open proceedings

Scalable and Dynamic Regeneration of Big Data Volumes

Anupam Sanghi, Raghav Sood, Jayant Haritsa
Indian Institute of Science
Bangalore, India
{anupam,raghav,haritsa}@dsl.cds.iisc.ac.in

Srikanta Tirthapura
Iowa State University
Ames, USA
snt@iastate.edu

ABSTRACT

A core requirement of database engine testing is the ability to create synthetic versions of the customer's data warehouse at the vendor site. A rich body of work exists on synthetic database regeneration, but suffers critical limitations with regard to: (a) maintaining statistical fidelity to the client's query processing, and/or (b) scaling to large data volumes. In this paper, we present **HYDRA**, a workload-dependent database regenerator that leverages a declarative approach to data regeneration to assure volumetric similarity, a crucial aspect of statistical fidelity, and materially improves on the prior art by adding scale, dynamism and functionality. Specifically, Hydra uses an optimized linear programming (LP) formulation based on a novel *region-partitioning* approach. This spatial strategy drastically reduces the LP complexity, enabling it to handle query workloads on which contemporary techniques fail. Second, Hydra incorporates deterministic post-LP processing algorithms that provide high efficiency and improved accuracy. Third, Hydra introduces the concept of *dynamic regeneration* by constructing a minuscule *database summary* that can on-the-fly regenerate databases of arbitrary size during query execution, while obeying volumetric specifications derived from the query workload. A detailed experimental evaluation on standard OLAP benchmarks demonstrates that Hydra can efficiently and dynamically regenerate large warehouses that accurately mimic the desired statistical characteristics.

1 INTRODUCTION

In industrial practice, a common requirement for database vendors is to adequately test their database engines with representative data and workloads that accurately mimic the data processing environments at customer deployments. This need can arise either in the analysis of problems currently being faced by clients, or in proactively assessing the performance impacts of planned engine upgrades on client applications. While, in principle, clients could transfer their original data and workloads to the vendor for the intended evaluation purposes, this is often infeasible due to privacy and liability concerns. Moreover, even if a client is willing to share the data, transferring and storing the data at the vendor's site may prove to have impractical space and time overheads, especially in the anticipated Big Data era. For instance, if a customer faces a problem on exabyte (10^{18}) sized relational tables, transferring and storing such data is likely to be infeasible even on the best of systems. Therefore, an important requirement, looking into the future, is to be able to *dynamically* regenerate representative databases, at query execution time that accurately mimic the behavior of the client's data processing environment.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

A rich body of literature exists on data regeneration, beginning with *workload-independent* techniques (e.g [12, 15]), which provide scalable and efficient solutions, but fail to retain complex statistical characteristics such as the sizes of intermediate relations created during execution of a query plan. To address this problem, a particularly potent approach of *workload-dependent* database regeneration was introduced in QAGen [11], and has served as the foundation for many of the practicable systems proposed over the last decade [6, 18]. Workload-dependent techniques aim to generate synthetic data whose behavior is *volumetrically similar* to the client database on the pre-specified query workload. That is, assuming a common choice of query execution plans at the client and vendor sites (ensured through "plan forcing" [3] or "metadata matching" [8]), the output row cardinalities of individual operators in these plans are very similar in the original and synthetic databases. This similarity helps to preserve the multi-dimensional layout and flow of the data, a pre-requisite for achieving similar performance on the client's workload. As a case in point, the DataSynth [6, 7] tool from Microsoft expresses such volumetric constraints as a Linear Program (LP) whose solution is used to construct the synthetic database.

A common limitation of contemporary techniques (reviewed in detail in Section 8), is that they run into issues of *scale* and *efficiency* at one stage or the other in the regeneration pipeline. This is partly due to their focus on *materialized* static solutions, making them impractical at large volumes. Further, the ability to scale to large query workloads and data volumes has not been clearly established, and validations have been typically restricted to relatively simple and small benchmarks such as TPC-H [2]. These limitations become especially problematic from a futuristic "Big Data" perspective, where we have to contend with enormous data volumes and complex query workloads.

To materially address this challenge, we present **HYDRA**, a data regeneration tool, which ensures that scale and efficiency are addressed through *the entire regeneration pipeline*. As a concrete example, Hydra was able to accurately regenerate the data processing environment of a 100 GB TPC-DS client database with a workload of 131 distinct representative queries, by generating a *database summary* in less than 2 minutes on a vanilla machine. This summary can be used to statically generate a materialized database, or more potently, to *dynamically* regenerate the desired database during query execution. When the former option is chosen, the static database was successfully created in less than 11 minutes. It is important to note here that the summary construction time is *independent* of the data scale – therefore, even the exabyte-sized data scenario alluded to earlier could be modeled in just a few minutes using Hydra!

The key contributions of Hydra are the following:

Extended Workload Coverage: Hydra incorporates a novel LP formulation technique, *region-partitioning*, that can encode volumetric constraints with an LP of low complexity. When compared with the *grid-partitioning* approach used in DataSynth, region-partitioning reduces the LP complexity by many orders of magnitude. For instance, an LP with

more than a *billion* variables in DataSynth is reduced to an LP with a few *thousand* variables in Hydra– in fact, in this case, the LP solver crashes on the DataSynth formulation, but runs to completion in less than a minute on the Hydra formulation. The beneficial outcome of the low LP complexity is that it facilitates the efficient handling of much richer query workloads.

Apart from enhancing the workload scale, Hydra also expands the database scope to include relational schemas that have DAG-structured dependency graphs, and the query scope to include DNF filter predicates.

Database Summary and Dynamic Regeneration: A unique feature of our data regeneration approach is that it delivers a *database summary* as the output, rather than the static data itself. This summary is of negligible size, depending only on the query workload and *not* on the database scale. It can be used for *dynamically* generating data during query execution, or for materializing static relations if so desired. This summary-based approach eliminates the enormous time and space overheads incurred by prior techniques in generating and storing data before initiating analysis.

Accuracy with Efficiency: Hydra replaces the *sampling-based* approach to data regeneration in DataSynth by a *deterministic alignment* strategy. The alignment operates directly on the database summary, and is therefore extremely efficient. Further, it does not suffer the probabilistic errors that affect the sampling approach, and therefore delivers better fidelity with regard to volumetric similarity.

Enhanced Evaluation: We evaluate Hydra on a diverse workload of 100-plus queries constructed from the complex TPC-DS benchmark, and the results show that it can efficiently regenerate databases for such workloads at various data scales. Further, our evaluation is more comprehensive than prior techniques, which have largely been evaluated on simpler and small-sized query workloads operating on modest databases. For instance, DataSynth has been evaluated on simple TPC-H database environments that resulted, with their formulation, in LPs with only a few thousand variables.

Integration with CODD: CODD [8] is a graphical tool through which database environments with desired meta-data characteristics can be efficiently simulated without persistently generating and/or storing their contents – i.e. a "dataless" approach. We have integrated Hydra with CODD, thus providing an end-to-end system that fully replicates the client data processing environment at the vendor's site, and is compliant with the CODD's "dataless" philosophy.

Organization. The remainder of this paper is organized as follows: A brief background on the key underlying concepts is outlined in Section 2. The Hydra architecture is presented in Section 3, and our new region-based LP formulation in Section 4. The database summary generator and the tuple generator are described in Sections 5 and 6, respectively. Our experimental results are analyzed in Section 7. Related work is reviewed in Section 8, and our conclusions are summarized in Section 9.

2 PRELIMINARIES

In this section, we provide background information on the key foundations – Annotated Query Plans [11] and Cardinality Constraints [6] – that lie under this data regeneration framework.

2.1 Annotated Query Plans

Consider a toy scenario (for ease of presentation) where the client has the database schema shown in Figure 1a, where pk and fk refer to primary-key and foreign-key attributes, respectively.

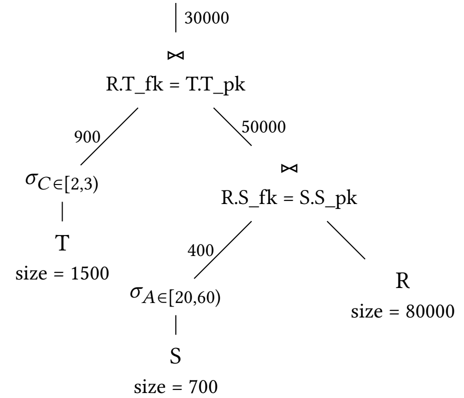
A sample client query on this schema is shown in Figure 1b, with the corresponding query execution plan in Figure 1c. Note that this execution plan has the output edge of each operator annotated with the associated row cardinality (as evaluated during the client's execution) – for instance, there are 50000 rows resulting from the join of R and (filtered) S. Such a plan is referred to as an "Annotated Query Plan" (AQP) in [11]. The goal now is to generate synthetic data at the vendor site such that when the above query is executed on this data, we obtain an identical, or very similar, AQP.

R (R_pk, S_fk, T_fk)   S (S_pk, A, B)   T (T_pk, C)

(a) Database Schema

```
select * from R, S, T
where R.S_fk = S.S_pk   and   R.T_fk = T.T_pk
and S.A >= 20 and S.A < 60   and   T.C >= 2 and T.C < 3
```

(b) Example Query



(c) Annotated Query Plan (AQP)

$|R| = 80000 \quad |S| = 700 \quad |T| = 1500$
$|\sigma_{S.A\in[20,60)}(S)| = 400 \quad |\sigma_{T.C\in[2,3)}(T)| = 900$
$|\sigma_{S.A\in[20,60)}(R \bowtie S)| = 50000$
$|\sigma_{S.A\in[20,60)\wedge T.C\in[2,3)}(R \bowtie S \bowtie T)| = 30000$

(d) Cardinality Constraints (CCs)

Figure 1: Example Database Scenario

2.2 Cardinality Constraints

A unified and declarative mechanism for representing AQP data characteristics, called *cardinality constraints* (CCs), was proposed in [6]. For instance, the CCs expressing the AQP of Figure 1c are shown in Figure 1d. The data regeneration technique takes the schematic information and the set of CCs from the client site and produces synthetic data that closely meets these CCs. To make the problem tractable, it is assumed that CCs consist of filters on only *non-key* attributes, and that all joins are between primary keys and foreign keys, typically the case in data warehouses.

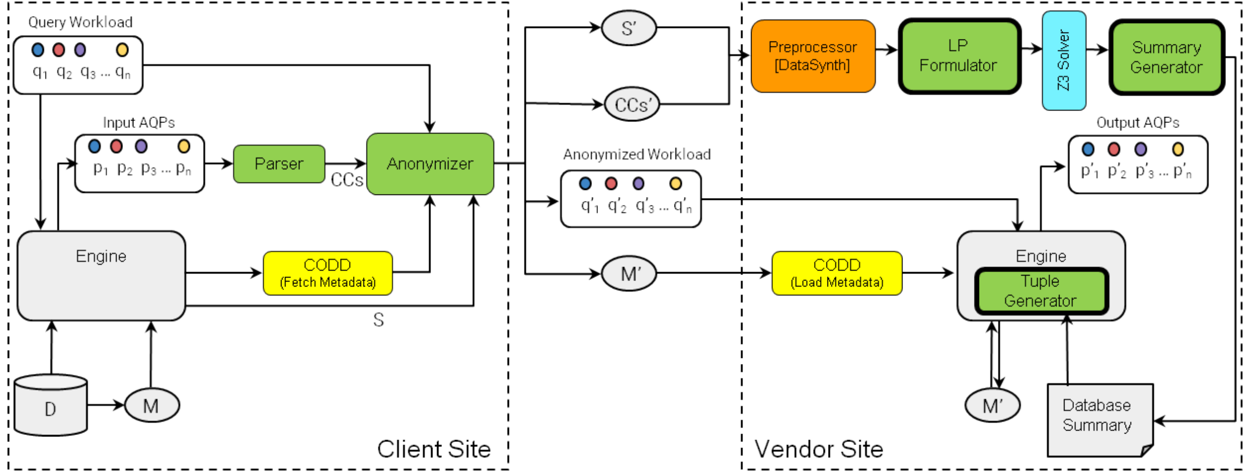


Figure 2: Hydra Architecture

3 THE HYDRA ARCHITECTURE

In this section, we present an overview of Hydra's architecture, along with a summary of its various components and their interactions with the database engine. A pictorial view of the architecture is presented in Figure 2 – in this picture, the green boxes represent the new components designed specifically for Hydra. Among these, the primary components are the LP Formulator, the Summary Generator, and the Tuple Generator, all shown with thick borders. The other modules have been sourced from the literature, including the preprocessor (orange) from DataSynth [7], the CODD metadata processor (yellow) [8], and the Z3 solver (blue) [14]. (Refer [21] for complete details.)

3.1 Client Site

The information flow from the client to the vendor is as follows: At the client site, Hydra fetches the schema information (S), and the query workload ($q_1, q_2, q_3, ..., q_n$) with its corresponding AQPs ($p_1, p_2, p_3, ..., p_n$) obtained from the database engine. The AQPs are converted to equivalent cardinality constraints (*CCs*) using a Parser. The metadata (M) from the database catalogs is captured with the help of CODD. In order to address client security concerns, all this information (schema, metadata, queries and CCs) is passed through an Anonymizer that suitably masks the information before shipping it to the vendor. Also in this process, non-numeric constants appearing in the queries and plans are mapped to numbers to facilitate LP formulation at the vendor site. Due to this mapping, the final database summary generated at the vendor site also consists of only numeric datatypes. It is possible to reverse this mapping to get back the original datatypes, but is not a relevant consideration with regard to satisfying CCs.

3.2 Vendor Site

The main modules at the vendor site are as follows:

Preprocessor [7]: In this module, sourced from DataSynth, the schema information and CCs obtained from the client are processed to create the input for the LP Formulator. Each relation is solved independently, and this process is initiated by first creating a *view* comprised of its own non-key attributes, augmented with the non-key attributes of the relations on which it depends through referential constraints (both directly or transitively). This transformation results in replacing the join-expression present in a CC with a view that covers all the attributes (non-key) featured in the relations participating in the join-expression. As a case in point, following views are generated for the example in Figure 1:

R_view (A, B, C)   S_view (A, B)   T_view (C)

Further, the last two constraints in Figure 1d can be rewritten as:

$$|\sigma_{A\in[20,60)}\text{ (R_view)}| = 50000$$
$$|\sigma_{A\in[20,60)\wedge C\in[2,3)}\text{ (R_view)}| = 30000$$

An LP is independently formulated for each view created by the above process. Since the LP complexity is adversely affected by the number of attributes in the view, the view is first decomposed into a set of *sub-views* to reduce the effective complexity. This is achieved as follows: Construct a "view-graph" by first creating a node for each attribute, and then inserting an edge between a pair of nodes if the corresponding attributes appear together in one or more CCs. Further, additional edges are added (if required) to make the view-graph to be *chordal*, a property required to ensure acylicity in the subsequent processing. Now, the sub-views are identified as the *maximal cliques* in the view-graph.

LP Formulator and Solver: For each view, the LP Formulator takes as input the corresponding set of subviews and applicable CCs, and then constructs the LP. The domain corresponding to each sub-view is partitioned into regions using a novel *region-partitioning* algorithm that takes as input the different cardinality constraints. There is one variable for each region, corresponding to the number of tuples chosen from the region. Each cardinality constraint is encoded as an LP constraint on these variables, and the solution of the LP is used in deciding which tuples to include in the sub-view. The complete details of this algorithm are enumerated in Section 4.

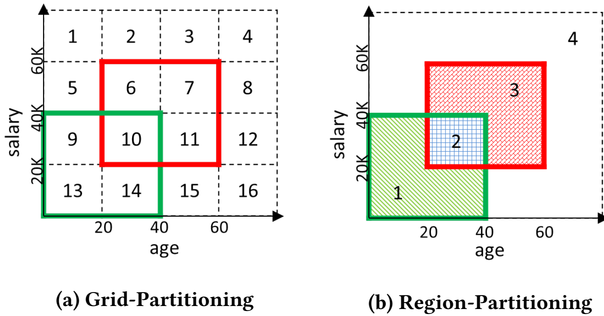
Our region-partitioning strategy is in marked contrast to the *grid-partitioning* strategy used in DataSynth. Grid-Partitioning first intervalizes the domain of each attribute based on the constants appearing in the CCs, and divides the domain into a grid aligned with the interval boundaries for each attribute. If a sub-view has n attributes, and each attribute gets divided into ℓ intervals, then the domain of the sub-view is partitioned into a grid of ℓ^n cells. For each cell in the grid, a variable is created that represents the number of data rows present in that cell. In contrast, our region-partitioning strategy divides the domain into only the number of regions required to precisely write out each

cardinality constraint, and assigns one variable to each region – this typically leads to far fewer variables than grid-partitioning.

To make the above concrete, consider a single view "Person" with the following three selection CCs:

$$|age < 40 \wedge salary < 40K \text{ (Person)}| = 1000$$
$$|20 \leq age < 60 \wedge 20K \leq salary < 60K \text{ (Person)}| = 2000$$
$$|Person| = 8000$$

Grid-partitioning divides the domain of the view as shown in Figure 3a. With a variable assigned to each grid cell, there is a total of 16 variables. In contrast, the region-partitioning strategy partitions the space into 4 regions as shown in Figure 3b, resulting in a tally of only 4 variables.



(a) Grid-Partitioning **(b) Region-Partitioning**

Figure 3: Grid-Partitioning vs Region-Partitioning

The CCs of Person, expressed in terms of LP constraints, are shown below in Figure 4a and 4b for grid-partitioning and region-partitioning, respectively.

$$x_9 + x_{10} + x_{13} + x_{14} = 1000$$
$$x_6 + x_7 + x_{10} + x_{11} = 2000$$
$$x_1 + x_2 + \ldots + x_{16} = 8000$$

(a) Grid-Partitioning

$$y_1 + y_2 = 1000$$
$$y_2 + y_3 = 2000$$
$$y_1 + y_2 + y_3 + y_4 = 8000$$

(b) Region-Partitioning

Figure 4: LP Constraints

The LPs are passed on to the solver, which provides one of the feasible solutions as the output – we have used Z3 [14], a popular SMT solver, to implement this functionality. With region-partitioning, the LP is usually much simpler due to the smaller number of variables. Further, as the cardinality constraints get more complex, the differences in complexity of the LPs produced by region-partitioning and grid-partitioning become more pronounced. This effect is quantified in Section 7.

Summary Generator: This module generates the database summary from the LP solutions obtained on the views. Since partitioning is carried out at a sub-view level, the LP solution, which is expressed in terms of sub-view variables, needs to be mapped to equivalents in the original view space. A sampling-based approach was proposed in [6] for this purpose – for example, say a view (A, B, C) is split into a pair of sub-views (A, B) and (B, C), the algorithm computes the distributions $Prob(A, B)$ and $Prob(C|B)$. Then, each tuple is generated by first sampling a point from the former distribution, and then sampling a point from the latter conditioned on this outcome.

However, we have chosen not to take this approach since the computational overheads incurred are enormous, and the sampling process introduces errors in volumetric fidelity. Instead, we have designed and implemented an alternative data-scale free, deterministic alignment algorithm (details in Section 5), which produces an intermediate database summary in the output. This component is also responsible for ensuring that the generated summary obeys referential integrity. Finally, summarized relations from corresponding view summary are obtained. An example database summary finally obtained from the AQP shown in Figure 1c, along with additional two AQPs, is shown in Figure 5. Here, entries of the type a - b in the primary key columns (e.g. 101-250 for S_pk in table S), mean that the relation has $b - a + 1$ tuples with values $(a, a+1, a+2, \ldots, b)$ for that column, keeping the other columns unchanged.

R

R_pk	S_fk	T_fk
1 – 30K	321	1
30001 – 50K	621	601
50001 – 60K	71	601
60001 – 70K	121	1
70001 – 80K	1	1

S

S_pk	A	B
1–100	0	15
101–250	20	15
251–500	20	10
501–700	0	5

T

T_pk	C
1–600	0
601–1500	2

Figure 5: Example Database Summary

Tuple Generator: The Tuple Generator resides in the database engine. It ensures that whenever a query is fired, data is not fetched from the disk but instead gets generated *on-demand*, using the database summary. The details of this component and its implementation in PostgreSQL are presented in Section 6.

We note in closing that in order to ensure the execution plan chosen at the vendor site is the same as that in the client site, metadata matching is implemented in Hydra using CODD's metadata transfer feature.

4 LP FORMULATION

An LP for a view V is constructed as follows: For each sub-view s in V, every CC that is within its scope is formulated as an LP constraint. Since sub-views may share common attributes, additional *consistency constraints* are added to the LP to ensure that the marginal distributions along the common set of attributes are identical in the solutions for the sub-views.

In this section, we first present the mathematical basis underlying our formulation of LP constraints for a set of CCs applicable on a sub-view. We then present an algorithm that partitions the domain into the minimum number of regions required to capture each CC precisely, resulting in an LP with the optimal number of variables. Finally, we discuss the formulation of additional consistency constraints to ensure consistency across multiple sub-views belonging to V.

4.1 Mathematical Basis for LP Formulation

Let n denote the number of attributes in the given sub-view s, $\mathcal{D}_i$ the domain of the ith attribute, and $\mathcal{D}$ the data universe $\mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \mathcal{D}_n$.

We are given a set of m CCs that are applicable on s. For $1 \leq j \leq m$, each constraint C_j is a pair $\langle \sigma_j, k_j \rangle$ where σ_j is a selection predicate and k_j is a non-negative integer equal to the number of rows satisfying predicate σ_j. We assume that each predicate is in disjunctive normal form (DNF).

Simple LP Formulation. Let us first consider a simple way of formulating an LP that encodes all CCs. For each tuple $t \in \mathcal{D}$, assign a variable x_t that denotes the number of copies of t in the sub-view s. Then, the LP formulation shown in Figure 6 ensures that a feasible solution satisfies all CCs, including a constraint on the total size of s.

The problem with this formulation is that the number of variables in the resulting LP is as large as the size of the universe $\mathcal{D}$. Hence, it is infeasible to work directly with this formulation.

(1) For each $t \in \mathcal{D}, x_t \geq 0$

(2) $\left\lfloor \sum_{t \in \mathcal{D}} x_t \right\rfloor = k$

(3) For each $j, 1 \leq j \leq m, \left\lfloor \sum_{t:\sigma_j(t)=true} x_t \right\rfloor = k_j$

Figure 6: Simple LP formulation

Reduced LP Formulation. We can derive an LP with far fewer variables as follows: We first note that in the simple formulation, variables corresponding to a pair of points $t_1, t_2 \in \mathcal{D}$ that behave identically with respect to a constraint C_j (i.e. $\sigma_j(t_1) = \sigma_j(t_2)$) can be combined together as $(x_{t_1} + x_{t_2})$, for the purposes of satisfying constraint C_j. If this is true that with respect to every constraint C_j for $j = 1 \ldots m$, $\sigma_j(t_1) = \sigma_j(t_2)$, then there is no need to treat t_1 and t_2 separately – instead, they can be combined into a single region, and the variables x_{t_1} and x_{t_2} can be merged into a single variable $(x_{t_1} + x_{t_2})$ in every equation, leading to fewer variables in the LP. By repeating this variable merging process recursively until it is no further possible, we arrive at a vastly reduced LP.

We hasten to add that the above LP construction process based on merging variables is only for illustrating the concept – the actual algorithm employed in our system *directly* derives the regions, as described in Section 4.2.

For constraint C and $t \in \mathcal{D}$, let $C(t)$ be an indicator variable:

$$C(t) = \begin{cases} \text{true} & \text{if } t \text{ satisfies } C \\ \text{false} & \text{otherwise} \end{cases}$$

Definition 4.1. For a pair of points $p, q \in \mathcal{D}$ and a set of constraints $\mathbb{C}$, we say $pR^{\mathbb{C}}q$ if for each $C \in \mathbb{C}$, $C(p) = C(q)$.

Observation 1. *$R^{\mathbb{C}}$ is an equivalence relation on $\mathcal{D}$.*

Proof. It can be easily seen that $R^{\mathbb{C}}$ is reflexive and symmetric. For transitivity, suppose that for $p, q, r \in \mathcal{D}$, $pR^{\mathbb{C}}q$ and $qR^{\mathbb{C}}r$. Note that for each $C \in \mathbb{C}$, it must be true that $C(p) = C(q)$ and $C(q) = C(r)$. Therefore, it must be true that $C(p) = C(r)$ for each $C \in \mathbb{C}$, showing that the relation is transitive. □

A partition of $\mathcal{D}$ is a set of subsets of $\mathcal{D}$ such that every element $x \in \mathcal{D}$ is in exactly one of these subsets. The individual sets in a partition are called *blocks*.

Definition 4.2. A set of points b is said to be valid with respect to a set of constraints $\mathbb{C}$ if for any two points $p, q \in b$, $pR^{\mathbb{C}}q$. Given a set of constraints $\mathbb{C}$, a partition $\mathbb{P}$ of $\mathcal{D}$ is said to be a *valid partition* if for each block $b \in \mathbb{P}$, b is valid with respect to $\mathbb{C}$.

In a valid partition of $\mathcal{D}$ with respect to $\mathbb{C}$, any pair of points within the same block satisfy the same set of CCs. Once we obtain a valid partition $\mathbb{P}$, the LP can be re-formulated as shown in Figure 7. Instead of a variable for each point $t \in \mathcal{D}$, there is now a single variable x_b for each block $b \in \mathbb{P}$ representing the number of tuples of the sub-view that are contained in this block. Note that the tuples in a sub-view need not be unique, therefore x_b may include duplicates in its count.

(1) For each $b \in \mathbb{P}, x_b \geq 0$

(2) $\left\lfloor \sum_{b \in \mathbb{P}} x_b \right\rfloor = k$

(3) For each $j, 1 \leq j \leq m, \left\lfloor \sum_{b:\sigma_j(b)=true} x_b \right\rfloor = k_j$

Figure 7: Reduced LP formulation

The total number of variables in the reduced LP shown in Figure 7 is equal to the number of blocks in the partition $\mathbb{P}$ and is potentially much smaller than the number of variables in the original LP, shown in Figure 6. Since we desire an LP with the smallest number of variables, we look for a valid partition of $\mathcal{D}$ with the minimum number of blocks. A valid partition with respect to $\mathbb{C}$ is an *optimal partition* if it has the smallest number of blocks from among all valid partitions of $\mathcal{D}$ with respect to $\mathbb{C}$.

Lemma 4.3. *The quotient set of $\mathcal{D}$ by $R^{\mathbb{C}}$ is the (unique) optimal partition of $\mathcal{D}$ with respect to $\mathbb{C}$.*

Proof. Let $\mathbb{P}_1$ denote the quotient set[1] of $\mathcal{D}$ by $R^{\mathbb{C}}$. By the definition of an equivalence relation, for any block $b \in \mathbb{P}_1$, all points in b are related to each other by $R^{\mathbb{C}}$, and hence $\mathbb{P}_1$ is a valid partition.

Suppose that $\mathbb{P}_1$ is not the unique optimal partition. Then, there must exist another valid partition $\mathbb{P}_2$ such that $\mathbb{P}_2 \neq \mathbb{P}_1$ and $|\mathbb{P}_2| \leq |\mathbb{P}_1|$. This implies that there exist two points $p, q \in \mathcal{D}$ such that p and q are in different blocks in $\mathbb{P}_1$, but in the same block in $\mathbb{P}_2$. Since p and q belong to different blocks in $\mathbb{P}_1$, it must be true that p and q are not related by $R^{\mathbb{C}}$. But, in $\mathbb{P}_2$ points p and q belong to the same block, which implies that $\mathbb{P}_2$ cannot be a valid partition, a contradiction. □

4.2 Deriving the Optimal Partition

We now present an algorithm to derive the optimal partition of $\mathcal{D}$ with respect to $\mathbb{C}$. Each constraint $C \in \mathbb{C}$ is in DNF, and is expressed as the union of many smaller "sub-constraints". Each sub-constraint is the conjunction of many per-attribute constraints, and each per-attribute constraint is a constraint on the values that the attribute is permitted to take. For example, the following constraint on attributes A_1 and A_2:

$$((A_1 \leq 20) \wedge (A_2 > 30)) \vee (A_1 > 50)$$

is divided into the basic sub-constraints:

$$(A_1 \leq 20) \wedge (A_2 > 30) \text{ and } (A_1 > 50)$$

Algorithm 1 (Optimal Partition) takes a set of DNF constraints as input, and returns a partition with the smallest number of regions with respect to this set. Internally, it invokes Algorithm 2 (Valid Partition) that takes a set of sub-constraints as input and returns a valid partition of the domain with respect to this set.

[1] The quotient set is the set of equivalence classes resulting from $R^{\mathbb{C}}$ on $\mathcal{D}$.

Algorithm 1: Optimal Partition($\mathcal{D}, \mathbb{C}$)

Input: Universe $\mathcal{D}$, set of DNF constraints $\mathbb{C}$
Output: An optimal partition $\mathbb{P}^*$ of $\mathcal{D}$ subject to $\mathbb{C}$

1. Generate the set of sub-constraints $\mathbb{C}'$ resulting from the constraints in $\mathbb{C}$;
2. Construct a valid partition $\mathbb{P}'$ of $\mathcal{D}$ subject to $\mathbb{C}'$ using Valid-Partition($\mathcal{D}, \mathbb{C}'$) (Algorithm 2);
3. For each block $b \in \mathbb{P}'$, compute the label $\ell(b)$, equal to the set of all constraints in $\mathbb{C}$ that b satisfies. Let L denote the set of all distinct labels from $\{\ell(b) | b \in \mathbb{P}'\}$;
4. Coarsen partition $\mathbb{P}'$ into $\mathbb{P}^*$ as follows: For each label $l \in L$, merge all blocks in $\mathbb{P}'$ whose labels equal l into a single block;
5. Return $\mathbb{P}^*$;

Lemma 4.4. *Given a set of DNF constraints $\mathbb{C}$, Algorithm 1 returns an optimal partition of $\mathcal{D}$ with respect to $\mathbb{C}$.*

Proof. As in the algorithm, let $\mathbb{C}'$ denote the set of sub-constraints resulting from constraints in $\mathbb{C}$. From Lemma 4.7, we know that $\mathbb{P}'$ is a valid partition with respect to $\mathbb{C}'$. Consider any block $b \in \mathbb{P}'$. Since b is valid with respect to $\mathbb{C}'$, and each constraint in $\mathbb{C}'$ is stricter than a corresponding constraint in $\mathbb{C}$, b is valid with respect to $\mathbb{C}$. Hence, $\mathbb{P}'$ is a valid partition with respect to $\mathbb{C}$.

Next, consider that each block b^* in $\mathbb{P}^*$ was obtained by merging blocks in $\mathbb{P}'$ that have the same label. For any pair of points p, q in b^*, it is true they satisfy the same set of constraints in $\mathbb{C}$, showing that $\mathbb{P}^*$ is a valid partition wrt $\mathbb{C}$. Also, any two blocks in $\mathbb{P}^*$ have distinct labels (if they had the same label, they would have been merged). Therefore, we conclude using arguments similar to Lemma 4.3 that $\mathbb{P}^*$ is an optimal partition of $\mathcal{D}$ with respect to $\mathbb{C}$. □

Deriving a Valid Partition for a Set of Sub-Constraints: We now present an algorithm for deriving a valid partition with a small number of blocks, for a set of sub-constraints $\mathbb{C}$.

Definition 4.5. For a sub-constraint C and dimension i, let C^i denote the restriction (projection) of C to dimension i. Further, let $C_1^i = \bigwedge_{k=1\dots i} C^k$ denote the restriction of C to dimensions $1, 2, \dots, i$. For instance, if $C = (A_1 \geq 1) \wedge (A_2 \geq 4) \wedge (A_2 \leq 5) \wedge (A_3 > 6)$, then $C^2 = (A_2 \geq 4) \wedge (A_2 \leq 5)$, and $C_1^2 = (A_1 \geq 1) \wedge (A_2 \geq 4) \wedge (A_2 \leq 5)$. For convenience, if C does not have a constraint along dimension i, then C^i is defined to be "true".

Our algorithm, described in Algorithm 2, proceeds iteratively, one dimension at a time. Before processing dimension i, it has a partition of $\mathcal{D}$ that is a valid partition subject to constraints along dimensions 1 till $(i-1)$. In processing dimension i, it refines the current partition as follows: For each block b in the current partition, it appropriately divides the block along dimension i if there is a constraint $C \in \mathbb{C}$ such that there are some points in b that satisfy constraint C^i, and some that do not.

Definition 4.6. A constraint C is said to split a block $b \subseteq \mathcal{D}$ if there exist a pair of points $p_1, p_2 \in b$ such that $C(p_1)$ = true and $C(p_2)$ = false. If C splits b, then refining b by C partitions b into two subsets $b^+(C) = \{x \in b | C(x) = \text{true}\}$ and $b^-(C) = \{x \in b | C(x) = \text{false}\}$.

Lemma 4.7. *Given a set of sub-constraints $\mathbb{C}$, Algorithm 2 returns a valid partition of $\mathcal{D}$ with respect to $\mathbb{C}$.*

Algorithm 2: Valid-Partition($\mathcal{D}, \mathbb{C}$)

Input: Universe $\mathcal{D}$, set of sub-constraints $\mathbb{C}$
Output: A valid partition $\mathbb{P}$ of $\mathcal{D}$ subject to set of sub-constraints $\mathbb{C}$

```
1  P^0 = {D} // A partition with one set, D.
2  for i from 1 to n do
3      M ← P^{i-1};
4      foreach C ∈ ℂ do
5          M' ← ∅;
6          foreach block b ∈ M do
7              if C^i splits b then
8                  Let b^+ and b^- result from refining b with C^i;
9                  Add b^+ and b^- to M';
10             else
11                 Add b to M';
12         M ← M';
13     P^i ← M;
14 Return P^n;
```

Proof. For $1 \leq i \leq n$, let $\mathbb{C}_1^i = \{C_1^i | C \in \mathbb{C}\}$. We show by induction on i that after the ith iteration of the outermost for loop in the algorithm, $\mathbb{P}^i$ contains a valid partition of $\mathcal{D}$ with respect to $\mathbb{C}_1^i$. Since $\mathbb{C}_1^n = \mathbb{C}$, it follows that after n iterations, $\mathbb{P}^n$ contains a valid partition of $\mathcal{D}$ with respect to $\mathbb{C}$. We consider $i = 0$ as the base case, and the set $\mathbb{C}_1^0$ as a set of "always true" constraints. Hence, $\mathbb{P}^0$, which consists of only one element, $\mathcal{D}$, is a valid partition with respect to $\mathbb{C}_1^0$.

For the inductive step, suppose that for $i > 0$, $\mathbb{P}^{i-1}$ is a valid partition of $\mathcal{D}$ with respect to $\mathbb{C}_1^{i-1}$. For each block $b \in \mathbb{P}^{i-1}$, two cases are possible: (1) b is not split by C^i, for any $C \in \mathbb{C}$. Then b is valid with respect to $\mathbb{C}_1^i$, and will be retained in $\mathbb{P}^i$. (2) b is split by one more constraints C^i. The algorithm iterates through all such constraints that split b, and partitions block b such that every resulting block is valid with respect to each C^i, $C \in \mathbb{C}$.

We next note that $\mathbb{P}^i$ is indeed a partition of $\mathcal{D}$ (i.e. the union of all blocks equals $\mathcal{D}$). To see this observe that each block $b \in \mathbb{P}^{i-1}$ is either present in $\mathbb{P}^i$ or has been refined and all its constituent blocks (whose union equals b) are in $\mathbb{P}^i$. Thus, $\mathbb{P}^i$ is a valid partition with respect to $\mathbb{C}_1^i$. This proves the inductive step. □

Consistency Constraints. Since different sub-views can have common attribute(s), additional constraints need to be added to ensure that their distributions for the common attribute(s) are the same. In order to do so, we may need to further refine the partition generated from the above procedure. Specifically, consider a pair of sub-views s_1 and s_2 with attribute sets $\mathbb{A}_1$ and $\mathbb{A}_2$ respectively, such that $\mathbb{A}_1 \cap \mathbb{A}_2 \neq \emptyset$. Let $\mathcal{D}^1 = \prod_{i \in \mathbb{A}_1} \mathcal{D}_i$, and $\mathcal{D}^2 = \prod_{j \in \mathbb{A}_2} \mathcal{D}_j$ be the corresponding domains for s_1 and s_2 respectively, and $\mathcal{D}^{1,2} = \prod_{i \in \mathbb{A}_1 \cap \mathbb{A}_2} \mathcal{D}_i$. Let the partitions obtained on $\mathcal{D}^1$ and $\mathcal{D}^2$ be $\mathbb{P}_1$ and $\mathbb{P}_2$, respectively. In order to keep $\mathbb{P}_1$ and $\mathbb{P}_2$ consistent with each other, we need to ensure that their region boundaries are aligned with each other, and this is achieved by refining $\mathbb{P}_1$ and $\mathbb{P}_2$ so that they have common boundaries along dimensions $\mathbb{A}_1 \cap \mathbb{A}_2$. We consider the union of the "split points" of $\mathbb{P}_1$ and $\mathbb{P}_2$ along dimensions $\mathbb{A}_1 \cap \mathbb{A}_2$ and further for each block in $\mathbb{P}_1$ (and $\mathbb{P}_2$), we refine this block until it no longer crosses such a split point. Finally, we add LP constraints that equate distributions of the common attributes in $\mathbb{P}_1$ and $\mathbb{P}_2$.

5 DATABASE SUMMARY GENERATOR

This component takes the LP solution for each view as the input and generates the database summary, which as mentioned previously, can be used for dynamically generating data for query execution, or can optionally be used to generate the materialized database.

Recall that a variable in the LP (for a view) represents an underlying block in a sub-view's partition, and its assigned value is the number of rows present in that block – this value is hereafter referred to generically as NumTuples. The collection of NumTuples values represent the sub-view solutions, and these solutions are integrated to obtain the solution for the complete view. However, since each view is solved independently, the referential constraints that exist between the corresponding relations may be lost in these view solutions. Therefore, they may have to be modified to ensure global consistency. Finally, it is necessary to extract relations from the views in order to populate the database. Accordingly, the summary generator component in Hydra is responsible for the following sequence of tasks:

(1) Constructing a solution for complete views
(2) Instantiating view summaries
(3) Making view summaries consistent wrt each other
(4) Extracting relation summaries from view summaries

5.1 Constructing Solution for the View

For integrating the sub-view solutions to obtain the collective solution for the complete view, we first *order* the sub-views. Then, we iteratively build the view-solution by *aligning* and *merging* the next sub-view solution in the given order. Let $\mathbb{S}$ denote the input list of sub-view solutions, and $viewSol$ be the final view solution that we wish to compute. Algorithm 3 describes the high-level process for constructing $viewSol$ from $\mathbb{S}$, and its ordering, aligning and merging procedures are described in the remainder of this sub-section.

Algorithm 3: View Solution Construction

```
1 𝕊 ← OrderSubViews(𝕊);
2 viewSol ← ∅;
3 foreach s ∈ 𝕊 do
4     viewSol, s ← Align(viewSol, s) ;
5     viewSol ← Merge(viewSol, s);
```

5.1.1 Sub-View Ordering. Ordering is implemented through a greedy iterative algorithm where we can start with any sub-view as the first choice. Subsequently, at iteration i, let the set of visited sub-views until now be $\mathbb{S}$. A sub-view s from outside this set can be chosen to be the next in the ordering only if it satisfies the following condition: On removing the common vertices between s and $\mathbb{S}$ in the (chordal) view-graph, there should not exist any path between the remaining vertices of s and the remaining vertices of $\mathbb{S}$. This algorithm is described in detail in [21].

5.1.2 Aligning. After obtaining the sub-view merge order as per above, in every iteration we merge the next sub-view solution (s) in the sequence to the current view-solution ($viewSol$), after a process of alignment. The alignment algorithm is a two step exercise, as shown in the example of Figure 8:

Solution Sorting: First, the $viewSol$ and s solutions are each sorted on their common set of attributes to facilitate direct comparison of their matching ranges. For instance, the solutions A, B and A, C in Figure 8a are each sorted on the intervals enumerated in the common attribute A.

Row Splitting: Our addition of consistency constraints during the LP formulation ensured that the distribution of tuples along the common set of attributes is the same in the various sub-views. Therefore it easy to see that the sum of NumTuples values in any interval of the common attributes is the same for the sub-view solutions under alignment. For example, in Figure 8a, the total number of tuples with $A = [40, 60)$ is 30K in both the A, B and A, C solutions. Likewise, the other entries in column A also have matching total number of tuples across the solutions. The align step *splits* the rows in these solutions such that the corresponding rows in both solutions have the same number of tuples. The sub-view solutions of Figure 8a are shown in Figure 8b after undergoing the alignment process, with both solutions now having identical NumTuples in the corresponding rows.

A	B	NumTuples
[60, inf)	[0,inf)	30K
[40, 60)	[0, 5) U [15, inf)	20K
[40, 60)	[5, 15)	10K
[20, 40)	[5, 10)	10K
[20, 40)	[15, inf)	10K

A	C	NumTuples
[60, inf)	[0, inf)	30K
[40, 60)	[2, 3)	30K
[20, 40)	[0, 2) U [3, inf)	20K

(a) Sub-view Solution

A	B	NumTuples
[60, inf)	[0,inf)	30K
[40, 60)	[0, 5) U [15, inf)	20K
[40, 60)	[5, 15)	10K
[20, 40)	[5, 10)	10K
[20, 40)	[15, inf)	10K

A	C	NumTuples
[60, inf)	[0, inf)	30K
[40, 60)	[2, 3)	20K
[40, 60)	[2, 3)	10K
[20, 40)	[0, 2) U [3, inf)	10K
[20, 40)	[0, 2) U [3, inf)	10K

(b) View Alignment

A	B	C	NumTuples
[60, inf)	[0,inf)	[0, inf)	30K
[40, 60)	[0, 5) U [15, inf)	[2, 3)	20K
[40, 60)	[5, 15)	[2, 3)	10K
[20, 40)	[5, 10)	[0, 2) U [3, inf)	10K
[20, 40)	[15, inf)	[0, 2) U [3, inf)	10K

(c) Merged View Solution

Figure 8: Align and Merge Example

5.1.3 Merging. This is the last step in the construction of the view solution. Here we simply merge the two solutions obtained after alignment through a "position" based join, where the physically corresponding rows in each solution are combined, with the common attributes being represented once. For example, the aligned solutions of Figure 8b are merge-joined using the positions (or row identifiers) to deliver the final view solution of Figure 8c.

As discussed earlier, DataSynth adopted a sampling algorithm for constructing the view solutions post LP solving. In marked contrast, Hydra *deterministically* generates the view solutions, facilitating us to operate purely in the summary space. There are

two tangible benefits of this deterministic strategy: (a) elimination of the time and space overheads due to sampling, and (b) elimination of sampling-based errors in satisfying CCs.

5.2 Instantiating View Summaries

As shown in Figure 8c, each row in the view solution is comprised of a series of intervals (across various attributes) and the number of tuples in the region represented by these intervals. We now need to decide as to how these tuples are distributed *within* the attribute intervals. Our current solution is very simple: Assign the *entire* cardinality to the *left boundaries* of the intervals. For example, the third row in Figure 8c would result in generation of 10000 tuples all having $A = 40, B = 5, C = 2$ values.

Note that, in principle, we could have used a more sophisticated cardinality distribution within the intervals. However, our simple deterministic choice helps to reduce the subsequent additive errors that are incurred while ensuring referential integrity across views (described in next subsection). This is so because choosing values deterministically within a bucket minimizes the likelihood of encountering an fk value that is not present in the corresponding pk column.

5.3 Making View Summaries Consistent

Since the solution for each view is obtained independently, there could be inconsistencies across them. For example, referring back to the view schema shown in Section 3.2, R_view has attributes borrowed from S_view and T_view, and its solution may feature values that are not present in the corresponding attributes of these two views. To address this problem, we first carry out a *topological sort* on the "referential dependency graph"[2] and then iteratively make the current view consistent with its predecessors. Since a topological sort is employed, Hydra can handle dependency graphs that are DAGs unlike DataSynth which is restricted to tree traversals.

To make a pair of views V_i and V_j consistent with each other, where V_i is dependent on V_j, we iterate over the rows in the view solution of V_i and look for the value combination that each row has for the attributes borrowed from V_j. If that value combination is not present in the solution of V_j, we add a new row in its solution with the corresponding NumTuples attribute set to 1. This results in an additive error in the total number of tuples in the view as compared to the original AQP at the client. But we hasten to add that the error is a fixed number of rows, determined by the nature of the constraints and the LP solution, and *not* by the data scale. Therefore, at Big Data volumes, the discrepancy can be expected to be minuscule, and our experiments empirically confirm this expectation.

The inter view consistency component is present in DataSynth as well, but since its view solutions are comprised of complete database instantiations, and not just summaries, the time and space overheads incurred for making the views consistent can be large. Moreover, the additive error in DataSynth is amplified due to its inherent sampling errors. Our experiments also capture this distinction between the errors incurred due to referential constraints in Hydra and DataSynth.

5.4 Constructing Relation Summaries

After constructing consistent solutions across all the views, we next need to obtain the corresponding relation summaries. For this, we create a summarized relation schema $\widetilde{R_i}$ for each relation R_i. This schema consists of all attributes in R_i except the primary key attribute, and additionally, the NumTuples value for each entry in $\widetilde{R_i}$, as sourced from the view solutions.

For the common attributes between the summarized relation and the corresponding view solution, the value combinations and corresponding NumTuples value are directly borrowed from the solution. What remains are the foreign key attributes. For filling a foreign key attribute fk, we need to first consult the view corresponding to fk's s target relation, say V_j. To fill the fk value in row r of $\widetilde{R_i}$, we extract the value combination in row r of view solution of V_i, and then project the attributes corresponding to V_j – let this be denoted by v. Now, we iterate over the solution set of V_j and compute the cumulative sum of the cardinality entries till v is reached. This sum provides the fk value corresponding to the rth row of $\widetilde{R_i}$, and we thus obtain $\widetilde{R_i}$ for each relation R_i.

The set of relation summaries, computed as described above, provides the entire database summary – a sample such summary was previously shown in Figure 5 (for simplicity, the figure shows the PK columns instead of the number of tuples).

Like before, DataSynth again iterates over the complete instantiated (consistent) views to construct the corresponding materialized relations. Obviously, this leads to enormous time and space overheads in contrast to our data-scale independent summary based approach.

6 TUPLE GENERATOR

The Tuple Generator component resides inside the database engine, and needs to be explicitly incorporated in the engine codebase by the vendor. As a proof of concept, we have implemented it for the PostgreSQL v9.3 engine by adding a new feature called *datagen*, which is included as a property for each relation in the database. Whenever this feature is enabled for a relation, the scan operator for that relation is replaced with the dynamic generation operator. As a result, during query execution, the executor does not fetch the data from the disk but is instead supplied by the Tuple Generator in an *on-demand* manner, using the available relation summary.

Each row in the relation summary has a value combination and an associated NumTuples entry. We consider the pk values to be the row numbers of the relation. Therefore, to get the rth tuple of a relation R, the pk is chosen as r and the rest of the attributes come from the relation summary. We iterate over the rows of $\widetilde{R}$ and take the cumulative sum of the NumTuples entries until the sum exceeds r. Say the summation crosses the value r in jth row of $\widetilde{R}$. Then the rest of the values of the rth tuple are assigned to be precisely the same as those present in the jth row of $\widetilde{R}$. For example, the 120th row of relation S in Figure 5, would be $\langle 120, 20, 15 \rangle$.

Note that this form of tuple generation is expected to be efficient since the attribute value assignments are deterministic and independent, and these expectations are confirmed in the experiments shown in the following section.

7 EXPERIMENTS

We have implemented the Hydra design, described in the previous sections, in a Java tool running to over 15K lines of code. The popular Z3 [14] solver is leveraged to compute solutions for the LP formulations. In this section, we evaluate Hydra's empirical performance, using our implementation of DataSynth as the comparative yardstick in the analysis.

[2] A graph where each relation is represented by a node and an edge (u, v) is added if relation u is dependent on relation v through a referential constraint.

Database Environment. The TPC-DS [1] decision-support benchmark database, with a default size of 100GB, is used as the baseline in our experiments. The database is hosted on a PostgreSQL v9.3 engine [4] with the hardware platform being a vanilla HP workstation (3.2 GHz 16 core processor, 32 GB memory, 500 GB SSD hard drive) running Ubuntu Linux 16.04.3.

A complex query workload, WL_c, featuring 131 distinct queries (enumerated in [21]), was created by customizing the 99 queries of the benchmark such that only non-key filter predicates and PK-FK joins were retained, and all nested queries were separated into independent sub-queries[3]. The AQPs for these queries were generated on the PostgreSQL query processor, resulting in 351 cardinality constraints. The distribution of the cardinalities for these CCs are shown in Figure 9, with the cardinalities measured on a log-scale. The figure clearly indicates that a wide range of cardinalities are present in the constraints, going from a few tuples to almost a billion.

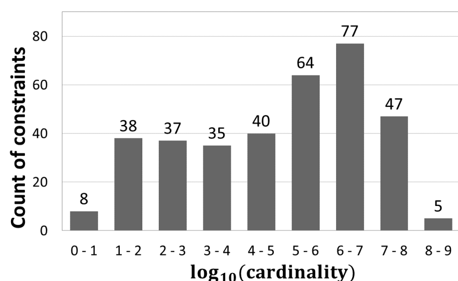


Figure 9: Distribution of Cardinality in CCs (WL_c)

The above constraints result in a large number of geometrically overlapping regions. Hydra, due to its region-partitioning approach, comfortably handles this scenario. In marked contrast, DataSynth, due to its grid-partitioning construction, generates a very large number of LP variables (in the several billion) from the constraints, overwhelming the solver's capabilities. We therefore also created an alternative simplified query workload, called WL_s, with 311 CCs, wherein the variables created by DataSynth were less than a million, and therefore well within the solver's processing power.

7.1 Quality of Volumetric Similarity

We begin by investigating how closely the volumetric similarity, with regard to operator output cardinalities, is achieved between the client and vendor sites for the WL_s workload by the Hydra and DataSynth regenerators. This behavior is captured in Figure 10, which plots the percentage of CCs that are within a given relative error of volumetric similarity. From the plot it is evident that Hydra satisfies around 90 percent of the CCs with virtually no error, and the remaining CCs are also satisfied within a relative error of less than 10%. This is in contrast to DataSynth, which accurately satisfies around 80 percent of the CCs, but then incurs as much as 60% relative error to achieve complete coverage of the remaining CCs.

There are two reasons for the error-prone behavior of DataSynth: (1) the probabilistic sampling technique, and (2) the maintenance of referential integrity. While Hydra also is forced to

[3]Similar to DataSynth, the restriction to non-key-based filters is because the conversion from relations to views lose the key attributes. Likewise, only PK-FK joins are supported since they are inherently present in the design of views.

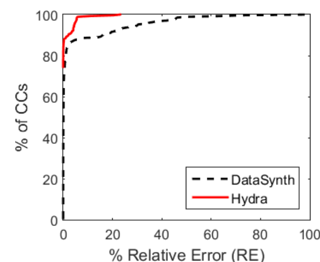


Figure 10: Quality of Volumetric Similarity (WL_c)

insert additional tuples to maintain referential integrity, the number is substantially smaller than those injected by DataSynth. This is because the integrity errors are *amplified* by the impact of the sampling errors. This effect is quantified in Figure 11, where the number of extra tuples inserted is plotted on a log-scale for representative TPC-DS tables. We see here that Hydra is often an *order-of-magnitude* smaller with regard to the addition of these extra tuples as compared to DataSynth. Also, recall that integrity errors in Hydra are independent of the data scale and therefore are minuscule at Big Data volumes. We also show this in [21].

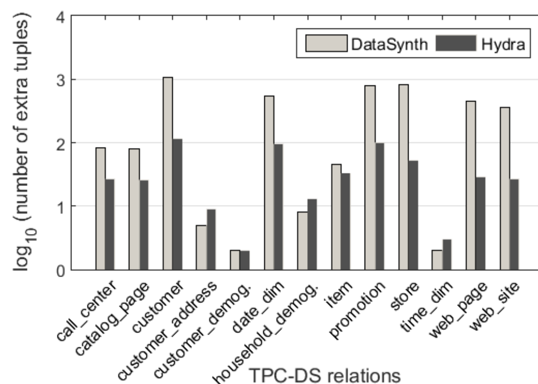


Figure 11: Extra tuples for Referential Integrity (WL_c)

As a final observation, it is interesting to note that DataSynth has to contend with both *negative* (volumes less than desired) and *positive* (volume greater than desired) relative errors, due to its sampling strategy – in fact, about one-third of the CCs suffered negative relative errors. In contrast, Hydra only generates positive errors due to the inclusion of extra tuples for satisfying referential integrity. From a practical standpoint, it is perhaps preferable to have positive errors since they induce greater stress on the data processing elements in the engine.

7.2 Scalability with Workload Complexity

We now turn our attention to evaluating the complexity of the underlying LP that is formulated by Hydra and DataSynth. Since LP complexity is essentially proportional to the number of variables in the problem, we compare this number for the two techniques. Further, since LP complexity is, to the first degree of approximation, independent of the database size, we present the comparison only for the 100 GB instance. [4] The number of LP variables for a representative set of TPC-DS relations, including the major fact and dimension tables (catalog_sales, store_sales,

[4]Of course, the database engine's choice of query plans may change to some extent with database size, leading to a slightly different set of CCs.

item) is captured, on a log-scale, in Figure 12 for the WL_c complex workload. We observe here that the LPs formulated using the region-partitioning strategy in Hydra generate *several orders of magnitude* fewer variables than the corresponding LPs derived from the grid-partitioning in DataSynth. As a case in point, consider the catalog_sales table – the number of variables created by DataSynth was almost 5.5 million, which is reduced to as low as 1620 by Hydra. Even more dramatic is the change for item table, where the number of variables is reduced from an enormous 10^{11} to around 3700.

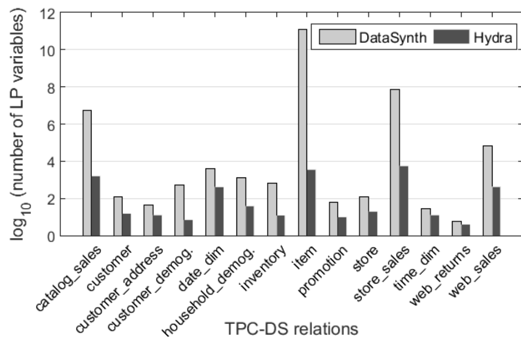


Figure 12: Number of variables in the LP (WL_c)

From an absolute perspective also, the large number of variables created by DataSynth is a critical problem since, as mentioned previously, the LP solver crashed in handling these cases. In marked contrast, the few thousands of LP variables generated by Hydra were easily solvable in less than a minute. Moreover, even when we switched to the simple workload, WL_s, the LP solution time for DataSynth was almost an hour, whereas Hydra completed in a few seconds as shown in Figure 13.

Complex Workload (WL_c)		Simple Workload (WL_s)	
DataSynth	**Hydra**	**DataSynth**	**Hydra**
crash	58 sec	50 min	13 sec

Figure 13: LP Processing Time

7.3 Scalability with Materialized Data Size

This experiment compares the data instantiation times, post LP solution, of DataSynth and Hydra on the WL_s workload. While Hydra, in principle, due to its summary-based approach, does *not* have to instantiate the data immediately, we assume in this experiment that the vendor requires complete materialization.

The experimental results are shown in Figure 14, where we also present, for comparative purposes, the performance with 10 GB and 1000 GB databases, apart from the default 100 GB database. We see here that there is a huge reduction in the materialization time of Hydra at all scales. Further, even in absolute terms, Hydra is able to output a 100 GB database in around 11 minutes, whereas DataSynth takes 42 hours to complete the same task.

The marked difference in the efficiency of the two techniques is attributed to the fact that DataSynth instantiates complete views through sampling, subsequently performs *several passes* on these instantiations to ensure referential integrity, and to derive relations from them. Hydra on the other hand, after LP-solving, constructs the database summary in just a few seconds, and then instantiates the materialized database directly from it.

Size (in GB)	DataSynth	Hydra
10	4 hours	2 min
100	42 hours	11 min
1000	> 1 week	1.6 hours

Figure 14: Data Materialization Time

7.4 Scalability to Big Data Volumes

In our next experiment, we validated the ability of Hydra, thanks to its summary-based technique, to scale to Big Data volumes. To demonstrate this feature, we modeled an exabyte-sized (10^{18} bytes) data scenario as follows: We used CODD, which is capable of modeling arbitrary metadata scenarios, to obtain the optimizer-chosen plans at the exabyte database scale for all the workload queries. To get AQPs for this database, we executed the obtained plans on the 100 GB instance and scaled the intermediate row counts with the appropriate scale factor. Hydra was able to formulate and solve the LPs (one per relation), and generate the database summary in less than **2 minutes**. Once the summary is generated, the database can begin to submit the workload queries since the data required for the execution can be produced on-the-fly by the Tuple Generator.

7.5 Dynamism in Data Generation

Our next experiment evaluates Hydra's ability, due to the Tuple Generator and Database Summary architecture, to produce tuples *on-the-fly* instead of first materializing them, and then reading from the disk. To verify whether dynamic generation can indeed produce data at rates that are practical for supporting query execution, we compared the total time that Hydra's tuple generator took to construct and supply tuples to the executor, while running simple aggregate queries, as compared to the standard sequential scan from the disk.

Rel. Name	Size	Row count	Scan time (secs)	
	(in GB)	(in millions)	**Disk**	**Dynamic**
store_returns	3	29	16	8
web_sales	10	72	43	25
inventory	19	399	107	74
catalog_sales	20	144	46	48
store_sales	34	288	168	87

Figure 15: Data Supply Times

The results of this experiment are shown in Figure 15 for the five biggest relations in the 100 GB database instance. We see here that the tuple generator is not only competitive with a materialized solution, but is in fact typically *faster*. Therefore, using dynamic generation can prove to be a good option since it can help to eliminate the large time and space overheads incurred in: (1) dumping generated data on the disk, and (2) loading the data on the engine under test.

7.6 Performance on JOB Benchmark

A legitimate concern with regard to the above encouraging results for Hydra is that they may be an artifact of the TPC-DS database, and perhaps might under-perform on other datasets. To address this concern, we consider in our final experiment, a schematically highly different database, namely the JOB benchmark [17], which is based on the IMDB real-world dataset. Here, we created a

workload of 260 queries, resulting in 523 CCs, whose cardinality distribution is again highly varied as seen in Figure 16.

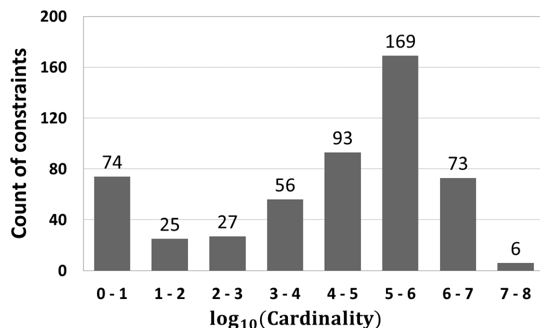


Figure 16: Cardinality distribution of CCs in JOB

We found that Hydra efficiently solved this workload as well, with the number of variables in each view being typically in the few thousands, and never exceeding a hundred thousand, as shown quantitatively in Figure 17. The overall database summary was quickly generated in around 20 seconds, and produced a database of high fidelity that satisfied all the constraints with no more than *2 percent* relative error.

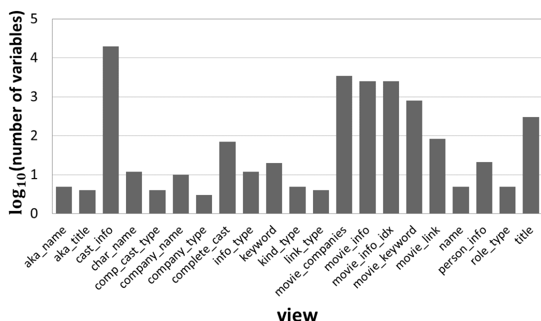


Figure 17: Number of Variables for JOB

8 RELATED WORK

Over the past few decades, a rich corpus of literature has developed on synthetic database construction. There are two broad streams of research on the topic, one dealing with the *ab initio* generation of new databases using standard mathematical distributions (e.g. [12, 15]), and the other with *regeneration* of an arbitrary existing database. In the latter category, there are two approaches, one of which uses only schematic and statistical information from the original database (e.g. [19, 22]). The other uses both the original database and the query workload to achieve statistical fidelity during evaluation (e.g [6, 11]) – our work on Hydra falls into this class. In this section, we briefly review recent literature on this spectrum of research categories.

Ab Initio Generation. Descriptive languages for the definitions of data dependencies and column distributions were proposed in [12, 16, 20]. For example, [12] proposed a special purpose language called Data Generation Language (DGL) that is used by the tool to generate synthetic data distributions by utilizing the concept of iterators. It supports a broad range of dependencies between relations but the construction of dependent tables always requires access to the referenced table, creating a bottleneck on the data generation speed.

In contrast to the above, MUDD [23] and PSDG [16] generate all related data at the same time. However, this approach can also be rendered inefficient if the referenced tables are large in size. MUDD proposes algorithms to parallelize the data generation process, and to efficiently generate dense-unique-pseudo-random sequences and derive nonuniform distributions. Both MUDD and PSDG decouple data generation details from data description, facilitating customization of the tool to suit user needs.

In the distributed setting, a faster way of generating references is through recomputing since it eliminates the I/O costs incurred to satisfy referential constraints across relations that are present across different nodes. PDGF [20] was designed with this goal of achieving scalability and decoupling. In PDGF, the user specifies two XML configuration files, one for the data model and one for the formatting instructions. The generation strategy is based on the exploitation of determinism in pseudo-random number generators (PRNG), which enables regeneration of the same sequences, hence eliminating the scan overheads. PDGF supports the generation of data with cyclic dependencies as well, but incurs high computation costs for generating the associated keys. Finally, PDGF comes with a set of fixed generators for different datatypes and basic distribution functions.

A similar generator is Myriad [5], which implements an efficient parallel execution strategy leveraged by extensive use of PRNGs with random access support. With these PRNGs, Myriad distributes the generation process across the compute nodes and ensures that they can run independently from each other, without imposing any restrictions on the data modeling language.

Finally, a rule-based probabilistic approach, based on an extension of Datalog, has been recently proposed in [9], which is capable of generating data characterized by parametrized classical discrete distributions – however, it is not always feasible to assign such distributions to real-world data, especially over multivariate spaces.

Database-dependent Regeneration. DBSynth[19] is an extension to PDGF, which builds data models from an existing database by extracting schema information, and using sampling to construct histograms and dictionaries of text-valued data. Further, if the textual data contains multiple words, Markov chain generators are used to analyze the word combination frequencies and probabilities. Finally, after the model construction is complete, PDGF is invoked to generate the corresponding data.

Like DBSynth, RSGen [22] takes a metadata dump, including 1-D histograms, as the input, and generates database tables along with a loading script as the output. It uses a bucket based model at its core, which is able to generate trillions of records with minimum memory footage. However, the proposed technique works well only for queries with only a single range predicate. Further, due to the inaccurate statistical models in the query optimizer, the volumetric similarity is poor for queries involving predicates on correlated attributes.

UpSizeR [24] is a graph-based tool that uses attribute correlations extracted from an existing database to generate an equivalent synthetic database. A derivative work, Rex [13] produces an extrapolated database given an integer scaling factor and the original database, while maintaining referential constraints and the distributions between the consecutive linked tables. Dscaler [26]

addresses the problem of generating a non-uniformly[5] scaled version of a database using fine-grained, per-tuple correlations for key attributes, but such information is typically hard to come by. Moreover, all these techniques only generate the *key* attributes, whereas the non-key values are sampled from the original database using these key values. Hence, the approach becomes impractical in Big Data and security-conscious environments. Finally, Dscaler fails to retain accuracy for some common query classes.

Query-dependent Regeneration. Apart from the above techniques, another line of work [6, 10, 11, 18] is based on workload dependence (as in the case of Hydra). Here the aim is to generate a database given a workload of queries such that volumetric similarity is achieved on these queries. In particular, RQP [10] gets a query and a result as input, and returns a possible database instance that could have produced the result for that query. The idea of using cardinalities from a query plan tree was first introduced in QAGen [11]. They start by constructing a *symbolic database*[6], and then translate the input AQPs to constraints over the symbols in the database. Subsequently, a constraint satisfaction program (CSP) is invoked to identify values for symbols that satisfy all the constraints.

On the positive side, these generators are capable of handling complex operators as they use a general CSP, but the performance cost is huge since the number of CSP calls also increases with the database size. Further, it requires operating on a symbolic database of matching size to the original database, and processing of the entire database during the algorithm execution. This makes it impractical for Big Data environments. Finally, QAGen supports only one query plan in the input. This limitation was addressed in a follow-up tool called MyBenchmark [18], which creates a symbolic database on a per query basis and at the end tries to heuristically merge the various databases into a small number of databases. Clearly, generating a database on a per query basis has enormous time and space overheads, and further, a single database is not guaranteed in the output.

DataSynth [6] identified the declarative property of cardinality constraints and its ability to specify data characteristics. Given a large number of cardinality constraints as input, the paper proposed algorithms based on the LP solver and graphical models to instantiate tables that satisfy those constraints. However, it suffers from high LP complexity, data scale dependencies, and inaccuracies with regard to volumetric similarity, as we have discussed in this paper. Hydra materially extends the DataSynth approach by adding dynamism, scale and functionality.

9 CONCLUSIONS

The ability to synthetically regenerate data that accurately conforms to the volumetric behavior on queries at client sites is of crucial importance to database vendors, and will become even more so with the advent of Big Data applications. In this paper, we have proposed Hydra, a data regeneration tool that takes a substantial step forward towards achieving this goal. Specifically, by reworking the basic LP problem formulation into a region-based variable assignment, Hydra improves on the state-of-the-art DataSynth's performance by orders of magnitude with regard to problem complexity, data materialization time, and scalability to large volumes. Secondly, by using a deterministic alignment technique for database consistency, it provides far better accuracy in meeting volumetric constraints as compared to the probabilistic approach employed in DataSynth. Finally, its summary-based framework organically supports the dynamic regeneration of streaming data sources, an essential pre-requisite for efficiently testing contemporary deployments.

In our future work, we plan to focus on covering a richer set of query operators, such as grouping functions, within the Hydra framework. Also, we would like to investigate how to leverage additional summary information (such as value-based correlations) that the client might be willing to provide for achieving stronger fidelity with the original database.

Acknowledgements. We thank the anonymous reviewers for their expert and constructive comments on the material presented here. We also thank Huawei Technologies India Pvt. Ltd. and the members of the Database Systems Lab at IISc for their valuable feedback and support in this work.

[5]In non-uniform scaling, individual tables are scaled by different factors.

[6]A symbolic database is similar to a regular database, but its attribute values are symbols (variables), not constants.

REFERENCES

[1] TPC-DS. http://www.tpc.org/tpcds/.
[2] TPC-H. http://www.tpc.org/tpch/.
[3] USE PLAN SQL Server. https://technet.microsoft.com/en-us/library/ms186954(v=sql.105).aspx.
[4] PostgreSQL. http://www.postgresql.org/docs/9.3/static/release.html.
[5] A. Alexandrov, K. Tzoumas and V. Markl. Myriad: Scalable and Expressive Data Generation. *PVLDB*, 5(12), 2012.
[6] A. Arasu, R. Kaushik and J. Li. Data generation using declarative constraints. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2011.
[7] A. Arasu, R. Kaushik and J. Li. DataSynth: Generating synthetic data using declarative constraints. *PVLDB*, 4(12), 2011.
[8] S. Ashoke and J. R. Haritsa. CODD: a dataless approach to big data testing. *PVLDB*, 8(12), 2015.
[9] V. Barany, B. Cate, B. Kimelfeld, D. Olteanu and Z. Vagena. Declarative Probabilistic Programming with Datalog. *Proc. of the 19th Intl. Conf. on Database Theory*, 2016.
[10] C. Binnig, D. Kossmann and E. Lo. Reverse Query Processing. *Proc. of the 23rd Intl. Conf. on Data Engineering*, 2007.
[11] C. Binnig, D. Kossmann, E. Lo and M. Tamer Özsu. QAGen: generating query-aware test databases. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2007.
[12] N. Bruno and S. Chaudhuri. Flexible database generators. *Proc. of the 31st Intl. Conf. on Very Large Data Bases*, 2005.
[13] T. S. Buda, T. Cerqueus, J. Murphy and M. Kristiansen. ReX: Extrapolating Relational Data in a Representative Way. *Proc. of the British Intl. Conf. on Databases*, 2015.
[14] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
[15] J. Gray, P. Sundaresan, S. Englert, K. Baclawski and P. J. Weinberger. Quickly Generating Billion-record Synthetic Databases. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.
[16] J. E. Hoag and C. W. Thompson. A parallel general-purpose synthetic data generator. *ACM SIGMOD Record*, 2007.
[17] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3), 2015.
[18] E. Lo, N. Cheng, W. W. K. Lin, W. Hon and B. Choi. MyBenchmark: generating databases for query workloads. *The VLDB Journal*, 23(6), 2014.
[19] T. Rabl, M. Danisch, M. Frank, S. Schindler and H. Jacobsen. Just Can'T Get Enough: Synthesizing Big Data. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2015.
[20] T. Rabl, M. Frank, H. M. Sergieh and H. Kosch. A Data Generator for Cloud-scale Benchmarking. *Proc. of the 2nd TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems*, 2010.
[21] A. Sanghi, R. Sood, J. R. Haritsa and S. Tirthapura. Scalable and Dynamic Workload Dependent Data Regeneration. *Tech. Report TR-2017-01*, DSL/CDS, IISc, 2017, dsl.cds.iisc.ac.in/publications/report/TR/TR-2017-01.pdf.
[22] E. Shen and L. Antova. Reversing statistics for scalable test databases generation. *Proc. of the 6th Intl. Workshop on Testing Database Systems*, 2013.
[23] J. M. Stephens and M. Poess. MUDD: A Multi-dimensional Data Generator. *Proc. of the 4th Intl. Workshop on Software and Performance*, 2004.
[24] Y. C. Tay, B. T. Dai, D. T. Wang, E. Y. Sun, Yong Lin and Yuting Lin. UpSizeR: Synthetically Scaling an Empirical Relational Database. *Inf. Syst.* 38(8), 2013.
[25] R. S. Trivedi, I. Nilavalagan and J. R. Haritsa. Codd: Constructing dataless databases. *Proc. of the 5th Intl. Workshop on Testing Database Systems*, 2012.
[26] J. W. Zhang and Y. C. Tay. Dscaler: Synthetically scaling a given relational database. *PVLDB*, 9(14), 2016.