans with hash joins and produced the same output cardinalities, execution times varied significantly – 18 min for D_1 and 28 min for D_2 (Table 1). The increased time for D_2 is due to spilling in the hash table computation, caused by data duplication. This underscores the importance of modeling Duplication Distribution in synthetic data generation.

DOLAP 2025: 27th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data, co-located with EDBT/ICDT 2025, March 25, 2025, Barcelona, Spain

anupam.sanghi@tu-darmstadt.de (A. Sanghi)
https://anupamsanghi.github.io/ (A. Sanghi)
0000-0003-4764-3583 (A. Sanghi)

Table 1
Query Execution Time for Different Data Duplications

Distribution Type	Running Time
D_1	18 min
D_2	28 min

2.2. Case Study 2: Presortedness

SQL operations such as order by, sort-merge joins, group by, distinct, and union often rely on sorting. The complexity of sorting depends on the tuple movements and number of comparisons. The degree of presortedness, or the order in the input data, directly influences this complexity. To demonstrate this, we used an instance of the *INVENTORY* table (8.4 GB, over 400 million tuples) from the TPC-DS [16] benchmark. We selected the column *inv_qty_on_hand* and created a new table $T(A, B)$ with one sorted and one randomized copy of the column. We then executed `ORDER BY ASC` and `ORDER BY DESC` queries on both columns. Table 2 shows the execution times, where Column Order indicates the existing order of the data, and Sort Order specifies the query's sorting direction. When the column order matched the sort order, no tuple movement was required, resulting in the shortest execution time.

3. Duplication Distribution (DD)

This section introduces a framework for Duplication Distribution (DD), covering its theoretical and computational aspects. It presents a pair-based representation for quantifying duplication, methods for measuring distance between DD representations, and techniques for extracting duplication information. It also explores initial strategies for mimicking DD in data generation.

3.1. Characterization

A DD, denoted as d, describes how often values are duplicated in a table T for a target set of columns C. It is represented as a set of pairs $\{(m, f)\}$, denoting that the number of distinct C values with multiplicity m is f. For example, the column values $[4, 2, 3, 1, 4]$ yield $d = \{(1, 3), (2, 1)\}$: three values $(1, 2, 3)$ appear once ($m = 1, f = 3$), and one value (4) appears twice ($m = 2, f = 1$).

The DD already captures the total row-cardinality information. This can be computed as: $\|d\| = \sum_{i=1}^{k}(m_i \times f_i) = |T|$, where $d = \{(m_1, f_1), (m_2, f_2), \ldots, (m_k, f_k)\}$, and k is the number of (m, f) pairs in d. Thus, ensuring that two tables have matching DDs inherently implies volumetric similarity.

Note that the DD captures the frequency distribution of value multiplicities, unlike histograms, which focus on the frequency of individual values. This allows DD to better account for data duplication without exposing data values. Furthermore, since histograms essentially capture row counts over a range query, existing work, such as [17], can integrate them into the data generation pipeline.

Table 2
Query Execution Time for Varied Column and Sort Orders

Column Order	Sort Order	Time (in min)
Ascending	Ascending	1.5
Random	Ascending	5.1
Ascending	Descending	3.9
Random	Descending	4.9

3.2. Distance Between DDs

Linearization. To compare two DDs, each is transformed into a one-dimensional array $\lambda(d)$. This is done by repeating each value m exactly f times and sorting in descending order. For example, $d_1 = \{(5, 1), (4, 2), (3, 1), (1, 2)\}$ becomes $\lambda(d_1) = [5, 4, 4, 3, 1, 1]$. This transformation simplifies the comparison while preserving the multiplicity distribution. When comparing DDs having linearizations of differing sizes, the shorter array is padded with zeros, reflecting that any additional values in the distribution have a frequency of zero. For $d_2 = \{(4, 4), (2, 1)\}$ compared to $\lambda(d_1)$, the padded form is $\lambda(d_2) = [4, 4, 4, 4, 2, 0]$. This ensures equal-length arrays, maintaining semantic fidelity while facilitating computation.

Distance Metric. The distance between two DDs is calculated as the normalized sum of absolute differences between their corresponding elements in the linearized arrays:

$$\Delta(d_1, d_2) = \frac{1}{2|T|} \sum_{i=1}^{|\lambda(d_1)|} |\lambda(d_1)[i] - \lambda(d_2)[i]| \tag{1}$$

For the above example, the distance is $\Delta(d_1, d_2) = \frac{|5-4|+|4-4|+|4-4|+|3-4|+|1-2|+|1-0|}{2\times 18} = 0.11$. The normalization factor ensures that Δ ranges between 0 (identical) and 1 (maximum disparity). The maximum possible distance occurs between two extremes: $\{(|T|, 1)\}$, where all values in T are identical, and $\{(1, |T|)\}$, where all values are distinct. The distance is $\Delta_{max} = 1 - \frac{1}{|T|}$, which approaches 1 as the table size increases.

This metric works effectively by first aligning the distributions in descending order and then comparing them element-wise. This ensures minimal dissimilarity, as matching the largest values first minimizes the difference.

3.3. DD Size

For scalability in data synthesis, the DD must remain compact. Its size, denoted as k, is determined by the number of distinct multiplicities for C values in T. The size is maximum for the case where $d_{max} = \{(1, 1), (2, 1), \ldots, (k, 1)\}$. Here, $\|d_{max}\| = 1 + 2 + \ldots + k = |T|$, leading to:

$$k = \mathcal{O}(\sqrt{|T|}) \tag{2}$$

Thus, even for a table with a trillion rows, the DD can be stored in just a few megabytes. Experimental results confirm this: for non-key columns in four tables from the 1 GB TPC-DS benchmark, the total size of DD vectors was under 40 KB. Table 3 summarizes the minimum, average, and maximum k values across these tables.

Scalable Approximation. To further enhance scalability, binning strategies approximate the DD by grouping similar multiplicities into fewer bins. Geometric means are used as bin representatives to minimize *q-error* [18], a common distance metric for cardinality estimation. Two alternative strategies can be employed for binning:

Table 3
DD vector Size

Table (#Rows in million)	k size Min., Avg., Max.	$\mathcal{O}(\sqrt{\lvert T \rvert})$
store_sales (2.6)	6, 257, 924	1620
catalog_sales (1.4)	6, 194, 864	1195
customer (0.1)	5, 24, 37	317
inventory (11.7)	1, 3, 5	3428

1. **Error Threshold**, which minimizes the number of bins while maintaining multiplicity error within a specified threshold ϵ. This greedy method (also optimal) groups multiplicities incrementally, creating a new bin whenever the distance between extreme multiplicities and the bin's mean exceeds ϵ; and
2. **Size Threshold**, which fixes the number of bins and minimizes error within this constraint. This approach reduces to one-dimensional k-means clustering, for which established techniques [19] can compute optimal bin boundaries.

These approximations balance accuracy and storage, ensuring DD's scalability for deployment, with the choice guided by priorities on error control or storage.

3.4. Extraction

Database systems expose input/output row cardinalities for operators in a query execution plan but lack duplication details. This necessitates DD extraction for *target operators*, who are sensitive to duplicates. We propose two strategies:

Offline Approach. This non-invasive approach computes the DD for C at the input of a target operator op using an SQL query. Two `GROUP BY` operations are performed: the first calculates the multiplicity of each distinct C value in table T (the input to op), and the second aggregates these multiplicities into the DD. The SQL query is:

```
Select m, count(*) as f From
 (Select C, count(*) as m FROM T Group By C)
Group By m;
```

Here, to capture the intermediate table serving as op's input, the inner query can include relevant constraints.

Online Approach. This dynamic method computes the DD incrementally during query execution using two structures: (a) `ValueMultiplicity`, tracking the multiplicity of each distinct C value, and (b) `MultiplicityFrequency`, counting values with specific multiplicity. As each row hits op, the multiplicity of its value is incremented in `ValueMultiplicity`. Simultaneously, `MultiplicityFrequency` is adjusted by decrementing the old count and incrementing the new one. This mirrors the offline approach, where the inner query computes value multiplicities, and the outer query aggregates them. Implementing this approach requires query executor modifications, enabling real-time updates of the DD during execution.

Performance Considerations. Since system testing is not a real-time activity, the offline approach remains viable. However, for complex queries or large datasets, the additional queries per target operator may pose scalability challenges. In such scenarios, the online approach can offer better performance. It can also leverage advancements in approximate frequency counting for streaming data [20], enabling rapid computations with minimal accuracy loss.

3.5. Mimicking

Mimicking duplication distribution is closely tied to satisfying projection constraints [21], which take the form $|\pi_A(\sigma_p(T_1 \bowtie T_2 \bowtie \ldots \bowtie T_N))| = c$. Here, $|\pi_A(\sigma_p(\cdot))|$ represents the count of distinct values in column-set A after applying a filter predicate p on the join of tables $T_1, T_2, \ldots, T_N$, constrained to equal a constant c. Projection constraints, in fact, are a special case of DD constraints, as the DD vector encapsulates both distinct counts and their multiplicities.

To highlight the key difference in incorporating DD into the data generation pipeline, this section focuses on the simpler case of single-column table synthesis. This approach can be extended to the more general case of constraints spanning multiple columns or overlapping column sets using techniques from [21, 22]. We now formally discuss the specific case under consideration.

Consider a single-column table C with a set of filter predicates P. For each predicate $p \in P$, let the corresponding DD of values satisfying p be d_p. The predicates in P can be used to partition the domain of C into a set of disjoint intervals I, where each interval is fully included in or excluded from each predicate [10]. Define a mapping $\phi(p) \subseteq I$ as the set of intervals $i \in I$ contained in the predicate p.

For each interval $i \in I$, we identify predicates in P that include i. For each multiplicity m common to the DDs of these predicates, a variable $x_{m,i}$ represents the number of values with multiplicity m in i. The DD $d_p = \{(m_1, f_1), (m_2, f_2), \ldots, (m_k, f_k)\}$ is expressed as a system of equations enforcing that the sum of variables corresponding to m_j across all intervals in $\phi(p)$ containing m_j equals f_j:

$$\sum_{i \in \phi(p)} x_{m_j,i} = f_j \quad \forall (m_j, f_j) \in d_p \tag{3}$$

Solvers like Z3 [23] can compute non-negative integral solutions to this linear feasibility problem. The solution provides the DD (d_i) for each interval i. To generate values for an interval i based on $d_i = \{(m_1, f_1), (m_2, f_2), \ldots, (m_k, f_k)\}$, we select $f_1 + f_2 + \ldots + f_k$ distinct values within i, generating m_1 copies for the first f_1 values, m_2 copies for the next f_2 values, and so forth.

4. Presortedness

This section formalizes the concept of Presortedness, presents a method to extract it from query execution, and outlines initial strategies for integrating Presortedness into data synthesis pipelines.

4.1. Characterization

Given a table T, let C denote the target set of columns defining the sorting criteria. To compute the degree of Presortedness of T with respect to C, we quantify how closely the values in C align with their sorted counterpart. Let X represent the original values in C and Y represent the fully sorted version of these values. The Spearman's rank correlation coefficient [24] captures the monotonic relationship between

Table 4
Execution Time of Order By Queries on various base tables and columns of TPC-DS 1 GB instance without and with Presortedness computation

Table Name	Column	Running Time	
(Row Count)	Name	original	with ρ
store (12)	store_name	0.1 ms	0.2 ms
customer_address (50K)	city	0.7 s	0.9 s
customer (100K)	first_name	0.9 s	0.9 s
store_sales (2.6M)	quantity	9 s	10 s
inventory (11.7M)	warehouse_sk	28 s	29 s

X and Y by computing the correlation between their respective rank transformations. In this case, the *rank* of a value is its position in the sorted array Y. Therefore, Presortedness ρ is given by:

$$\rho = \frac{\text{cov}(\text{rank}(X), \text{rank}(Y))}{\sigma_{\text{rank}(X)}\sigma_{\text{rank}(Y)}}, \tag{4}$$

where rank(X) and rank(Y) denote the ranks of the original and sorted values, cov represents covariance, and σ denotes standard deviation. When the values in C are distinct, the formula simplifies to:

$$\rho = 1 - \frac{6\sum_{i=1}^{|T|}(\text{rank}(X_i) - \text{rank}(Y_i))^2}{|T|(|T|^2 - 1)}. \tag{5}$$

The value of Presortedness ranges from -1 to 1. A value of 0 reflects maximum randomness in the arrangement of the data. A positive value suggests that more elements are closer to their sorted positions, whereas a negative value indicates greater deviation from sorted order.

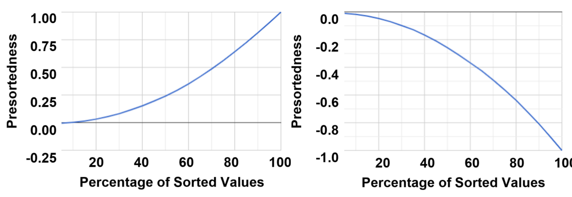
4.2. Extraction

To extract Presortedness for C used by the target sort operator *op* during query execution, we provide the input tuples (original array) to and output tuples (sorted array) from *op* to a Spearman's rank correlation coefficient calculator. The calculator computes the ranks using the sorted array and calculates Presortedness as described in Section 4.1.

We implemented the above strategy within the PostgreSQL engine. The time overheads incurred due to the additional code for Presortedness computation are shown in Table 4. The results indicate that the overheads are viable. A non-invasive extraction would require materializing the input and output tables of the sort operator and performing the same implementation outside the system.

4.3. Mimicking

To replicate Presortedness from the original data in synthetic data, we utilize the relationship between the percentage of sorted tuples and Presortedness.



(a) Ascending (b) Descending

Figure 1: Presortedness vs. Percentage of Sorted Tuples

Table 5
Comparing Expected vs Obtained Presortedness

#Tuples	Desired ρ	Obtained ρ
1000	0.53	0.58
10000	-0.67	-0.65
10000	0.12	0.13
100000	0.82	0.84

Presortedness vs. Percentage of Sorted Tuples. To establish this relationship, we begin with an array of n values ranging from 1 to n, which is shuffled to achieve a ρ value close to 0. Next, we incrementally select different percentages of the array, sort them, and replace the selected tuples in their original positions, but in sorted order. Specifically, if the tuples are selected from positions $i_1, i_2, \ldots, i_k$, the first tuple in the sorted order is placed at i_1, the second at i_2, and so on. This process is repeated for varying percentages of sorted tuples and different values of n, considering both ascending and descending order. The resulting relationship, illustrated in Figure 1 for $n = 10000$, shows similar behaviour for other values of n as well. The Presortedness for each percentage is averaged over different sets of selected tuples.

Mimicking Presortedness. To achieve the desired Presortedness in a table T, we sort the required percentage of tuples. This percentage can be determined using the inverse of the established relationship between sorted tuples and Presortedness or by applying binary search, as the relationship is monotonic. In our experiments, we implemented the binary search that iteratively adjusts the percentage to match the desired Presortedness. For each percentage, the selected tuples are chosen randomly. The results, comparing the desired and obtained Presortedness values, are shown in Table 5. The computed correlation coefficient is very close to the actual correlation coefficient, suggesting that this method offers a promising direction for mimicking Presortedness.

5. Conclusion

This paper highlights the need to go beyond volumetric similarity in workload-aware data synthesis by incorporating critical characteristics like Duplication Distribution and Presortedness. Case studies demonstrate their impact on query performance, underscoring their importance for realistic data generation. We formalized these characteristics, proposed extraction methods, and outlined strategies to integrate them into synthesis pipelines, enhancing the fidelity of synthetic data for benchmarking. Future work will explore incorporating query execution metrics, such as buffer usage, CPU load, and disk I/O patterns, to further simulate real-world scenarios.

Acknowledgments

I would like to thank Jayant Haritsa, Carsten Binnig, Tarun Patel and Shadab Ahmed for their support and feedback.

References

[1] T. Rabl, M. Danisch, M. Frank, S. Schindler, H.-A. Jacobsen, Just can't get enough: Synthesizing big data, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, 2015, p. 1457–1462. doi:10.1145/2723372.2735378.

[2] E. Shen, L. Antova, Reversing statistics for scalable test databases generation, in: Proceedings of the Sixth International Workshop on Testing Database Systems, DBTest '13, 2013, pp. 1–6. doi:10.1145/2479440.2479445.

[3] A. Sanghi, J. R. Haritsa, Synthetic data generation for enterprise dbms, in: Proceedings of the 2023 IEEE 39th International Conference on Data Engineering, ICDE '23, 2023, pp. 3585–3588. doi:10.1109/ICDE55515.2023.00274.

[4] C. Binnig, D. Kossmann, E. Lo, M. T. Özsu, Qagen: generating query-aware test databases, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, 2007, p. 341–352. doi:10.1145/1247480.1247520.

[5] E. Lo, C. Binnig, D. Kossmann, M. Tamer Özsu, W.-K. Hon, A framework for testing dbms features, The VLDB Journal 19 (2010) 203–230. doi:10.1007/s00778-009-0157-y.

[6] E. Lo, N. Cheng, W.-K. Hon, Generating databases for query workloads, Proc. VLDB Endow. 3 (2010) 848–859. doi:10.14778/1920841.1920950.

[7] E. Lo, N. Cheng, W. W. Lin, W.-K. Hon, B. Choi, Mybenchmark: generating databases for query workloads, The VLDB Journal 23 (2014) 895–913. doi:10.1007/s00778-014-0354-1.

[8] A. Arasu, R. Kaushik, J. Li, Data generation using declarative constraints, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, 2011, p. 685–696. doi:10.1145/1989323.1989395.

[9] A. Arasu, R. Kaushik, J. Li, Datasynth: generating synthetic data using declarative constraints, Proc. VLDB Endow. 4 (2011) 1418–1421. doi:10.14778/3402755.3402785.

[10] A. Sanghi, R. Sood, J. R. Haritsa, S. Tirthapura, Scalable and dynamic regeneration of big data volumes, in: Proceedings of the 21st International Conference on Extending Database Technology, 2018, EDBT '18, 2018, pp. 301–312. doi:10.5441/002/edbt.2018.27.

[11] A. Sanghi, R. Sood, D. Singh, J. R. Haritsa, S. Tirthapura, Hydra: a dynamic big data regenerator, Proc. VLDB Endow. 11 (2018) 1974–1977. doi:10.14778/3229863.3236238.

[12] A. Gilad, S. Patwa, A. Machanavajjhala, Synthesizing linked data under cardinality and integrity constraints, in: Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data, SIGMOD '21, 2021, p. 619–631. doi:10.1145/3448016.3457242.

[13] Y. Li, R. Zhang, X. Yang, Z. Zhang, A. Zhou, Touchstone: generating enormous query-aware test databases, in: Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC '18, 2018, p. 575–586.

[14] Q. Wang, Y. Li, R. Zhang, K. Shu, Z. Zhang, A. Zhou, A scalable query-aware enormous database generator for database evaluation, IEEE Transactions on Knowledge and Data Engineering 35 (2023) 4395–4410. doi:10.1109/TKDE.2022.3153651.

[15] J. Yang, P. Wu, G. Cong, T. Zhang, X. He, Sam: Database generation from query workloads with supervised autoregressive models, in: Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data, SIGMOD '22, 2022, p. 1542–1555. doi:10.1145/3514221.3526168.

[16] Transaction Processing Performance Council, TPC Benchmark^{TM} DS Standard Specification, 2021. URL: http://www.tpc.org/tpcds/, version 3.2.0.

[17] A. Sanghi, R. Santhanam, J. R. Haritsa, Towards generating hifi databases, in: Proceedings of the 26th International Conference on Database Systems for Advanced Applications, 2021, DASFAA '21, 2021, p. 105–112. doi:10.1007/978-3-030-73194-6_8.

[18] G. Moerkotte, T. Neumann, G. Steidl, Preventing bad plans by bounding the impact of cardinality estimation errors, Proc. VLDB Endow. 2 (2009) 982–993. doi:10.14778/1687627.1687738.

[19] H. Wang, M. Song, Ckmeans. 1d. dp: optimal k-means clustering in one dimension by dynamic programming, The R journal 3 (2011) 29. doi:10.32614/RJ-2011-015.

[20] G. S. Manku, R. Motwani, Approximate frequency counts over data streams, Proc. VLDB Endow. 5 (2012) 1699. doi:10.14778/2367502.2367508.

[21] A. Sanghi, S. Ahmed, J. R. Haritsa, Projection-compliant database generation, Proc. VLDB Endow. 15 (2022) 998–1010. doi:10.14778/3510397.3510398.

[22] A. Sanghi, S. Ahmed, P. Rawale, J. R. Haritsa, Data Generation using Join Constraints, Technical Report, Indian Institute of Science, 2022. URL: https://dsl.cds.iisc.ac.in/publications/report/TR/TR-2022-01.pdf.

[23] L. De Moura, N. Bjørner, Z3: an efficient smt solver, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, 2008, p. 337–340. doi:10.1007/978-3-540-78800-3_24.

[24] Spearman's rank correlation coefficient, In Wikipedia, URL: https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient, 2024. Accessed: 16-02-2025.