High Fidelity Database Generators

Anupam Sanghi   Rajkumar Santhanam   Jayant R. Haritsa

Technical Report
TR-2021-01
(January 2021)

Database Systems Lab
Dept. of Computational and Data Sciences
Indian Institute of Science
Bangalore 560012, India

`https://dsl.cds.iisc.ac.in`

Abstract

Generating synthetic databases that capture essential data characteristics of client databases is a common requirement for database vendors. Over the last decade, several database generators have been proposed that aim for statistical fidelity on metrics such as volumetric similarity. While these techniques have broken new ground, their applicability is restricted to the predefined query workloads, i.e. they have a poor accuracy on unseen queries. This is because of (a) treating all candidate databases equally, (b) inserting an artificial skew in the database, (c) the generated data does not comply with the metadata statistics. In this report, we present HF-Hydra, a new database generator that explicitly aims to address these issues while retaining the ability to scale to big data volumes. Specifically, improved fidelity is achieved through careful choices among the set of candidate synthetic databases and incorporation of metadata constraints. With a detailed experimental study, we validate the quality and efficiency of the proposed approach.

1 Introduction

Database vendors often need to generate synthetic databases for a variety of use-cases, including: (a) testing engine components, (b) data masking, (c) testing of database applications with embedded SQL, (d) performance benchmarking, (e) analyzing *what-if* scenarios, and (f) proactively assessing the performance impacts of planned engine upgrades. Accordingly, several synthetic data generation techniques ([7], [4], [13], [3], [9]) have been proposed in the literature over the past few decades, the most recent being Hydra ([11], [12]), which has been leveraged in the software and telecom industries.

Hydra is based on a workload-aware declarative approach to database regeneration, and aims to create synthetic databases that achieve *volumetric similarity* on pre-defined query workloads. Specifically, it takes as input from the client, the metadata and query execution plans corresponding to a query workload. With this input, it generates a synthetic database at the vendor site such that a similar volume of data is processed at each stage of the query execution pipeline for the workload queries. A special feature of Hydra that makes it amenable to Big Data environments is that the data generation can be done “on-demand”, obviating the need for materialization. This functionality is achieved through the construction of a concise *database summary*.

While Hydra has the above-mentioned desirable characteristics, its applicability is restricted to handling volumetric similarity on *seen* queries. Generalization to new queries can be necessary requirement at the vendor site as part of the ongoing evaluation exercise. However, Hydra’s poor robustness is an artifact of its following design choices:

No Preference among Feasible Solutions: There can be several feasible solutions to the SMT problem. However, Hydra does not prefer any particular solution over the others. Moreover, due to the usage of Simplex algorithm internally, the SMT solver returns a sparse solution, i.e., it assigns non-zero cardinality to very few regions. This leads to very different *inter-region distribution* of tuples in the original and synthetic databases.

Artificial Skewed Data: Within a region that gets a non-zero cardinality assignment, Hydra generates a single unique tuple. As a result, a highly skewed data distribution is generated, which leads to an inconsistent *intra-region distribution* of tuples. Furthermore, the artificial skew can cause hindrance in efficient testing of queries, and gives an *unrealistic look* to the data.

Non-compliance with the Metadata: The metadata statistics present at the client site are transferred to the vendor and used to ensure matching plans at both sites. However, these statistics are not used in the data generation process, leading to data that is out of sync with the client meta-data.

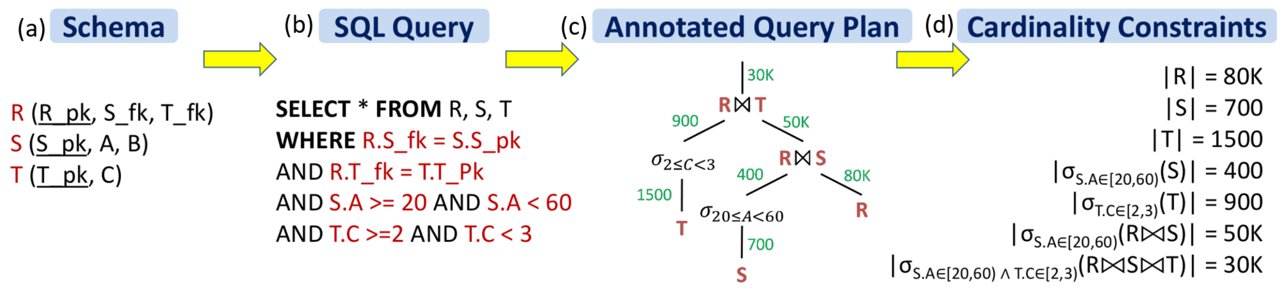


Figure 1: Example Annotated Query Plan and Cardinality Constraints

In this report, we present **HF-Hydra**, a new data generation algorithm, which attempts to address the above robustness-related limitations of Hydra while retaining its desirable features of similarity, scalability and efficiency. Specifically, the SMT problem is replaced with a linear program (LP) that works towards finding a solution that is close to the metadata stats as well. Further, instead of generating just a few unique tuples, HF-Hydra generates tuples that achieve a more realistic spread over the domain, while supporting dynamic data generation.

The efficacy of HF-Hydra is evaluated by comparing its volumetric similarity over unseen queries with Hydra. Our results indicate a substantive improvement – for instance, the volumetric similarity on filter constraints of unseen queries improves by more than **30 percent**, as measured by the UMBRAE model-comparison metric [6]. Further, we also show that along with better generalization, HF-Hydra also ensures metadata compliance and the generated data has no artificial skew. These experiments are conducted over a vanilla computing platform using the TPC-DS benchmark hosted on the PostgreSQL engine.

Organisation. The remainder of this report is organized as follows: The problem framework is outlined in Section 2, along with a summary description of Hydra. Then, we present an overview of HF-Hydra in Section 3, followed by its primary constituents – LP Formulation and Data Generation, in Sections 4 and 5, respectively. A detailed experimental evaluation of HF-Hydra is highlighted in Section 6. The related literature is reviewed in Section 7, and our conclusions are summarized in Section 8.

2 Background

In this section, we present the volumetric similarity concept and a summary description of the Hydra approach.

2.1 Volumetric Similarity

Consider the toy database example shown in Figure 1. It shows a database with three relations (R, S, T) (Figure 1(a)) and a sample SQL query (Figure 1(b)) on this database. In the query execution plan (Figure 1(c)), each edge is annotated with the associated cardinality of tuples flowing from one operator to the other. This is called an annotated query plan (AQP) [4]. From an AQP, a set of cardinality constraints (CCs)[3] (Figure 1(d)) are derived. To ensure volumetric similarity on a given query workload, when these queries are executed on the synthetic database, the result should produce similar AQPs. In other words, the synthetic database should satisfy all the CCs.

2.2 Hydra Overview

Hydra's data generation pipeline begins by deriving CCs from the input AQPs. Subsequently, for each relation, a corresponding *view* is constructed. The view comprises of the relation's own non-key attributes, augmented with the non-key attributes of the relations on which it depends through referential constraints. This transformation (predicated on the assumption that all joins are between primary key-foreign key (PK-FK) attributes) results in replacing the join-expression present in a CC with a minimal view that covers the relations participating in the join-expression. After this rewriting, we get, a set of CCs, where each CC is a filter constraint on a view along with its associated output cardinality.

Each view is decomposed into a set of *sub-views* (need not be disjoint) to reduce the complexity of the subsequent modules. After obtaining these sub-views, the following two main modules are executed:

2.2.1 SMT Problem Formulation:

For a view, the set of CCs is taken as input and an SMT Problem (Hydra uses the term LP, however technically it is a satisfiability problem because of the absence of an objective function) is constructed for this set. Each sub-view's domain space is partitioned into a set of regions using the CCs applicable on that sub-view. A variable is created for each region, representing the number of tuples chosen from the region. Each CC is encoded as a linear constraint on these variables.

For the view S from Figure 1, an example constraint is shown by the red box (A in [20, 40) and B in [15000, 50000)) in Figure 2a has an associated cardinality 150 (say). Likewise, another green constraint is also shown - say with cardinality 250. Let the total number of tuples in the S be 700. These constraints can be expressed as the following linear constraints:

$$\begin{gathered} x_1 + x_2 = 250, \\ x_2 + x_3 = 150, \\ x_1 + x_2 + x_3 + x_4 = 700, \\ x_1, x_2, x_3, x_4 \geq 0 \end{gathered}$$

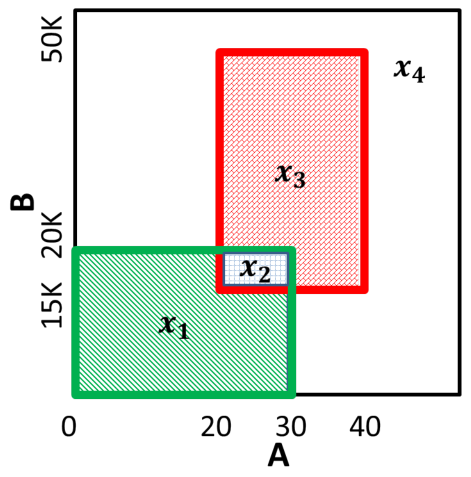
where the four variables represent the tuple cardinality assigned to the respective regions. Lastly, additional constraints are added to ensure that the marginal distributions along common attributes among sub-views are identical.

2.2.2 Database Generation:

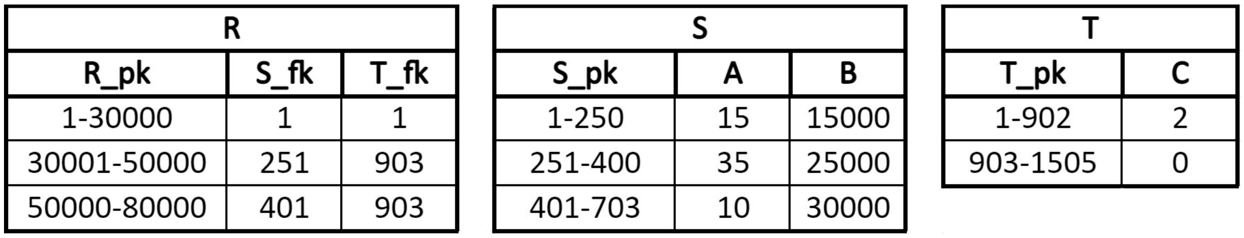
The solution to the SMT problem for a view, gives the cardinalities for various regions. These regions, being at a sub-view level, are merged to obtain the equivalents in the original view space. Since each view is solved separately, the view solutions are altered such that they obey referential integrity. In this process, some additional (spurious) tuples may be added in the views corresponding to the dimension tables. Finally, once the consistent view-solutions are obtained, a *database summary* is constructed.

An example database summary is shown in Figure 2b. Here, entries of the type a - b in the PK columns (e.g. 101-250 for S_pk in table S), mean that the relation has $b - a + 1$ tuples with values (a, a + 1,...,b) for that column, keeping the other columns unchanged. Using the summary, *Tuple Generation module* (implemented inside DBMS query executor module) dynamically generates tuples on demand during query execution.

To ensure that the client's plans are chosen to execute queries at the vendor site, Hydra uses the *CoDD* tool [1] to copy the metadata catalogs.



(a) Region Partitioning



R		
R_pk	S_fk	T_fk
1-30000	1	1
30001-50000	251	903
50000-80000	401	903

S		
S_pk	A	B
1-250	15	15000
251-400	35	25000
401-703	10	30000

T	
T_pk	C
1-902	2
903-1505	0

(b) Example Database Summary

Figure 2: Region Partitioning and Database Summary in Hydra

3 HF-Hydra

In this report, we present **HF-Hydra** (High-Fidelity Hydra), which materially extends Hydra to address the above robustness-related limitations while retaining its desirable data-scale-free and dynamic generation properties.

The end-to-end pipeline of HF-Hydra's data generation is shown in Fig. 3. The client AQPs and metadata stats are given as input to *LP Formulator*. Using the inputs, the module constructs a refined partition, i.e. it gives finer regions. Further, a linear program (LP) is constructed by adding an objective function to pick a *desirable* feasible solution. From the LP solution, which is computed using the popular Z3 solver [15], the *Summary Generator* produces a richer database summary.

In this section, we provide an overview of how our new data generator, HF-Hydra, addresses the robustness limitation of Hydra. The details of the constituent components are described in the following sections.

Inter-Region Distribution: Note that the original database also satisfies the SMT problem that Hydra constructs, but there can be several assignments of cardinalities to the regions such that all of them satisfy the constraints. For example, if we consider the constraints corresponding to the example shown in Figure 2a, both the following solutions satisfy the constraints:

$$\textbf{Solution 1:} \quad x_1 = 250, x_2 = 0, x_3 = 150, x_4 = 300$$

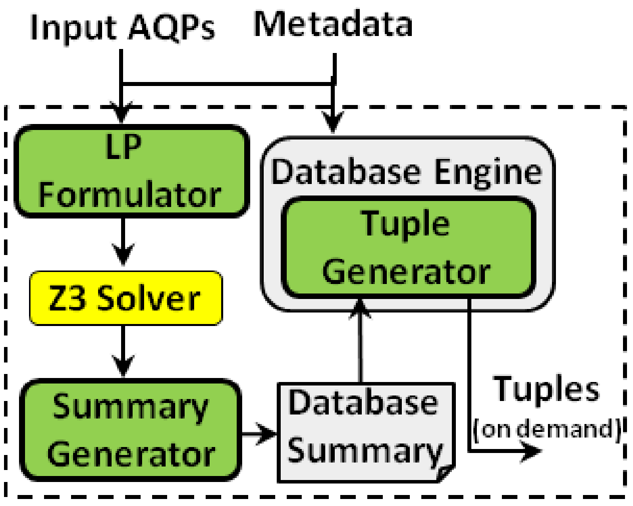


Figure 3: HF-Hydra Pipeline

$$\textbf{Solution 2:} \quad x_1 = 100, x_2 = 150, x_3 = 0, x_4 = 450$$

Hydra does not prefer any particular solution over the other. Therefore, the volume of data that is present in various regions of the original database may be very different from the cardinalities assigned by the solvers to these regions. The SMT solver assigns non-zero cardinality to very few regions. This is because of the use of the Simplex algorithm internally, which seeks a basic feasible solution, leading to a sparse solution. Further, not only is the solution sparse but also the assigned cardinalities may be very different from the original since no explicit efforts are made to bridge the gap. For example, the two solutions illustrated above are both similar from the sparsity point of view but the values themselves are very different. It is very easily possible that one of the two is picked by the solver and the other is actually the desired solution. Therefore, the *distribution* of tuples among the various regions in the original and synthetic databases can be very different. Hence, volumetric similarity for unseen queries can incur enormous errors. In HF-Hydra, we construct finer regions by additionally using metadata stats. Subsequently, an LP is formulated, by adding an optimization function that picks up a feasible solution that is close to the estimated solution derived from the stats. This also helps us ensure metadata compliance.

Intra-Region Distribution: Within a region that gets a non zero cardinality assignment, Hydra generates a single unique tuple. As a result, it leads to inconsistent *intra-region distribution*. That is, even if the inter-region distribution was matched between the original and synthetic databases, the distribution of tuples *within* a region can vary enormously. This is again not surprising since no efforts have been put to match the intra-region distribution. This not only affects the accuracy for unseen queries but single point instantiation per region also creates an artificial skew in the synthetic database. In HF-Hydra, we instead try to obey the distribution based on the information extracted from metadata stats. At the finer granularity where volumes cannot be estimated any further, we resort to uniform distribution.

4 Inter-Region Distribution: LP Formulation

Here we discuss our LP formulation module to get a better inter-region tuple cardinality distribution. We propose two strategies for LP formulation – MDC and OE, which augment the basic HF-Hydra with metadata and query optimizer estimates, respectively.

4.1 MDC: Optimization Function using Metadata Constraints

In this strategy we directly model constraints from the metadata stats. To set the stage, we first discuss some of the stats maintained by the Postgres engine.

MCVs and MCFs: Postgres maintains two lists for most attributes in a relation namely, Most Common Values(MCVs) and Most Common Frequencies(MCFs). Specifically, for an attribute A of a relation R, the frequency of an element stored at position i in the MCVs list will be stored at the matching position i in the MCFs array.

Equi-depth histogram: Postgres additionally maintains equi-depth histograms for most of the attributes in a relation. It stores the bucket boundaries in an array. All the buckets for an attribute are assumed to have the same frequency. The bucket frequency is computed by dividing the total tuple count obtained after subtracting all frequencies present in the MCFs array from the relation's cardinality, with the number of buckets.

We can express this information in the form of CCs as follows:

- For a value a stored in MCVs with frequency c_a, the corresponding CC is:

$$|\sigma_{A=a}(R)| = c_a$$

- Say for an attribute A, a bucket with a boundary $[l, h)$ and frequency B exists. Further, within this bucket, say two MCVs exist, namely p and q with frequencies c_p and c_q respectively, then the following CC can be formulated:

$$|\sigma_{A\in[l,h)}(R)| = B + c_p + c_q$$

Most database engines maintain similar kinds of metadata statistics which can be easily modeled as CCs in the above manner. To distinguish these CCs from AQP CCs, we hereafter refer to them as metadata CCs. One issue with metadata CCs is that they may not be completely accurate as they may have been computed from a sample of the database. Therefore, to ensure LP feasibility, we do not add metadata CCs explicitly in the LP and instead include them as an optimization function that tries to satisfy the metadata CCs with minimal error.

While this method provides additional constraints that help to obtain a better LP solution, it can still suffer from inconsistent inter-region distribution from the solver – in fact, we may even obtain a sparse solution. This is because there is no explicit constraint that works on the individual regions. However, in general, we expect that this optimization is likely to improve the result quality in comparison to Hydra. The complete algorithm is as follows:

1. Run region partitioning (as discussed in Section 2) using all the CCs, i.e. CCs from AQPs and metadata.

2. The CCs from AQPs are added as explicit LP constraints as before.

$$\text{minimize} \sum_{j=1}^{m} \epsilon_j \text{ subject to:}$$
$$1.\ -\epsilon_j \leq \left(\sum_{i:I_{ij}=1} x_i \right) - k_j \leq \epsilon_j, \quad \forall j \in [m]$$
$$2.\ C_1, C_2, ..., C_q$$
$$3.\ x_i \geq 0 \quad \forall i \in [n], \quad \epsilon_j \geq 0 \quad \forall j \in [m]$$

(a) MDC LP Formulation

$$\text{minimize} \sum_{i=1}^{n} \epsilon_i \text{ subject to:}$$
$$1.\ -\epsilon_i \leq x_i - \tilde{x}_i \leq \epsilon_i \quad \forall i \in [n]$$
$$2.\ C_1, C_2, ..., C_q$$
$$3.\ x_i, \epsilon_i \geq 0 \quad \forall i \in [n]$$

(b) OE LP Formulation

Figure 4: HF-Hydra LP Formulation

3. The CCs from metadata are modeled in the optimization function that minimizes the L1 norm of the distance between the output cardinality from metadata CCs and the cardinality from the sum of variables that represent the CCs. If there are n regions (variables) together obtained from m metadata CCs and q AQP CCs, then the LP is as shown in Figure 4a. Here, I_{ij} is an indicator variable, which takes value 1 if region i satisfies the filter predicate in the jth metadata CC, and takes value 0 otherwise. Further, $C_1, ..., C_q$ are the q LP constraints corresponding to the q AQP CCs.

4.2 OE: Optimization Function using Optimizer's Estimates

We now consider an alternative technique, OE, where instead of directly adding constraints from the metadata information, an *indirect* approach is used where the estimate for each region's cardinality is obtained from the engine itself. Hence, not only is the metadata information obtained, but also the optimizer's selectivity estimation logic. Once the estimates are obtained for all the regions, we find a solution that is close to these estimates while satisfying all the explicit CCs coming from the AQPs. The complete algorithm is as follows:

1. Same as Step 1 in MDC.
2. For each region obtained from (1), we construct an SQL query equivalent. Any query that can capture the region precisely is acceptable. The query generation process is described in section 4.3

As an example, the queries for the four regions from Figure 2a is as follows:

```
Select * from S where ((S.A < 20 and S.B < 20,000) or ((S.A ≥ 20 and
S.A < 30 and S.B < 15,000));
```

```
Select * from S where ((S.A ≥ 20 and S.A < 30 and S.B ≥ 15,000 and
S.B < 20,000);
```

```
Select * from S where ((S.A ≥ 20 and S.A < 30 and S.B ≥ 20,000) or (S.A
≥ 30 and S.A < 40 and S.B ≥ 15,000));
```

```
Select * from S where ((S.A < 20 and S.B ≥ 20,000) or (S.A ≥ 20
and S.B ≥ 50,000) or (S.A ≥ 40) or (S.A ≥ 30 and S.B < 15,000));
```

3. Once the region-based SQL queries are obtained, their estimated cardinalities are obtained from the optimizer. This is done by dumping the metadata on a dataless database using the CoDD metadata processing tool [1], and then obtaining the compile-time plan (for example using EXPLAIN $\langle$query$\rangle$ command in Postgres).

4. We construct an LP that tries to give a feasible solution, i.e. that satisfies all AQP constraints, while minimizing the L1 distance of each region from its estimated cardinality as obtained from the previous step. Let the estimated cardinality for the n regions be $\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_n$. Then the new LP that is formulated is shown in Figure 4b.

Our choice of minimizing L1 norm in these strategies is reasonable because from query execution point of view, the performance is linearly proportional to the row cardinality. This is especially true for our kind of workloads where the joins are restricted to PK-FK joins. Further, note that Hydra's preprocessing optimization of decomposing the view into sub-views can be easily applied with HF-Hydra as well.

Given the MDC and OE alternative strategies, the following aspects need to be considered for choosing between them:

1. MDC is comparatively simpler as it adds fewer constraints and hence is computationally more efficient. However, since generating data is typically a one-time effort, any practical summary generation time may be reasonable.

2. The solution quality depends entirely on the quality of the metadata and the optimizer estimates. Since OE works at a finer granularity, it is expected to provide higher fidelity assuming a reasonably accurate cardinality estimator. On the other hand, the accuracy of the constraints in MDC is usually more reliable as they are derived as it is from the metadata.

4.3 SQL Query Generation for Regions

An SQL query is made up of two components - **(a) From clause** - where the participating relation name(s) are mentioned and **(b) Where clause** - where **Join predicates** and **Filter predicates** are mentioned. Join predicates contain information about the attributes used in joining the relations are mentioned (in the cases where there are more than one relation participating in the query), and Filter predicates contain the filter constraint(s) applied on the participating relation(s).

A query in the client workload may or may not feature joins. If there are any joins, Hydra distinctly captures the join path (all the PK-FK attributes that were used in the join clause) for each view that is formed. So, just by extracting the join sequence information that is associated with all the attributes from all the CCs that are part of this region, we can form the Join predicates for our SQL query. From these join predicates, we can then extract all participating relation names, and construct the From clause of our SQL query.

A region in its most general form can be represented as a collection of multidimensional arrays, where each multi-d array has an array of intervals for its constituent dimensions (relation attributes). We term each dimension as a **Bucket**, and each collection of such Buckets as a **BucketStructure**. One or more such BucketStructures constitutes the region. After partitioning, all the intervals for each attribute, corresponding to a region are captured in the corresponding buckets and bucketstructures of that region. Within each bucket, the split points may or may not be continuous. The convention in Hydra is that every continuous interval is in the form of $[a, b)$ i.e. the included values are from a to $b - 1$.

The filter predicates are generated as follows :

1. Within a bucket, a continuous interval is picked. The interval is then converted to a bounded filter predicate with the end points of the interval as its bound. Incase of intervals in the buckets where there is only one known end point, and the other end point is either the start or the end of the corresponding domain, then the corresponding filter predicate generated will have only one bound with the known endpoint, and will remain unbounded on the other direction.

2. Step 1 is repeated for each continuous interval in the bucket. All the predicates generated therein are conjuncted together, and a filter predicate representing the entire bucket is generated. Each filter predicate generated for a bucket is applied against the particular attribute representing the bucket.

3. Similarly, individual filter predicates representing each bucket is created for each bucket participating in the bucketstructure. These are again conjuncted to form a filter predicate representing the entire bucketstructure. Each filter predicate for a bucketstructure is a conjunction of filter predicates on all the individual attributes participating in the bucketstructure.

4. All filter predicates representing each bucketstructure obtained from Step 3 are then disjuncted together, to form the filter predicate representing the whole region.

The final SQL query is generated as :

```
SELECT * FROM <From clause>
WHERE <Join predicates> (AND) <Filter predicates>;
```

Let us consider an example Region R to better understand the whole process. Say there are three relations X, Y and Z joined with (X.PK and Y.FK) and (Y.PK and Z.FK) with attributes columns X.A, Y.B, Z.C. Say post the LP formulation phase of Hydra, we have a region with two bucketstructures as follows :

BucketStructure 1		BucketStructure 2	
Column X.A	$[a_1, a_2, a_3, a_4, a_5)$	Column X.A	$[a_6, a_7, a_8, a_9)[a_{11}, a_{12}, a_{13})$
Column Y.B	$[-\infty, b_1)[b_3, b_4, b_5)$	Column Y.B	$[b_6, b_7)$
Column Z.C	$[c_1, c_2, c_3)$	Column Z.C	$[c_5, c_6, c_7, c_8)[c_{10}, \infty)$

Here, **From Clause = X,Y,Z** and **Join Predicates = (X.PK = Y.FK) AND (Y.PK = Z.FK)**

The filter predicates representing each bucket are:

BucketStructure 1	
Column X.A	(X.A $\geq a_1$ AND X.A $< a_5$)
Column Y.B	((Y.B $< b_1$) AND (Y.B $\geq b_3$ AND Y.B $< b_5$))
Column Z.C	(Z.C $\geq c_1$ AND Z.C $< c_5$)

BucketStructure 2	
Column X.A	((X.A $\geq a_6$ AND X.A $< a_9$) AND (X.A $\geq a_{11}$ AND X.A $< a_{13}$))
Column Y.B	(Y.B $\geq b_6$ AND Y.B $< b_7$)
Column Z.C	((Z.C $\geq c_5$ AND Z.C $< c_8$) AND (Z.C $\geq c_{10}$))

The filter predicates representing each bucketstructure are:

BucketStructure 1
((X.A $\geq a_1$ AND X.A $< a_5$) **AND** ((Y.B $< b_1$) AND (Y.B $\geq b_3$ AND Y.B $< b_5$)) **AND** (Z.C $\geq c_1$ AND Z.C $< c_5$))

BucketStructure 2
((X.A $\geq a_6$ AND X.A $< a_9$) AND (X.A $\geq a_{11}$ AND X.A $< a_{13}$)) **AND** (Y.B $\geq b_6$ AND Y.B $< b_7$) **AND** ((Z.C $\geq c_5$ AND Z.C $< c_8$) AND (Z.C $\geq c_{10}$))

The filter predicate representing the whole region is :

Region R
((X.A $\geq a_1$ AND X.A $< a_5$) AND ((Y.B $< b_1$) AND (Y.B $\geq b_3$ AND Y.B $< b_5$)) AND (Z.C $\geq c_1$ AND Z.C $< c_5$)) **OR** ((X.A $\geq a_6$ AND X.A $< a_9$) AND (X.A $\geq a_{11}$ AND X.A $< a_{13}$)) AND (Y.B $\geq b_6$ AND Y.B $< b_7$) AND ((Z.C $\geq c_5$ AND Z.C $< c_8$) AND (Z.C $\geq c_{10}$))

The final SQL query equivalent generated for region R is :

```
SELECT * FROM X,Y,Z
WHERE (X.PK = Y.FK) AND (Y.PK = Z.FK) AND
((X.A ≥ a1 AND X.A < a5) AND ((Y.B < b1) AND (Y.B ≥ b3 AND Y.B < b5))
AND (Z.C ≥ c1 AND Z.C < c5)) OR
((X.A ≥ a6 AND X.A < a9) AND (X.A ≥ a11 AND X.A < a13 )) AND
(Y.B ≥ b6 AND Y.B < b7) AND ((Z.C ≥ c5 AND Z.C < c8) AND (Z.C ≥ c10));
```

A point to note here is, the SQL query we get for a region by this process is not the only way to represent the region. There could be many other queries which can be equivalently generated to represent the region, and any of these is acceptable for our use case to get the optimizer's estimate in OE.

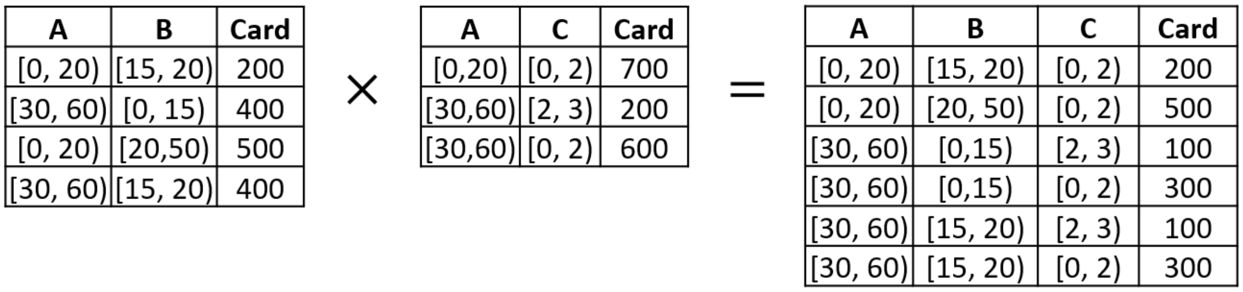
5 Intra-Region Distribution: Data Generation

We saw in the previous section how HF-Hydra handles the representation of inter-region cardinality distributions. In this section, we go on to discuss the strategy to represent the *intra-region* distributions.

The LP solution returns the cardinalities for various regions within each sub-view. As discussed earlier, the regions across sub-views need to be merged to obtain the solution at the view level. After merging, the views are made consistent to ensure referential integrity (RI) and finally a database summary is constructed from these views. This summary is used for either on-demand tuple generation or, alternatively, for generating the entire materialized database. We briefly discuss these stages now.

5.1 Merging Sub-Views

Say two sub-views R and S are to be merged and they have a common set of attributes $\mathbb{A}$. If $\mathbb{A}$ is empty then we can directly take the *cross* of the regions in R and S to obtain the regions in the view space. But when $\mathbb{A}$ is non-empty, the merging of regions happen among *compatible pair*. A region from R is compatible with a region form S if the two have an identical projection along $\mathbb{A}$. Several regions in R and S can have identical projection along $\mathbb{A}$. Therefore, we define compatibility at a set level. That is a compatible pair is a set of regions from R and S such that all these regions have identical projections along $\mathbb{A}$. For such a compatible pair, let $r_1, r_2, ..., r_m$ be the regions from R and $s_1, s_2, ..., s_n$ be the regions from S. In HF-Hydra we merge each region r_i with each region s_j from a compatible pair and assign it a cardinality $\frac{|r_i||s_j|}{\sum_{i\in[m]}|r_i|}$. Note that $\sum_{i\in[m]}|r_i| = \sum_{j\in[n]}|s_j|$, which is ensured by consistency constraints in the LP. Hence in each pair, $m \times n$ regions are constructed. Merging of two regions can be thought of as taking a join of the two regions along $\mathbb{A}$. A sample sub-view merging is shown in Figure 5 where two sub-views (A, B) and (A, C) have four and three regions respectively. Further, there are two compatible pairs that finally lead to six regions after merging.



A	B	Card
[0, 20)	[15, 20)	200
[30, 60)	[0, 15)	400
[0, 20)	[20,50)	500
[30, 60)	[15, 20)	400

$\times$

A	C	Card
[0,20)	[0, 2)	700
[30,60)	[2, 3)	200
[30,60)	[0, 2)	600

$=$

A	B	C	Card
[0, 20)	[15, 20)	[0, 2)	200
[0, 20)	[20, 50)	[0, 2)	500
[30, 60)	[0,15)	[2, 3)	100
[30, 60)	[0,15)	[0, 2)	300
[30, 60)	[15, 20)	[2, 3)	100
[30, 60)	[15, 20)	[0, 2)	300

Figure 5: Sub-view Merging

5.2 Ensuring Referential Integrity

Since each view is solved separately, it can lead to RI errors. Specifically, when F, the fact table view (having FK column), has a tuple where the value combination for the attributes that it borrows from D, the dimension table view (having PK column), does not have a matching tuple in D, then it causes an RI violation. Our algorithm for ensuring referential integrity is as follows:

1. For each region f of F, project the region on the attributes that are borrowed from D. Let the projected region be f_p.

2. Iterate on the regions in D that have non zero cardinality and find all the regions that have an intersection with f_p.

3. For each region d of D obtained from (2), split d in two disjoint sub-regions d_1 and d_2 such that d_1 is the portion of d that intersects with f_p and d_2 is the remaining portion. Cardinality of d is split between d_1 and d_2 using the ratio of their domain volumes. A corner case to this allocation is when the cardinality of d is equal to 1 – in such a case, we replace d with d_1.

4. If no region is obtained from (2), then we add f_p in D and assign it a cardinality 1. This handles RI violation but leads to an additive error of 1 in the relation cardinality for dimension table. Collectively, these errors can be considered negligible. Also, they are independent of the scale of the database we are dealing with, and therefore, as the database size grows, the relative error keeps shrinking.

As we saw that the algorithm takes projections of regions along borrowed set of dimensions. Since, the regions neither have to be hyper rectangles nor have to be continuous, they may not be *symmetric* along the borrowed attribute(s), further adding to the complexity of ensuring consistency. If the set of the attributes in a view V is A and the set of attributes that it borrows are B, then a region r in V is symmetric along B iff:

$$r = \sigma(V) = \sigma(\pi_B(V)) \times \sigma(\pi_{A \setminus B}(V))$$

In order to ensure regions are symmetric, before starting the referential integrity step, regions are split into sub-regions to make them symmetric along the borrowed set of attributes.

5.3 Generating Relation Summary

Once we get the consistent summary for each view, where for each view we have the set of (symmetric) regions and their corresponding cardinalities, we need to replace the borrowed attributes in a view with appropriate FK attributes. Here the challenge is to remain in the summary world and still achieve a good span among all the FK values within a region. To handle this, instead of picking a single value in the FK column attribute for a region, we indicate a range of FK values.

Before discussing how the FK column value ranges are computed, let us discuss how the PK columns are assigned values. Each region in any view has an associated region cardinality. PK column values are assumed to be auto-number so, given two regions with cardinality a and b, the PK column value ranges for the two regions are 1 to a and $a + 1$ to $a + b$, respectively. Now, to compute the FK column values, for each region in the fact table view F, the corresponding matching regions from the dimension table view D are fetched. Once these regions are identified, their PK column ranges are fetched and the union of these ranges is assigned to the FK column for the given F's region.

Recall that a region in its most general form can be represented as a collection of multidimensional arrays, where each multi-d array has an array of intervals for its constituent dimensions (relation attributes). For example, Figure 6 shows the structure of a region that is symmetric along dimension A. It can be expressed in words as the domain points where

$$((A \in [a1, a2) \text{ or } [a3, a4)) \text{ and } (B \in [b1, b2) \text{ or } [b3, b4)) \text{ and } C \in [c1, c2)) \text{ or}$$
$$((A \in [a1, a2) \text{ or } [a3, a4)) \text{ and } B \in [b5, b6) \text{ and } C \in [c3, c4))$$

Now, we split the region into sub-regions, where each sub-region is a single multi-d array. Further, we divide the parent region's cardinality among the sub-regions in the ratio of their volumes. In this way summary for each relation collectively gives the database summary. A sample database summary produced by HF-Hydra is shown in Figure 7 ((analogous to Figure 2b). As a sample, the second row in summary for table T means that the last 603 rows in the table have PK column value 903 to 1505, while the value for column C is the range $[0, 2)$ or $[3, 10)$.

5.4 Tuple Generation

The summary generation module gives us the database summary featuring various (sub) regions and their associated cardinalities. We want to spread the region's cardinality among several points. Now depending on our requirement of either dynamic generation or materialized database output, the strategy would slightly differ as follows:

Materialized Database If a materialized database output is desired, randomness can be introduced. For each (non-PK) attribute, the values are generated by first picking up an interval using a probability distribution where each interval's selection probability is proportional to the length of the interval. Once the interval is picked, a random value in that interval is generated.

Dynamic Generation With dynamic generation, we cannot generate values randomly because the resultant tables will not be consistent across multiple query executions. Therefore, we generate values

A	[a1, a2)	[a3, a4)
B	[b1, b2)	[b3, b4)
C	[c1, c2)	

A	[a1, a2)	[a3, a4)
B	[b5, b6)	
C	[c3, c4)	

⋮

Figure 6: Region Structure

R		
R_pk	S_fk	T_fk
1-10000	1-200	1-902
10001-30000	201-250	1-902
30001-45000	251-300	903-1505
45001-50000	301-400	903-1505
50001-60000	401-450	903-1505
60001-65000	451-550	903-1505
65001-75000	551-600	903-1505
75001-80000	601-703	903-1505

S		
S_pk	A	B
1-200	[0,20)	[0-20000)
201-250	[20,30)	[0-15000)
251-300	[20,30)	[20000-50000)
301-400	[30-40)	[15000-50000)
401-450	[0-30)	[50000-80000)
451-550	[30-40)	[0-15000) U [50000-80000)
551-600	[40-60)	[0-20000) U [50000-80000)
601-703	[0-20) U [40,60)	[20000-50000)

T	
T_pk	C
1-902	[2,3)
903-1505	[0,2) U [3,10)

Figure 7: HF-Hydra Example Database Summary

in a deterministic way. Based on the lengths of the intervals that are contained for an attribute in the region, the ratio of tuples to be generated from each interval is computed. Now, if n values have to be generated within an interval, the interval is split into n equal sub-intervals and the center point within each interval is picked. If the range does not allow splitting into n sub-intervals, then it is split into the maximum possible sub-intervals, followed by a round-robin instantiation.

5.5 Comparison with Hydra

Hydra's strategy for merging two sub-views is different. For a compatible pair where we construct $m \times n$ regions (as described above), Hydra may generate as few as $max(m, n)$ regions. Therefore, it leads to several "holes" (zero-cardinality regions) in the view, further leading to poor generalisations to unseen queries.

For regions that are constructed after merging the sub-views, a single tuple is instantiated. This is again a bad choice as it creates holes.

RI violations lead to addition of spurious tuples as we discussed. In HF-Hydra these violations are sourced primarily from *undesirable* solution, where the solver assigns a non-zero cardinality to a region in fact table for which the corresponding region(s) in dimension table has a zero cardinality. In such a case, then no matter how we distribute tuples within the regions, it will always lead to a violation. However, Hydra does not even do any careful choice of the single tuple that it instantiates within a region. Therefore, it has additional sources of RI violations.

Finally, the single point instantiation per region in Hydra generates an artifical skew and gives an unrealistic appeal to the data.

6 Experimental Evaluation

In this section, we empirically evaluate the performance of HF-Hydra as compared to Hydra. We implemented HF-Hydra on top of the Hydra codebase, which was sourced from [2]. In these experiments, the popular Z3 solver was used to compute solutions for the linear programs. The performance is evaluated using a 1 GB version of the TPC-DS decision-support benchmark. The database is hosted on a PostgreSQL v9.6 engine, with the hardware platform being a vanilla standard HP Z440 workstation. Our setup is similar to the one used in [11]).

We constructed a workload of 110 representative queries based on the TPC-DS benchmark query suite. This workload was split into *training* and *testing* sets of 90 and 20 queries, respectively. The corresponding AQPs for the training query set resulted in 225 cardinality constraints, while there were 51 such CCs for the testing query set. The distribution of the cardinalities in these AQPs covered a wide range, from a few tuples to several millions. The distribution of tuples for the CCs from test queries are shown in Table 1, with a separation into two groups – CCs that are pure selection filters on

Base Filters		Joins	
Row Count	# CCs	Row Count	# CCs
1-5	3	0-10K	8
30-300	13	15-200K	13
1000-80K	8	250K-2.5M	6
Total	24		27

Table 1: Row Cardinality Distribution in Test CCs

base relations, and CCs that involve such filters along with 1 to 3 joins. 2622 CCs were derived from metadata statistics, such as MCVs, MCFs and histogram bounds.

6.1 Volumetric Similarity on Unseen Queries

For evaluating the volumetric accuracy on the constraints derived from the unseen queries, we use the **UMBRAE** (Unscaled Mean Bounded Relative Absolute Error) model-comparison metric [6], with Hydra serving as the reference model. Apart from its core statistical soundness, UMBRAE's basis in the absolute error is compatible with the L1 norm used in the HF-Hydra optimization functions. UMBRAE values range over the positive numbers, and a value U has the following physical interpretation:

$U = 1$ denotes no improvement wrt baseline model
$U < 1$ denotes $(1 - U) * 100\%$ better performance wrt baseline model
$U > 1$ denotes $(U - 1) * 100\%$ worse performance wrt baseline model

The value U is computed using the formula:

$$U = \frac{MBRAE}{1 - MBRAE}, \text{ where } MBRAE = \frac{1}{n}\sum_{t=1}^{n} \frac{|e_t^j|}{|e_t^j| + |e_t^h|}$$

where $|e_t^j|$ and $|e_t^h|$ denote the absolute error for HF-Hydra and Hydra respectively with respect to constraint t; n denotes the total number of constraints.

The UMBRAE values obtained by the HF-Hydra variants over the 20 test queries in our workload are shown in Figure 8, with Hydra serving as the reference baseline ($U = 1$). For clearer understanding, the results for base filters, join nodes, and metadata statistics are shown separately. We see that HF-Hydra delivers more than **30%** better performance on filters, while also achieving an improvement of

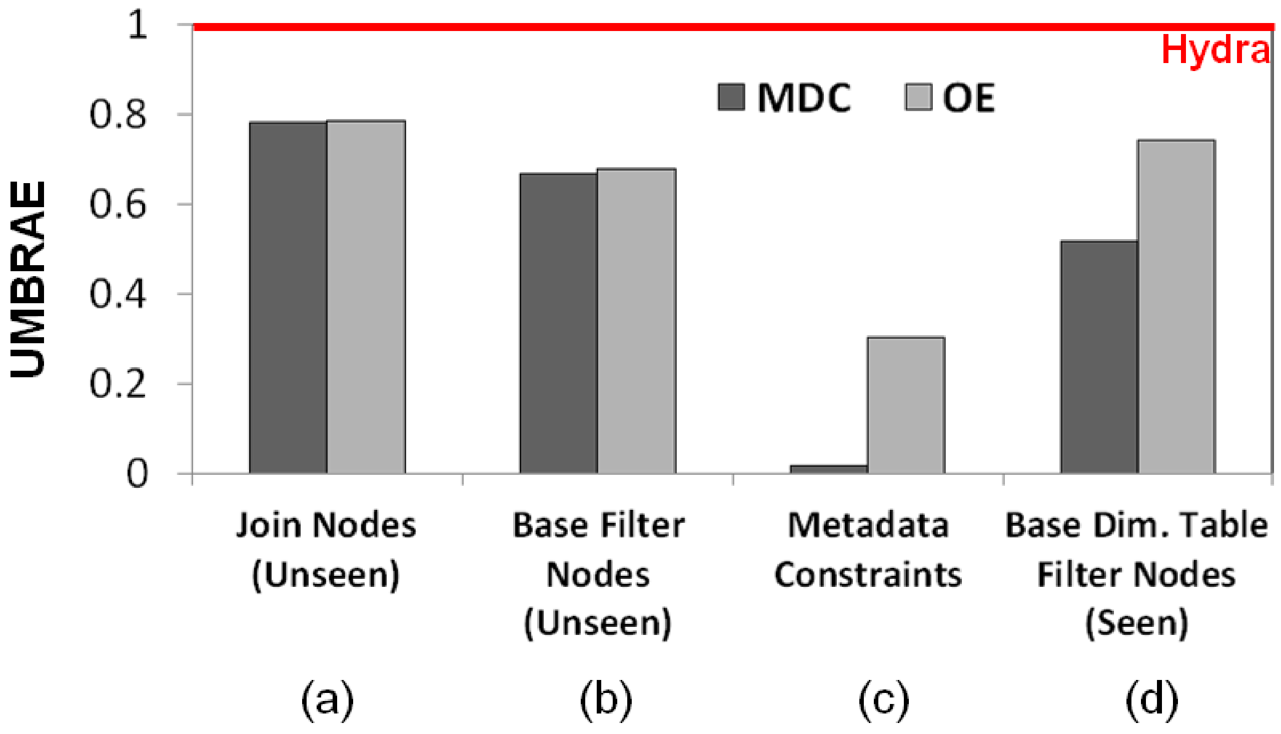


Figure 8: Volumetric Similarity on Unseen Queries and Constraints

over **20%** with regard to joins. The reason for the greater improvement on filters is that metadata is typically maintained on a per-column basis, making it harder to capture joins that combine information across columns. On drilling down we found that as the number of joins increase, the improvement keeps reducing, as should be expected due to the progressively reduced quality of the input statistics. Also, we found that among the two proposed techniques, OE did better than MDC for constraint with multiple join predicates. Specifically, we found that OE did **33%** better than MDC for 2 join cases and **13%** better for 3 join cases. This is again expected since MDC is not sensitive to join constraints beyond first level joins while OE, due to its operations on the regions directly, uses the optimizer's estimates for all the join constraints.

6.2 Metadata Compliance

We now turn our attention to evaluating the compatibility of the generated data wrt the metadata constraints. This is quantitatively profiled in Figure 8(c), and again we observe a very substantial improvement over Hydra– **98%** and **70%** for MDC and OE, respectively. Further, MDC outperforming OE is as expected because the former *explicitly* minimizes the errors in satisfying metadata CCs in its optimization function.

6.3 Database Summary Overheads

A key difference between HF-Hydra and Hydra is with regard to the structure of the database summary. Firstly, there are many more regions in the HF-Hydra summary. Secondly, instead of picking a single point per region, the HF-Hydra summary retains the entire region and generates a wider spread of tuples intra-region. Due to these changes, a legitimate concern could be the impact on the size of the database summary and the time taken to generate the database from the summary. To quantitatively evaluate this concern, the space and time overheads are enumerated in Table 2. We see here that there is certainly a large increase in summary size, going from kilobytes to megabytes – however, in *absolute* terms, the summary size is still small enough to be easily viable on contemporary computing platforms. When we consider the time aspect, again there is an expected increase in the generation time from a few seconds to several tens of seconds, but here too the absolute values are small enough (sub-minute) to make HF-Hydra usable in practice.

	Hydra	MDC	OE
Database Summary Size	40 KB	6 MB	985 MB
Tuples Materialization Time	6 seconds	37 seconds	51 seconds

Table 2: Space and Time Analysis

6.4 Data Scale Independence

The summary sizes and the time taken to generate the summary are independent of the client database size. We evaluate this by taking two instances of TPC-DS benchmark as the client database, namely 1 GB and 10 GB, and generating summary from both strategies MDC and OE. The results are enumerated in Table 3. While going from $1\times$ to $10\times$ with the database size, the summary sizes and the generation time do not vary much. Also note that although the summary generation time looks large for OE, most of the time is incurred in getting a feasible solution from the LP Solver.

Strategy	MDC		OE	
Data Scale (TPC-DS)	1 GB	10 GB	1 GB	10 GB
Database Summary Size	6 MB	6.3 MB	985 MB	1.3 GB
Summary Generation Time	12m 55s	13m 8s	22h 57m	16h 48m
Summary Generation Time (excluding LP Solver Time)	54.3s	54.3s	20m 54s	24m 41s

Table 3: Data Scale Experiment Analysis

6.5 Data Skew and Realism

Finally, to showcase the difference of skew, we drill down into the "look" of the data produced by HF-Hydra. For this purpose, a sample fragment of the TPC-DS Item relation produced by Hydra is shown in Figure 9a, and the corresponding fragment generated by HF-Hydra is shown in Figure 9b. It is evident from these pictures that unlike Hydra which has heavily-repeated attribute values – for instance, all the REC_START_DATE values are the same, HF-Hydra delivers more realistic databases with significant variations in the attribute values.

item_sk	color	price	rec_start_date
0	Coral	10.01	1991-02-01
1	Coral	10.01	1991-02-01
...	...	...	...
21	Coral	10.01	1991-02-01
17908	Beige	7.00	1991-02-01
17909	Beige	7.00	1991-02-01
...	...	...	...
17999	Beige	7.00	1991-02-01

(a) Hydra Database (Item)

item_sk	color	price	rec_start_date
7125	Beige	9.91	1990-05-08
3847	Coral	4.13	1990-03-26
1618	Dark	4.56	1990-04-06
8450	Floral	2.46	1990-06-17
2836	Navy	27.33	1990-03-06
3086	Pink	63.66	1990-04-14
1827	Red	1.61	1990-03-08
3651	Violet	7.43	1990-03-24

(b) HF-Hydra Database (Item)

Figure 9: Data Skew

7 Related Work

The early database generators (e.g. [5, 7]) produced synthetic databases that were largely agnostic to the client environments. Subsequently, to provide more realism with regard to statistical properties, a potent *declarative approach* to data generation was introduced. While QAGen [4] initiated this trend through the use of AQPs, the concept was generalised to CCs in DataSynth [3]. The data generation algorithm of DataSynth was extended to handling large sets of constraints and scaling to big data volumes in Hydra [11]. And, in this paper, we have attempted to take this prior stream of work materially forward through the introduction of robustness considerations in the design of HF-Hydra.

In parallel to the above workload-aware work, there are techniques such as DBSynth [10] and RS-Gen [13] that construct a model solely on the basis of the metadata, and then generate the database as per this model. However, it has been found that their accuracy turns out to be impractically poor for queries involving predicates on correlated attributes, a common situation in practice. While HF-Hydra also uses metadata statistics, it does so in *conjunction* with workload-awareness, and therefore does not fall prey to such errors.

8 Conclusions

Testing database engines efficiently is a critical issue in the industry, and the ability to accurately mimic client databases forms a key challenge in this effort. In contrast to the prior literature which focused solely on capturing database fidelity with respect to a known query workload, in this report we have looked into the problem of generating databases that are robust to unseen queries. In particular, we presented HF-Hydra, which materially extends the state-of-the-art Hydra generator by bringing the potent power of metadata statistics and optimizer estimates to bear on the generation exercise. In particular, this information was captured in the form of additional cardinality constraints and optimization functions. The resulting fidelity improvement was quantified through experimentation on benchmark databases, and the UMBRAE outcomes indicate that HF-Hydra successfully delivers high-fidelity databases. In future, we would like to incorporate the recently proposed selectivity estimation algorithms based on machine-learning techniques (e.g. NARU [14], MSCN [8]) into the HF-Hydra framework.

References

[1] CODD Metadata Processor. `dsl.cds.iisc.ac.in/projects/CODD`

[2] Hydra Database Regenerator. `dsl.cds.iisc.ac.in/projects/HYDRA`

[3] A. Arasu, R. Kaushik, and J. Li. Data Generation using Declarative Constraints. In *Proc. of ACM SIGMOD Conf.*, 2011, pgs. 685-696.

[4] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: Generating Query-Aware Test Databases. In *ACM SIGMOD Conf.*, 2007, pgs. 341-352.

[5] N. Bruno, S. Chaudhuri. Flexible Database Generators. In *31st VLDB Conf.*, 2005.

[6] C. Chen, J. Twycross, J. M. Garibaldi. A new accuracy measure based on bounded relative error for time series forecasting. *PLoS ONE*, 12(3): e0174202, 2017.

[7] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *ACM SIGMOD Conf.*, 1994.

[8] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR Conf.*, 2019.

[9] Y. Li, R. Zhang, X. Yang, Z. Zhang, and A. Zhou. Touchstone: Generating Enormous Query-Aware Test Databases. In *USENIX ATC*, 2018, pgs. 575–586.

[10] T. Rabl, M. Danisch, M. Frank, S. Schindler and H. Jacobsen. Just can't get enough - Synthesizing Big Data. In *ACM SIGMOD Conf.*, 2015.

[11] A. Sanghi, R. Sood, J. R. Haritsa, and S. Tirthapura. Scalable and Dynamic Regeneration of Big Data Volumes. In *21st EDBT Conf.*, 2018, pgs. 301-312.

[12] A. Sanghi, R. Sood, D. Singh, J. R. Haritsa, and S. Tirthapura. HYDRA: A Dynamic Big Data Regenerator *PVLDB*, 11(12):1974-1977, 2018.

[13] E. Shen and L. Antova. Reversing statistics for scalable test databases generation. In *DBTest Workshop*, 2013.

[14] Z. Yang et al. Deep Unsupervised Selectivity Estimation. In *PVLDB*, 13(3):279–292, 2019.

[15] Z3. `https://github.com/Z3Prover/z3`