Data Generation using Projection Constraints

Anupam Sanghi Shadab Ahmed Jayant R. Haritsa

Technical Report
TR-2021-03
(May 2021)

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

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

Abstract

A core requirement of database engine testing is the ability to generate synthetic databases that exhibit a desired set of characteristics. Expressing these characteristics through declarative formalisms has been advocated in contemporary testing frameworks. In particular, specifying operator output volumes through row-cardinality constraints has received considerable attention. However, thus far, adherence to these volumetric constraints has been limited to only the Filter and Join operators. A critical deficiency is the lack of support for the Projection operator, which forms the core of basic SQL constructs such as Distinct, Union and Group By. The technical challenge here is that cardinality *unions* in multi-dimensional space, and not mere summations, need to be captured in the generation process. Further, dependencies *across* different data subspaces need to be taken into account.

In this paper, we address the above lacuna by presenting **PiGen**, a dynamic data generator that incorporates Projection cardinality constraints in its ambit. The design is based on a projection subspace division strategy which supports the expression of constraints using optimized linear programming formulations. Further, techniques of symmetric refinement and workload decomposition are introduced to handle constraints across different projection subspaces. Finally, PiGen supports dynamic generation, where data is generated on-demand during query processing, making it amenable to Big Data environments. A detailed evaluation on TPC-DS-based query workloads demonstrates that PiGen can accurately and efficiently model Projection outcomes, representing an essential step forward in customized database generation.

1 Introduction

Database software vendors often need to generate synthetic databases for a variety of applications [19, 8], including: (a) Testing of database engines and applications, (b) Data masking, (c) Benchmarking, (d) Creating "what-if" scenarios, and (e) Assessing performance impacts of planned engine upgrades. The synthetic databases are targeted towards capturing the desired schematic properties (e.g. keys, referential constraints, functional dependencies, domain constraints), as well as the statistical data profiles (e.g. value distributions, column correlations, data skew, output volumes) hosted on these schemas.

1.1 Cardinality Constraints

The use of *declarative formalisms* to express data characteristics has been persuasively advocated in contemporary testing frameworks [8, 18, 22]. In particular, a *cardinality constraint* dictates that the output of a given relational expression over the generated database should feature the specified number of rows. For SPJ (SELECT-PROJECT-JOIN) formulations, the canonical representation in the constraint format is:

$$|\pi_{\mathbb{A}}(\sigma_f(T_1 \bowtie T_2 \bowtie ... \bowtie T_N))| = k \tag{1}$$

where f represents the *filter predicates* applied on the inner join of a group of tables $T_1, ...T_N$ in the database; $\mathbb{A}$ represents the *projection-attribute-set*, i.e. the set of attributes on which the projection is applied; and k is a count representing the output row-cardinality of the relational expression. The provenance of these constraints could be either from construction of *what-if* scenarios by the database vendor, or based on information sourced from a client installation – for instance, Annotated Query Plans (AQPs) [11]. Further, the constraints could be *parameterized* wrt predicate constants [19, 18], or more commonly in industrial practice, *strict*, where even these constants are prespecified [8, 22].

1.2 Data Generation using Cardinality Constraints

Generating synthetic data that adheres to a collection of strict cardinality constraints was first proposed in the pioneering work of **DataSynth** [8, 9]. This initial effort was later extended in **Hydra** [22, 23] to incorporate dynamism and scale in the generation process. The key idea in these frameworks is to express the input constraints using a *linear feasibility program* (LP), and then use the LP solution to construct the synthetic database. While these prior frameworks accurately and efficiently handle an important class of cardinality constraints, a critical lacuna is support for the *projection* operator. In this paper, we investigate the explicit incorporation of Projection into the data generation framework.

1.3 Incorporating Projection

Our motivation for modeling Projection stems from its core appearance in the DISTINCT, GROUP BY, and UNION SQL constructs – as a case in point, among the 22 queries in the TPC-H benchmark [5], as many as 16 feature the projection operation. Further, projection-compliant databases can be beneficial to database vendors in a variety of use-cases [1] in Software Testing and Tuning, and in Informed Decision Making. We outline below sample use-cases wherein projection-compliant databases can materially enhance the outcomes.

Regression Testing: In the context of engine upgrades, a critical requirement is to synthesize data that can mimic client environments for *regression testing*. This facility enables: (a) Catching optimizer bugs such as a change in query plan leading to performance degradation, or incorrect query rewriting leading to erroneous query results; (b) Performance evaluation of operators in the query execution pipeline. For instance, a thorough assessment of a new memory manager's ability to handle native projection-based operators (e.g. hash aggregate, sort) is predicated on accurate modeling of projection cardinalities; and (c) Given an operator of interest, evaluating its impact on the performance of downstream operations. For instance, in the 16 projection-featuring queries of TPC-H, 12 require a sort operation immediately following projection. Further, in 4 queries, the projection output serves as an intermediate staging for subsequent filter/join operations. In all these cases, the projection output cardinality affects the behavior of the downstream operations.

Execution Tuning: Due to the dynamic nature of production environments, it is often required to carry out on-the-fly platform tuning, especially with regard to system configurations or query execution plans. As a non-invasive and arm's length precursor, the DBA can evaluate the expected tuning impacts on a synthetic equivalent – here projection-compliance can be of particular utility since cardinality estimation techniques are known to have difficulty in accurately modeling projection outputs. For instance, the choice of projection implementation (e.g. index vs hash) can be customized based on the projection output on the synthetic database.

Application Testing: Organizations often outsource the testing of their database applications to other organizations. However, sharing the internal databases on which these applications operate may be infeasible due to privacy concerns. In these cases, testing the applications using the cardinality constraints derived from the database is a viable option. The queries embedded in these applications often feature projection-based SQL constructs. For instance, in the context of banking applications, a routine analytical query could be asking for the number of (distinct) bank accounts performing online transactions in remote areas. Therefore, by supporting PICs, PiGen can take the level of testing a step further.

[1] verified with industry experts in Dagstuhl Seminar [1].

System Benchmarking: When evaluating competing database platforms for hosting an application, carrying out the evaluation on an application-specific database is of much richer relevance as compared to a generic benchmark such as TPC-H or TPC-DS. Creating a synthetic database that models the application's environment allows for a detailed assessment of both current and future scenarios. In this context, greater realism of the synthetic data would help to make informed choices. Since PiGen supports the incorporation of projection constraints, which are fundamental to many applications, we can expect greater fidelity to the application's framework.

Our focus here is on the *duplicate-eliminating* version of projection where only the *distinct* rows are retained in the projected output (the alternative duplicate-preserving option does not alter the filter output's row-cardinality, and is therefore trivially handled by the existing frameworks). Additionally, since projection is a unary operator, we present the ideas using a single-table environment. To handle multi-relation environments, we can take recourse to the methodology of [8, 22], where denormalized relations are constructed as an intermediate step in the solution process.

1.4 Projection-inclusive Constraints

To represent a projection-inclusive cardinality constraint c on a table $\mathcal{T}$, we use the quadruple $\mathbf{c} : \langle \mathbf{f}, \mathbb{A}, \mathbf{l}, \mathbf{k} \rangle$, as a shorthand notation. Here, f represents the filter predicate applied on $\mathcal{T}$, $\mathbb{A}$ represents the projection attribute-set (PAS), l signifies the row-cardinality of the filtered table, and k represents the row-cardinality after projection on this filtered table.

As as sample instance, consider the following set of PICs on a generated table PURCHASES (PID, Qty, Amt, $Year$):

$$c_1 : \langle f_1, Amt, 500, \mathbf{5} \rangle \mid f_1 = (Qty < 20) \wedge (1100 \leq Amt < 2500)$$
$$c_2 : \langle f_2, Amt, 1000, \mathbf{3} \rangle \mid f_2 = (Qty \geq 20) \wedge (500 \leq Amt < 3000)$$
$$c_3 : \langle f_3, Qty, 3000, \mathbf{9} \rangle \mid f_3 = (Qty \geq 10)$$

Here, PIC c_1 denotes that applying the f_1 predicate on PURCHASES should produce 500 rows in the output, which is further reduced to 5 rows after projecting on the Amt column; the other PICs can be interpreted analogously.

1.5 Technical Challenges

There are two primary challenges to modeling PICs within the table generation process, related to handling dependencies within and across the data subspaces identified by these constraints, as described below.

Intra-Projection Subspace Dependencies. Consider the projection subspace spanned by a set of attributes $\mathbb{A}$. Dealing with projection requires computing union of groups of tuples. For example, for two tuples/group of tuples b_1 and b_2, the direct expression for computing projection along $\mathbb{A}$ is:

$$|\pi_{\mathbb{A}}(b_1 \cup b_2)|$$

However, even if b_1 and b_2 are disjoint in the original table, their projections onto $\mathbb{A}$ may *overlap*. Therefore, to handle PICs, explicitly computing the cardinality of the *union* of a group of tuples post-projection is required. Using the fact that projection distributes over union [25], we can rewrite the above expression as:

$$|\pi_{\mathbb{A}}(b_1) \cup \pi_{\mathbb{A}}(b_2)|$$

but even here the union does not translate to a simple summation. For instance, consider the following two sample rows from the PURCHASES table:

$u : (PID = 10001, Amt = 1500, Qty = 3, Year = 2020)$, and

$v : (PID = 10002, Amt = 1500, Qty = 16, Year = 2021)$.

Both rows satisfy the filter f_1, but the union of their projections along Amt yields a single outcome – namely, $Amt = 1500$.

Inter-Projection Subspace Dependencies. When a set of tuples b is subjected to multiple projections, the data generation for projection subspaces may be interdependent. Given a pair of PASs $\mathbb{A}_1$ and $\mathbb{A}_2$, sourced from two PICs, we have the inclusion property:

$$\pi_{\mathbb{A}_1 \cup \mathbb{A}_2}(b) \subseteq \pi_{\mathbb{A}_1}(b) \times \pi_{\mathbb{A}_2}(b)$$

For instance, consider a group of tuples b, from the table $Purchases$, satisfying the following disjunctive filter condition:

$$b = \{t \in Purchases \mid (t.Qty \geq 20 \wedge t.Amt \geq 3000) \vee (10 \leq t.Qty < 20 \wedge t.Amt \geq 2500)\}$$

Here, a tuple with $Amt = 2700$ and $Qty = 25$ can belong to both $\pi_{Amt}(b)$ and $\pi_{Qty}(b)$, but lies outside b's boundary.

Moreover, $\mathbb{A}_1$ and $\mathbb{A}_2$ may themselves intersect. Therefore, in general, expressing a set of PICs with an LP, while ensuring a physically constructible solution, is often infeasible – this is because the set of constructible solutions does not form a convex polytope [16]. Hence, alternative methods are needed to address this issue.

1.6 Our Contributions

We present here **PiGen**, a data generator that addresses the above challenges and extends the current scope of data generation to include projection in its ambit. The key design principles are: (a) *Projection Subspace Division*, which divides each projection subspace into regions that allow modeling the unions, thereby ensuring that the intra-subspace dependencies are resolved; and (b) *Isolating Projections*, for independent processing of each projection subspace, thereby tackling the inter-projection subspace challenge.

Additionally, PiGen leverages the concept of *dynamic regeneration* [22], and constructs an *Enriched Table Summary*, that ensures data can be generated on-demand during query processing while satisfying the input PICs. Therefore, no materialized table is required in the entire testing pipeline. Further, the time and space overheads incurred in constructing the summary is independent of the size of the table to be constructed and, in our evaluations, requires only a few 100 KBs of storage.

A detailed evaluation on multiple workloads of PICs, covering both real-world datasets (IMDB, Census), and synthetic benchmarks (TPC-DS) has been conducted. The results demonstrates that PiGen accurately and efficiently models Projection outcomes. As a case in point, for a workload of PICs, comprising over a hundred PICs in total, PiGen generated data that satisfied all the PICs, with *perfect accuracy*. Moreover, the entire summary production pipeline completed within viable time and space overheads.

Organization The remainder of the report is organized as follows: The prior literature is reviewed in Section 2. The problem framework is discussed in Section 3. Further, the key design principles of PiGen are introduced in Section 4, and then described in detail in Sections 5 through 8. The end-to-end implementation pipeline is presented in Section 9, while the experimental evaluation is reported in Section 10. Finally, our conclusions and future research avenues are summarized in Section 11.

2 Related Work

Over the past three decades, a variety of novel approaches have been proposed for synthetic database generation. The initial efforts (e.g. [15, 13]) focused on generating databases using standard mathematical distributions. Subsequently, data generation techniques that incorporated the notion of constraints were proposed – for instance, adherence to a given set of metadata statistics was addressed in [24, 20, 7]. In more recent times, generation techniques driven by constraints on query outputs have been analyzed. A particularly potent effort in this class was **RQP** [10], which receives a query and a result as input, and returns a minimal database instance that produces the same result for the query. An alternative fine-grained constraint formulation is to specify the row-cardinalities of the individual operator outputs, and the techniques advocated in [11, 19, 8, 22, 21, 18, 14] fall in this category. They can be classified into two groups based on the nature of constraints. In the first group, parameterized constraints form the input in **QAGen** [11], **MyBenchmark** [19] and **TouchStone** [18]. That is, the predicate constants are variables. From these constraints, these techniques generate a synthetic database and predicate instantiations, such that applying the instantiated constraints on the synthetic data produces the desired number of rows.

On the other hand, a stricter notion of fixed constraints was considered in [8, 22, 21, 14], where the predicate constants are prespecified in the input. This strict model helps to generate data that is (a) more directly representative of the source environment, and as a consequence (b) more robust to future queries outside of the original workload. However, while constraints with filter and join operators have been handled satisfactorily, support for the projection operator has been minimal, restricted to a few extreme cases. For instance, **DataSynth** [8] proposed a projection generator that catered to *single-column* tables. Here, due to the single-column restriction, there are by definition no intra/inter projection subspace dependencies. In contrast, in PiGen, we consider a general class of strict PICs, requiring us to explicitly address these challenges.

Complementary to the studies by the database community, the mathematical literature includes work such as [12, 26, 16], where they study the set of sanity constraints that need to be satisfied by a given set of projection results to ensure table constructibility. In this regard, a class of constraints called **BT** (Bollobás and Thomason) inequalities were proposed in [12], which capture the necessary conditions to be satisfied by projection output cardinalities. However, they are not sufficient, making it possible that no actual database can satisfy these values. Another class of constraints, called **NC** (non-uniform cover) inequalities, was proposed in [26]. These form sufficient conditions such that if the constraints are satisfiable, then a database construction is always possible. However, the limitation is that the satisfiability is not guaranteed. Further, the feasibility space does not exhibit a convex behaviour, and therefore, it cannot be expressed as a set of linear constraints [16]. To address these theoretical hurdles, PiGen incorporates the techniques of workload decomposition and symmetric refinement. Further, the set of sanity constraints added in the LP formulation ensure that the solution is always constructible within the assumptions.

3 Problem Framework

In this section, we summarize the basic problem statement, and the underlying assumptions of our PiGen solution.

Statement Given an input table schema S and a workload $\mathbb{W}$ of strict PICs on S, the objective of data generation is to construct a table $\mathcal{T}$, such that it conforms to S and satisfies $\mathbb{W}$.

Assumptions We assume that each PIC in $\mathbb{W}$ is of the form described in the Introduction, and that it is strict (i.e., with prespecified predicate constants). Further, for ease of presentation, we assume that $\mathbb{W}$ is *collectively feasible*, that is, there exists at least one legal database instance satisfying all the constraints – the infeasibility scenario is deferred to Section 9. Finally, for brevity, we present the ideas using tables with *continuous numeric* columns; the extension to other data types is straightforward.

Output Given S and $\mathbb{W}$, PiGen outputs a collection of table summaries. Each summary $s(\mathcal{T})$ can be used to deterministically produce the associated table $\mathcal{T}$. The tables produced are such that: (a) all of them conform to S, and (b) each input PIC in $\mathbb{W}$ is satisfied by at least one of them.

Notations The main acronyms and key notations used in the rest of the paper are summarized in Tables 1 and 2, respectively.

Table 1: Acronyms

Acronym	Meaning
PAS	Projection Attribute Set
PIC	Projection-inclusive Cardinality Constraint
FB	Filter Block
RB	Refined Block
PRB	Projected Refined Block
CPB	Constituent Projection Block
PSD	Projection Subspace Division

4 Design Principles

In this section we overview the core PiGen design principles, with the PURCHASES table of the Introduction used as the running example to explain their impact. Subsequently, in Sections 5 through 8, each principle is described in detail. To set the stage, here are some basic definitions underlying our work.

Definition 1. *A* block *is a bag of points (i.e. tuples) in the data space $\mathbb{D}$ of the synthetic table $\mathcal{T}$.*

Definition 2. *A* projection block *is a subset of points from $\mathbb{D}^{\mathbb{A}}$, where $\mathbb{D}^{\mathbb{A}}$ represents the data subspace of the synthetic table $\mathcal{T}$ spanned by a given PAS $\mathbb{A}$.*

Table 2: Notations

(a) **Input Related**

Symbol	Meaning
S	Table Schema
f	Filter predicate
$\mathbb{A}$	A PAS
l	Output row cardinality of a filtered table
k	Output row cardinality after projecting on a PAS
c	A PIC $\langle f, \mathbb{A}, l, k \rangle$
$\mathbb{W}$	Input PICs workload
$\mathbb{C}$	A compatible PICs workload

(b) **Output Table Related**

Symbol	Meaning
$\mathcal{T}$	Output Table
$s(\mathcal{T})$	Summary of $\mathcal{T}$
$\mathbb{U}$	attribute-set in $\mathcal{T}$
$\mathbb{D}$	Data space of $\mathcal{T}$
$\mathbb{D}^{\mathbb{A}}$	Data subspace spanned by $\mathbb{A}$

(c) **Block Related**

Symbol	Meaning
b	An FB
$\mathbb{R}$	Set of all RBs
r	An RB
$\overline{r}$	PRB wrt r and some PAS
$\overline{\mathbb{R}}^{\mathbb{A}}$	Set of PRBs for $\mathbb{A}$
p	A CPB
$\mathbb{P}^{\mathbb{A}}$	Set of CPBs for $\mathbb{A}$
x_r	variable for $\|r\|$
y_p	variable for $\|p\|$

(d) **Relation Related**

Symbol	Meaning
M	A relation btw $\mathbb{C}$, $\mathbb{R}$ (Definition 4)
$L^{\mathbb{A}}$	A relation btw $\mathbb{P}^{\mathbb{A}}$, $\overline{\mathbb{R}}^{\mathbb{A}}$ (Definition 6)

4.1 Region Partitioning

To model the filter predicates associated with $\mathbb{W}$, the data space $\mathbb{D}$ is logically partitioned into a set of blocks. Each block satisfies the condition that every data point in it satisfies the same subset of filter predicates.

The row cardinality of each block is represented using a variable in the LP. The resultant system is usually highly under-determined and therefore, to reduce the complexity of solving it, we leverage the *region partitioning* technique from [22], which partitions the data space into the minimum number of blocks.

Here, for a tuple $t \in \mathbb{D}$, and a PIC $c \in \mathbb{W}$, let $c(t)$ denote the indicator, set to 1 if t satisfies the filter predicate associated with c, 0 otherwise. Now, a pair of tuples t_1 and t_2 are said to be related by $R^{\mathbb{W}}$, if $c(t_1) = c(t_2)$, for all $c \in \mathbb{W}$. $R^{\mathbb{W}}$ is an equivalence relation, and the region partitioning algorithm returns the quotient set of $\mathbb{D}$ by $R^{\mathbb{W}}$. That is, the data points from the same equivalence class (wrt $R^{\mathbb{W}}$) form a block. Each resultant block is referred to as a *filter-block* (FB). The algorithm outputs the domain of each FB, which forms its logical condition. The domain of an FB b is denoted as $D(b)$.

To make the above concrete, consider the three filter predicates, f_1, f_2, f_3 on PURCHASES. For simplicity, Figure 1 shows only the 2D data space comprising the Qty and Amt attributes since no conditions exist on the other attributes. In this figure, the filter predicates are represented using regions delineated with colored solid-line boundaries. When region partitioning is applied on this scenario, it produces the four disjoint FBs: b_1, b_2, b_3, b_4, whose domains are depicted with dashed-line boundaries.

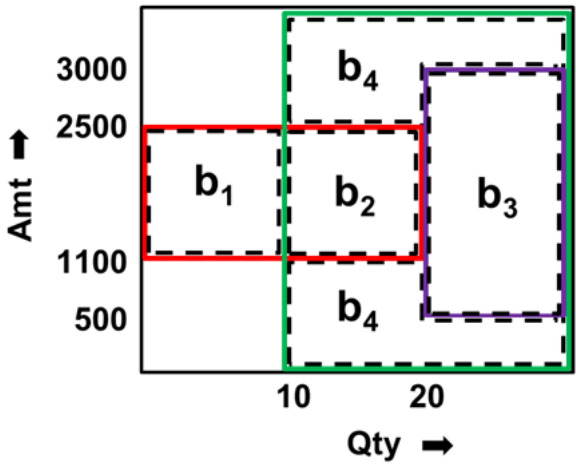


Figure 1: Region Partitioning

4.2 Isolating Projections

To circumvent inter-projection subspace dependencies, we first "isolate" the projections. Specifically, the following set of steps are taken in this process.

A *symmetric refinement* strategy is adopted that refines an FB into a set of disjoint *refined-blocks* (RBs) such that each resultant RB exhibits translation symmetry along each applicable projection subspace. That is, for each domain point of an RB r along a particular PAS, the projection of r along the remaining attributes is identical.

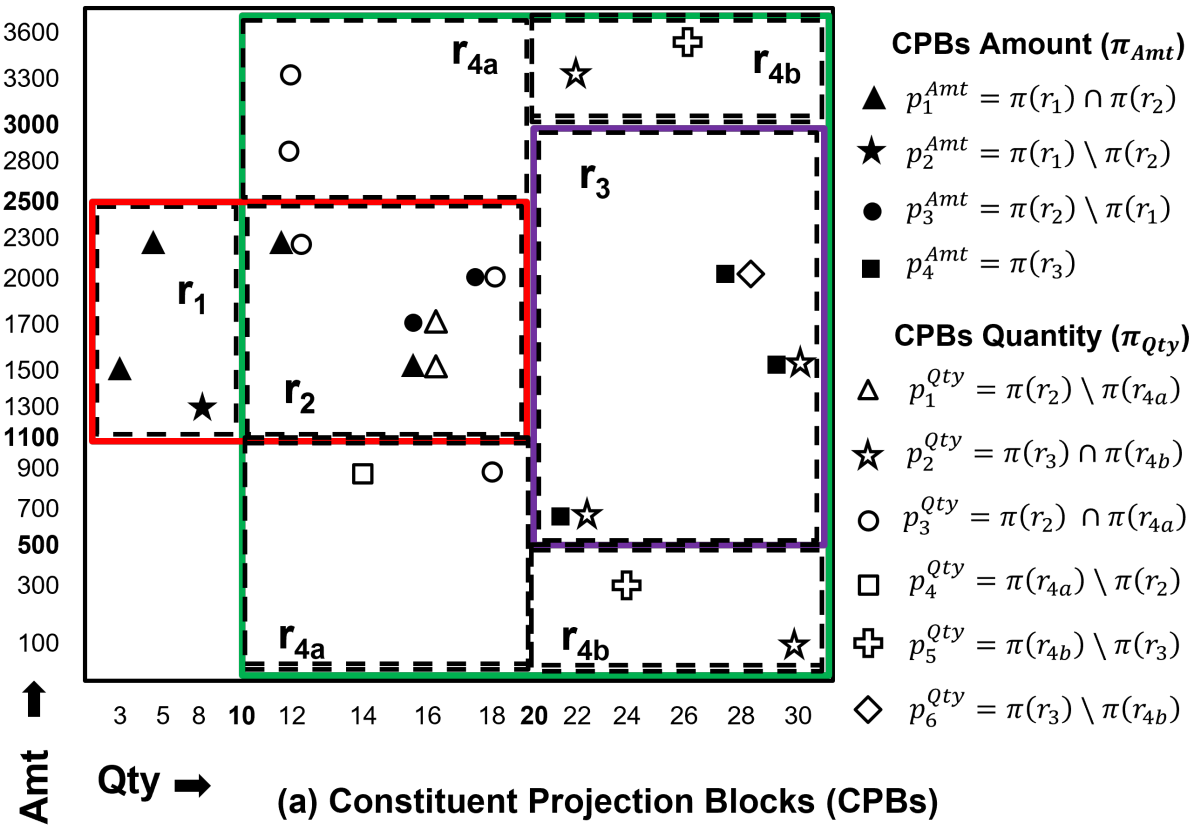
For instance, consider FB b_4 in Figure 1. Clearly, it is asymmetric along the PAS Qty – specifically, compare the spatial layout in the range $10 \leq Qty < 20$ with that in $Qty \geq 20$. After refinement, this block breaks into r_{4a} and r_{4b} as shown in Figure 2(a) – it is easy to see that r_{4a} and r_{4b} are symmetric. (The other FBs (b_1, b_2, b_3) happen to be already symmetric, and are shown as r_1, r_2 and r_3, respectively, in Figure 2(a)). This refinement allows for the values along different projection subspaces to be generated independently. That is, $D(r) = D(\pi_{Amt}(r)) \times D(\pi_{Qty}(r))$, for each RB r.

The above refinement, however, does not scale when the projections applied on an FB are along partially overlapping PASs, i.e. when different PASs share some attribute(s). Therefore, to eliminate such situations, we resort to *decomposing* the workload into non-overlapping sub-workloads using a *vertex coloring*-based strategy. As a consequence, for each such sub-workload, a separate summary is produced at the conclusion of the LP solution process. From a practical perspective, the multiplicity of summaries does not impose a substantive overhead since each summary is very small. However, to maximize the number of constraints that can share a common database, the number of sub-workloads required to eliminate all conflicts is minimized.

4.3 Projection Subspace Division

To deal with intra-projection subspace dependencies, the domain of each PAS is logically divided into a set of projection blocks, called *constituent-projection-blocks* (CPBs). This construction ensures that each projection cardinality is expressible as a *summation* over the cardinalities of these CPBs. Further, we ensure that the minimum number of CPBs is produced, aiding in efficient LP formulations.

For our example scenario, PiGen divides the data subspace associated with the Amt dimension into 4 CPBs: $p_1^{Amt}, p_2^{Amt}, p_3^{Amt}, p_4^{Amt}$, and the Qty dimension subspace into 6 CPBs: $p_1^{Qty}, p_2^{Qty}, ..., p_6^{Qty}$, as shown in Figure 2(a). Each CPB has a semantic meaning associated with it. For example, p_1^{Amt}



(a) Constituent Projection Blocks (CPBs)

Amt	1500	2300	1300	2300	2000	1700	1500	3300	2800
Qty	3	5	8	12	18	16	16	12	12

Amt	900	900	3300	3600	300	100	700	1500	2000
Qty	14	18	22	26	24	30	22	30	28

(b) Sample Purchases Table (Distinct Rows)

Figure 2: Symmetric Refinement and PSD

semantically represents the Amt values present in both r_1 and r_2. Further, the CPBs need not be mutually disjoint, as in the case of p_3^{Amt} and p_4^{Amt}. Finally, Figure 2(a) also shows the unique tuples enumerated by the sample output table shown in Figure 2(b), and the CPB (s) to which each of these tuples belongs.

4.4 Constraints Formulation

The LP solving procedure is constructed using variables representing the row cardinalities of RBs and CPBs. For instance, if x_i represents the cardinality of RB r_i, and y_j^{Amt} and y_k^{Qty} represent the cardinalities of CPBs p_j^{Amt} and p_k^{Qty}, respectively, then PICs are expressed by linear equations as follows:

$$
\begin{aligned}
c_1: &\quad x_1 + x_2 = 500, \quad y_1^{Amt} + y_2^{Amt} + y_3^{Amt} = 5 \\
c_2: &\quad x_3 = 1000, \quad y_4^{Amt} = 3 \\
c_3: &\quad x_2 + x_3 + x_{4a} + x_{4b} = 3000, \\
&\quad y_1^{Qty} + y_2^{Qty} + y_3^{Qty} + y_4^{Qty} + y_5^{Qty} + y_6^{Qty} = 9
\end{aligned}
$$

Finally, additional sanity constraints are added to the LP to ensure data constructibility. For example, the distinct row-cardinality of the projection of an RB is upper-bounded by the RB's native cardinality.

A sample solution to the above LP is shown below:

$$
\begin{aligned}
&x_1 = 500, \quad x_2 = 0, \quad x_3 = 1000, \quad x_{4a} = 0, \quad x_{4b} = 2000 \\
&y_1^{Amt} = 0, \quad y_2^{Amt} = 5, \quad y_3^{Amt} = 0, \quad y_4^{Amt} = 3, \quad y_1^{Qty} = 0 \\
&y_2^{Qty} = 5, \quad y_3^{Qty} = 0, \quad y_4^{Qty} = 0, \quad y_5^{Qty} = 0, \quad y_6^{Qty} = 4
\end{aligned}
$$

4.5 Enriched Database Summary

To construct the final summary, the domain of each PAS is divided into a set of intervals and then the CPBs are assigned these intervals. A sample summary for the PURCHASES table with respect to the aforementioned LP solution is shown in Figure 3, after incorporating an additional attribute $Year$ to illustrate a multi-dimensional projection.

Each segment of the summary corresponds to a populated RB. Specifically, the figure shows the tabulation for the r_1, r_3 and r_{4b} RBs. Each tabulation comprises of a column for each PAS acting on the RB, and an additional last column indicating the total number of tuples present in the RB. In each PAS column, the information for generating data of the associated projection subspace is present. Specifically, we maintain the intervals in the projection subspace along with their distinct counts. As a case in point, the first tabulation, corresponding to r_1, is interpreted as "generate 500 tuples, such that there are 5 distinct values of Amt in the interval `[1100,2500)`, and 20 distinct value pairs of $\{Qty, Year\}$ of which 12 are from the 2D interval `[1,10), [1990,2000)`, and the remaining 8 from the 2D interval `[1,10), [2010,2020)`."

r_1

Amt	Qty, Year	#Tuples
[1100, 2500): 5	(Q) [1, 10), (Y) [1990, 2000): 12 (Q) [1, 10), (Y) [2010, 2020): 08	500

r_3

Amt	Qty	Year	#Tuples
[500, 3000): 3	[20, 25): 5 [25, 40): 4	[1990,2020)	1000

r_{4b}

Qty	Amt, Year	#Tuples
[20, 25): 5	(A) [1,500) U [3000,3600), (Y) [1990, 2020): 6	2000

Figure 3: PiGen Table Summary

For attributes that do not feature in any projection subspace, no associated distinct cardinality is maintained – an example is $Year$ in r_3. Lastly, the primary-key column (PID in the example) is

omitted from the summary and is assumed to be a sequence of distinct natural numbers during on-demand tuple generation. Further, note that the intervals present in a summary may not be continuous. For instance, the $\{Amt, Year\}$ points in r_{4b} are sourced from two separate intervals: `[1,1500)` and `[3000,3600)` for Amt column. From a generation perspective, however, data can be constructed from either or both the sub-intervals.Finally, we observe that this summary is significantly different from that produced by Hydra [22]. The key difference lies in that Hydra neither maintains intervals nor distinct value counts. Also, since it only maintains FBs, which being inherently disjoint have no dependency among them. In Pigen, we need to handle the dependencies enforced due to common CPBs being shared between RBs.

This summary is used for deterministic tuple instantiation method, which ensures that despite the tuples being generated independently for various CPBs across all RBs, the row-cardinalities match the requirement.

In the following sections, we present the internal details of each of the aforementioned concepts.

5 Isolating Projections

To facilitate independent processing of projection sub-spaces, we refine the FBs so that the resultant blocks become symmetric. The symmetry is formally defined as follows:

Definition 3. *A block r in the data space of a $\mathbb{U}$-dimensional table $\mathcal{T}$ is symmetric along a PAS $\mathbb{A}$ iff*

$$D(r) = D(\pi_{\mathbb{A}}(r)) \times D(\pi_{\mathbb{U}\setminus\mathbb{A}}(r))$$

where $D(.)$ returns the domain of the input block.

Likewise r is symmetric along PASs $\mathbb{A}_1, \mathbb{A}_2, ..., \mathbb{A}_\alpha$ iff

$$D(r) = D(\pi_{\mathbb{A}_1}(r)) \times D(\pi_{\mathbb{A}_2}(r)) \times ... \times D(\pi_{\mathbb{A}_\alpha}(r)) \times D(\pi_{\mathbb{U}\setminus(\mathbb{A}_1\cup\mathbb{A}_2\cup...\cup\mathbb{A}_\alpha)}(r))$$

The Cartesian product implies that for a symmetric block, the data can be *independently* generated for each PAS considered. Therefore, Symmetric Refinement module refines each FB into a set of symmetric blocks along the PASs acting on it. Hence, post-refinement, the different projection spaces can be processed independently. The refinement algorithm is discussed in Section 5.1.

Impact of Overlapping Projection Subspaces. When partially overlapping PASs, say $\mathbb{A}_1$ and $\mathbb{A}_2$, are applied on an FB b, symmetric refinement becomes computationally challenging. This is because $\mathbb{A}_1, \mathbb{A}_2$ have to be made conditionally independent for b, requiring refinement such that each resulting block is symmetric along $\mathbb{A}_1$ and $\mathbb{A}_2$ for *each* domain point in $D(\mathbb{A}_1 \cap \mathbb{A}_2)$. This is easily done by enumeration for small cardinality domains, but does not scale in general. Hence, in PiGen we bypass such overlapping projection operations by ensuring, as described in Section 5.2, that the input workload is initially itself decomposed such that there are no projection subspace overlaps in the resulting sub-workloads.

5.1 Symmetric Refinement

The refinement for each FB is done independently. Given an FB b and its associated PASs, this module refines b into a group of RBs, such that each RB is symmetric along the input PASs.

Let us first understand the refinement procedure for an FB along a single PAS. Here, given a block b, and a PAS $\mathbb{A}$, the refinement of b along $\mathbb{A}$ is carried out as follows:

1. Let $\mathbb{I}$ be the subset of all interval-combinations in $D(\mathbb{A})$ that are present in b. The interval boundaries along an attribute are computed using the constants that appear in the filter predicates of the input PICs. For some interval-combination $\mathcal{I} \in \mathbb{I}$, let $b_{\mathcal{I}}$ denote the part of b whose projection along $\mathbb{A}$ is $\mathcal{I}$.

2. For each interval combination $\mathcal{I} \in \mathbb{I}$, the projection of $b_{\mathcal{I}}$ along $\mathbb{U} \setminus \mathbb{A}$ is computed, and denoted as $\pi(b_{\mathcal{I}})$.

3. A hashmap H is created with keys as $\pi(b_{\mathcal{I}})$ and value as $\mathcal{I}$. Hence, the parts of b where the projection of b along $\mathbb{U} \setminus \mathbb{A}$ do not alter with changing values of $\mathbb{A}$ are clubbed together into a single hash entry. This construction provides independence between $\mathbb{A}$ and the $\mathbb{U} \setminus \mathbb{A}$ subspaces.

4. Each entry e in H corresponds to an RB, constructed by taking the region stored as key in e for the $\mathbb{U} \setminus \mathbb{A}$ attribute-set, and a union of regions stored as value in e for the $\mathbb{A}$ attribute-set.

Interestingly, the above refinement strategy also ensures that the number of resultant blocks is kept to a minimum. Let the domain of b along $\mathbb{A}$ be denoted as $D^{\mathbb{A}}(b)$. Further, let $S_b^{\mathbb{A}}$ be a relation associated with the points in $D^{\mathbb{A}}(b)$. For a pair of points $t_1, t_2 \in D^{\mathbb{A}}(b)$, we say $t_1 S_b^{\mathbb{A}} t_2$ iff the projection t_1 and t_2 along the rest of the attributes i.e. $\mathbb{U} \setminus \mathbb{A}$ is identical. It is easy to verify that $S_b^{\mathbb{A}}$ forms an *equivalence relation*. For an equivalence relation, the *quotient set* of the relation gives the minimum partition.

Lemma 1. *The Symmetric Refinement algorithm returns the quotient set of $D^{\mathbb{A}}(b)$ by $S_b^{\mathbb{A}}$.*

The proof follows from the fact that Symmetric Refinement algorithm uses a hashmap, which enables grouping of points in $D^{\mathbb{A}}(b)$ together such that their projection on $\mathbb{U} \setminus \mathbb{A}$ are identical. Hence, for a PAS, the symmetric refinement algorithm produces the quotient set of $S_b^{\mathbb{A}}$, and hence returns the refinement with minimum number of blocks.

Extension to Multiple PAS

We now move on to the multiple PAS scenario. Let there be α PASs $(\mathbb{A}_1, \mathbb{A}_2, ..., \mathbb{A}_\alpha)$ applicable on b across all PICs. This implies that there are $\alpha + 1$ projection subspaces – $\pi_{\mathbb{A}_1}(b), \pi_{\mathbb{A}_2}(b), ..., \pi_{\mathbb{A}_\alpha}(b)$, and $\pi_{\mathbb{U} \setminus (\mathbb{A}_1 \cup \mathbb{A}_2 \cup ... \cup \mathbb{A}_\alpha)}(b)$. It is easy to see that the block becomes symmetric when refined along any α of these $\alpha + 1$ subspaces.

The refinement is done iteratively, where the output of refinement along one subspace is fed into the next in the sequence. Since any sequence among the chosen α subspaces results in a symmetric block, there are a total of $\binom{\alpha+1}{\alpha}\alpha!$ ways to do the refinement. The specific choice that we make from this large set of options is important because it has an impact on the number of variables in the LP, and hence the computational complexity and scalability of the solution procedure. In particular, the number of CPBs created depends on the geometry of the RBs, and usually more overlaps of RBs along a PAS results in more CPBs. More precisely, if we refine a block along a subspace, the overlaps in that space remain unaffected, but the overlaps along the *remaining subspaces* may increase. Therefore, to minimize this collateral impact, we adopt the following greedy heuristic in PiGen: The subspace having the maximum FB overlaps with b is chosen as the next subspace to be refined in the iterative sequence.

Mapping RBs to PICs

The set of RBs, denoted by $\mathbb{R}$, are connected with the set of PICs using the following relation:

Definition 4. *An RB $r \in \mathbb{R}$ is related by relation M to a PIC c containing filter predicate f, iff $D(r)$ satisfies f. That is,*

$$rMc \Leftrightarrow t \text{ satisfies } f, \forall t \in D(r)$$

For a PIC c, the associated filter predicate's output cardinality l can be expressed as the union of a group of RBs related to c by M, as follows:

$$|\bigcup_{r:rMc} r| = \sum_{r:rMc} |r| = l$$

Since all the RBs are mutually disjoint, the union could be replaced with summation in the above equation.

5.2 Workload Decomposition

As discussed previously, symmetric refinement is performed when distinct PASs applicable on an FB are non-overlapping. This holds true when, for each domain point t, the distinct PASs across various PICs that are applicable on t, are mutually disjoint. For any given collection of sets (PASs) to be mutually disjoint, it is equivalent to say that they are *pairwise disjoint*. This leads us to defining the concept of an *intersecting pair* of PICs.

Definition 5. *A pair of PICs ($c_1 : \langle f_1, \mathbb{A}_1, l_1, k_1 \rangle, \quad c_2 : \langle f_2, \mathbb{A}_2, l_2, k_2 \rangle$) intersect iff:*

- *their PASs partially intersect, i.e.,*

$$\mathbb{A}_1 \cap \mathbb{A}_2 \neq \emptyset, \mathbb{A}_1 \neq \mathbb{A}_2, \text{ and}$$

- *f_1 and f_2 overlap, i.e., there exists a point t in the domain space of $\mathcal{T}$ such that t satisfies f_1 and f_2.*

For example, consider the following pair of constraints, c_4 and c_5, on the PURCHASES table:

$$c_4 : \langle Amt \leq 2500 \wedge Year \geq 1990, (Qty, Year), 500, 20 \rangle$$
$$c_5 : \langle Qty \geq 20 \wedge Year \leq 2020, (Amt, Year), 2000, 6 \rangle$$

We see that the filters in the two constraints overlap, and the corresponding PASs also partially intersect.

In the Workload Decomposition module, the input workload is split such that there are no intersecting pairs of PICs in the resulting sub-workloads. We refer to a workload with no intersecting pairs as a *compatible* workload, and denote it using $\mathbb{C}$.

Given an original workload $\mathbb{W}$, the set of intersecting pairs IP is computed first. Subsequently, we construct compatible sub-workloads $\mathbb{C}_1, \mathbb{C}_2, .., \mathbb{C}_n$ that cover the entire workload. Additionally, we aim towards minimizing n, i.e. the number of sub-workloads. This minimization is desirable to facilitate common platform for workload performance evaluation. Since the minimization is NP-complete (reduction from vertex coloring), we adopt a heuristic based on greedy vertex coloring. The algorithm iterates over the PICs, and in each iteration, the PIC c with minimum intersections in IP is picked and assigned to a compatible sub-workload $\mathbb{C}_i$. If multiple compatible options are available, an assignment that minimizes the skew in the sub-workload sizes is made. On the other hand, if no such assignment is possible, a new sub-workload is constructed, and initialized with c.

In the worst case, the above algorithm can create one sub-workload per query. However, it is our experience that in practice, a small number of sub-workloads is usually sufficient. Further, we hasten to add that even if the worst case materializes, the overheads incurred would be marginal as only a single small summarized table is stored per sub-workload.

6 Projection Subspace Division

We now turn our attention to handling intra-projection subspace dependencies. The projection output cardinality with respect to a PIC c can be expressed using the relation M as follows:

$$|\bigcup_{r:rMc} \pi_{\mathbb{A}}(r)| = k$$

We use the shorthand $\overline{r}$ to represent the projection of an RB r on $\mathbb{A}$, i.e. $\overline{r} = \pi_{\mathbb{A}}(r)$, and this projection block is referred to as a *projected-refined-block* (PRB). The set of all PRBs for a PAS $\mathbb{A}$ is shown as $\overline{\mathbb{R}}^{\mathbb{A}}$. Further, for brevity, we overload the same relation M to establish an association between PRB $\overline{r}$ and a constraint c. That is, $\overline{r}Mc \Leftrightarrow rMc$. Hence we can rewrite the above equation as:

$$|\bigcup_{\overline{r}:\overline{r}Mc} \overline{r}| = k$$

The union here cannot be replaced with summation because unlike RBs, the PRBs need not be disjoint. Therefore, to be able to express the constraint as a linear equation, the projection subspace $\mathbb{D}^{\mathbb{A}}$ needs to be divided into a set of CPBs. The set of CPBs corresponding to a PAS $\mathbb{A}$ is denoted using $\mathbb{P}^{\mathbb{A}}$. Each element $p \in \mathbb{P}^{\mathbb{A}}$ logically represents a subset of $\mathbb{D}^{\mathbb{A}}$. Further, a relation $L^{\mathbb{A}}$ is provided that connects the elements of $\mathbb{P}^{\mathbb{A}}$ with elements of $\overline{\mathbb{R}}^{\mathbb{A}}$. We first define the notion of what constitutes a valid division, and then go on to presenting an algorithm that provides the (unique) optimal division.

6.1 Valid Division

A valid division is defined as follows:

Definition 6. *Given $\mathbb{C}, \overline{\mathbb{R}}^{\mathbb{A}}$ and M, a division $(\mathbb{P}^{\mathbb{A}}, L^{\mathbb{A}})$, with respect to a projection data subspace $\mathbb{D}^{\mathbb{A}}$, is called a* valid division *if it satisfies the following two requirements:*

Condition 1. *Each PRB $\overline{r} \in \overline{\mathbb{R}}^{\mathbb{A}}$ is expressible as a union of a group of elements from $\mathbb{P}^{\mathbb{A}}$, determined by relation $L^{\mathbb{A}}$, as shown below:*

$$\overline{r} = \bigcup_{p:pL^{\mathbb{A}}\overline{r}} p, \quad \forall\, \overline{r} \in \overline{\mathbb{R}}^{\mathbb{A}} \tag{2}$$

Condition 2. *All elements in $\mathbb{P}^{\mathbb{A}}$ that are related to a constraint $c \in \mathbb{C}$ through the composite relation*

$$M \circ L^{\mathbb{A}} = \{(p, c) | \exists \overline{r} \in \overline{\mathbb{R}}^{\mathbb{A}} : \overline{r}Mc \wedge pL^{\mathbb{A}}\overline{r}\}$$

that is, all elements of the set $\{p : (p, c) \in M \circ L^{\mathbb{A}}\}$, should be mutually disjoint *for all $c \in \mathbb{C}$.*

Condition 1 is needed to associate an PRB with its constituent CPBs. This is required during data generation in order to populate appropriate RBs based on the cardinalities of CPBs obtained from the LP solution. Condition 2 enforces that each constraint is comprised of disjoint constituent CPBs, thereby enabling expression of constraints as linear equations.

For ease of presentation, we drop $\mathbb{A}$, which can be assumed implicitly, from the superscript in the rest of this section.

We now give a bound on the number of CPBs required. Each element p of $\mathbb{P}$ maps to a collection of sets from $\overline{\mathbb{R}}$ using relation L. If there are m elements in $\overline{\mathbb{R}}$, then p has one of the total $2^m - 1$ possible mappings.

Lemma 2. *If a pair of CPBs in $\mathbb{P}$, p_1 and p_2, map to identical sets in $\overline{\mathbb{R}}$, they can be combined into a single element $p_1 \cup p_2$, without violating either condition.*

Proof. We are given that p_1 and $p_2 \in P$ are such that $p_1 L \overline{r} \Leftrightarrow p_2 L \overline{r}$ for $s \in S$. We need to prove that replacing p_1 and p_2 with $p_{1,2} = p_1 \cup p_2$ in P does not violate any of the two conditions.

- **Condition 1:** It is required that each $\overline{r} \in \overline{\mathbb{R}}$ is expressible as union of related elements of P through L.

  If $(p_1, \overline{r}) \notin L$, then $(p_2, \overline{r}) \notin L$ (and vice versa). Hence, the expression for $\overline{r}$ remains unaltered.

  If $(p_1, \overline{r}) \in L$, then $(p_2, \overline{r}) \in L$ (and vice versa). Let $\rho = \{p \in P \setminus \{p_1, p_2\} : pL\overline{r}\}$. Then, $\overline{r} = p_1 \cup p_2 \bigcup_{p \in \rho} p$. After replacing p_1 and p_2 with $p_{1,2}$, the expression would become $\overline{r} = p_{1,2} \bigcup_{p \in \rho} p$.

- **Condition 2:** Let c be any $c \in \mathbb{C}$ such that $(p_1, c) \in M \circ L^{\mathbb{A}}$ (and $(p_2, c) \in M \circ L^{\mathbb{A}}$). It is easy to see that (from Condition 2) p_1 will be disjoint with all the other elements of P that are related to c through $M \circ L^{\mathbb{A}}$. That is,

  $$p_1 \cap p' = \emptyset, \forall p' \in P \setminus \{p_1\} : (p, c) \in M \circ L^{\mathbb{A}}$$

  Likewise, p_2 will also be disjoint with all the other elements of P that are related to c. Therefore, on replacing p_1 and p_2 with their union $p_{1,2}$, $p_{1,2}$ will continue to remain disjoint with all the other elements of P that are related to c.

□

From Lemma 2, we know that at most one CPB is needed for each mapping. Therefore, $2^m - 1$ is the upper bound on the number of CPBs required for an $\overline{\mathbb{R}}$ of length m.

From this observation, let us first look at an extreme construction of $(\mathbb{P}, L)$ with $|\mathbb{P}| = 2^m - 1$, where there is a single element $p \in \mathbb{P}$ for each possible mapping.

Powerset Division

Consider a set $\mathbb{P}$ having $2^m - 1$ elements with a mapping relation L such that each element p in $\mathbb{P}$ maps to one of the non-empty subsets of $\overline{\mathbb{R}}$. Further, p's content is defined as follows:

$$p = \bigcap_{\overline{r}:(p,\overline{r}) \in L} \overline{r} \setminus \bigcup_{\overline{r}':(p,\overline{r}') \notin L} \overline{r}' \tag{3}$$

That is, p includes the data points that are present in all the PRBs that are related to p and absent from each of the remaining PRBs.

$\mathbb{P}$ satisfies the two conditions for valid division. This is because:

1. Each element $\overline{r} \in \overline{\mathbb{R}}$ can be expressed as a union of a subset of elements in $\mathbb{P}$, as shown below:

   $$\overline{r} = \bigcup_{p:pL\overline{r}} p$$

2. All the elements in $\mathbb{P}$ are mutually disjoint.

Consider the projection subspace of Amt in our running example. $\overline{\mathbb{R}}^{Amt} = \{\overline{r}_1, \overline{r}_2, \overline{r}_3\}$. Since there are three PRBs, seven possible mappings exist. Figure 4 illustrates these seven mappings. Powerset Division (`Pow-PSD`) creates seven CPBs, one CPB corresponding to each mapping. Hence, the seven resulting CPBs in $\mathbb{P}^{Amt}$ are as follows:

$$\overline{r}_1 \setminus (\overline{r}_2 \cup \overline{r}_3), \quad (\overline{r}_1 \cap \overline{r}_2) \setminus \overline{r}_3, \quad (\overline{r}_1 \cap \overline{r}_3) \setminus \overline{r}_2, \quad \overline{r}_1 \cap \overline{r}_2 \cap \overline{r}_3,$$
$$\overline{r}_2 \setminus (\overline{r}_1 \cup \overline{r}_3), \quad \overline{r}_2 \cap \overline{r}_3 \setminus \overline{r}_1, \quad \overline{r}_3 \setminus (\overline{r}_1 \cup \overline{r}_2)$$

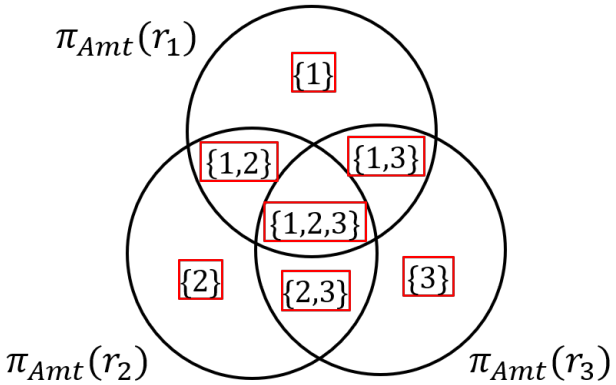


Figure 4: Partitioning in Projected Space

6.2 Optimal Division

The number of CPBs in $\mathbb{P}$ determine the number of variables in the LP. Therefore, reducing the size of $\mathbb{P}$ helps in reducing the complexity of LP, thereby providing workload scalability and computational efficiency. Hence, we define an *optimal division* as a valid division that has the minimum number of CPBs.

Definition 7. *A valid division $(\mathbb{P}, L)$ is called an optimal division iff there does not exist any other valid division $(\mathbb{P}', L')$ such that $|\mathbb{P}'| < |\mathbb{P}|$. We represent the optimal division by $(\mathbb{P}^*, L^*)$.*

We now shift our focus towards identifying the optimal division. As a first step, let us define some general characteristics of the set $\mathbb{P}$ and the corresponding relation L.

If a CPB p is related to a PRB $\overline{r}$, then p is a subset of $\overline{r}$. That is,

$$pL\overline{r} \implies p \subseteq \overline{r} \tag{4}$$

Alternatively, a second possibility is of disjointedness. Let p_1, p_2 be such that $(p_1, c), (p_2, c) \in M \circ L$ for some $c \in \mathbb{C}$. Further, let $\overline{\mathbb{R}}(p_1), \overline{\mathbb{R}}(p_2)$ represent the set of PRBs that are related to p_1 and p_2, respectively, through L. Using Condition 2 and Equation 4, we can say that

$$\begin{aligned} p_1 \cap \overline{r} &= \emptyset, \quad \text{where } \overline{r} \in \overline{\mathbb{R}}(p_2) \setminus \overline{\mathbb{R}}(p_1) \\ p_2 \cap \overline{r} &= \emptyset, \quad \text{where } \overline{r} \in \overline{\mathbb{R}}(p_1) \setminus \overline{\mathbb{R}}(p_2) \end{aligned} \tag{5}$$

Therefore, CPBs may have a disjoint relation with a PRB.

Finally, a third possibility is when a CPB does not have a relation with a PRB, which allows room for constructing CPBs that overlap.

Our division algorithm distinguishes these three possibilities using a vector v_p corresponding to each CPB p in $\mathbb{P}$. The vector is of length m, where each element is associated with an element of $\overline{\mathbb{R}}$. Further, the element associated with $\overline{r} \in \overline{\mathbb{R}}$ is denoted by $v_p(\overline{r})$. Specifically, element $v_p(\overline{r})$ is set to 1 iff $pL\overline{r}$. Using Equation 5, the elements in v_p corresponding to the sets $\overline{\mathbb{R}}(p') \setminus \overline{\mathbb{R}}(p)$ for all p' such that $(p, c), (p', c) \in M \circ L$ for some $c \in \mathbb{C}$, are represented as 0, denoting the absence of values from these sets. The remaining elements of v_p are set as '$\times$' denoting a *don't care* state, i.e. p and $\overline{r}$ may or may not have an intersection. Finally, using the vector v_p, p can be expressed as:

$$p = \bigcap_{\overline{r}:v_p(\overline{r})=1} \overline{r} \setminus \bigcup_{\overline{r}':v_p(\overline{r}')=0} \overline{r}' \tag{6}$$

Let $\mathbb{V}$ represent the set of all possible vectors. Further, let $\mathbb{Q}$ denote the collection of CPBs, where there is a projection-block q associated with each vector $v \in \mathbb{V}$. Therefore, $\mathbb{P}^* \subseteq \mathbb{Q}$. Let the subset of $\mathbb{V}$ corresponding to the elements in $\mathbb{P}^*$ be denoted as $\mathbb{V}^*$. Each position in vector v can have one of the three possibilities among $0, 1, \times$, and at least one position needs to mandatorily be 1. Therefore, $\mathbb{Q}$ comprises $3^m - 2^m$ elements. Note that $\mathbb{Q}$ forms a *partial-order* with respect to the subset relation, and can therefore be represented by a Hasse Diagram. As an exemplar, the Hasse Diagram for an $m = 3$ case is shown in Figure 5 (for simplicity, the elements of $\mathbb{V}$ are shown instead of $\mathbb{Q}$).

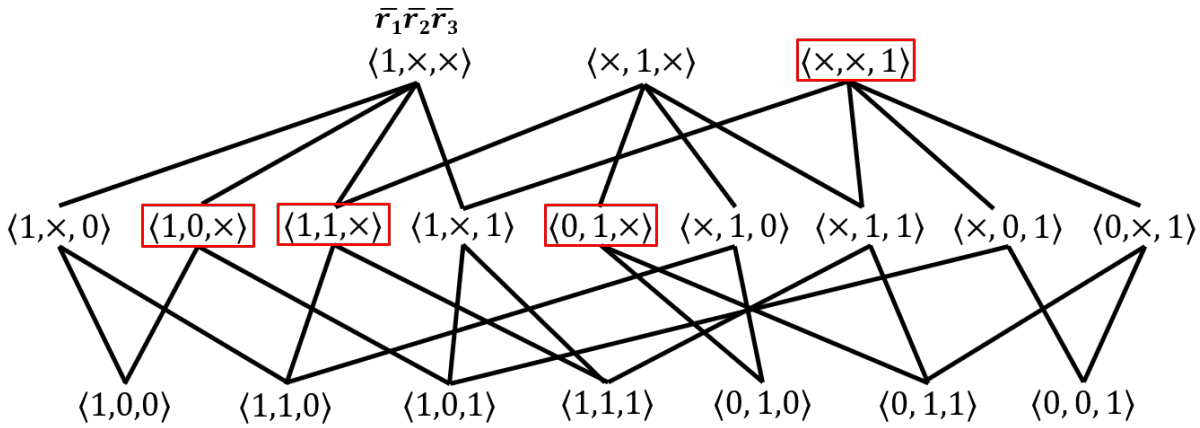


Figure 5: Hasse Diagram

We hasten to add that to compute $\mathbb{P}^*$, it is not necessary to iterate on all the elements of $\mathbb{Q}$. Instead, the division begins with the top nodes of the Hasse diagram and recursively splits a block only if required to satisfy the two conditions.

The detailed mechanics of the division algorithm, called `Opt-PSD`, with pseudocode as shown in Algorithm 1, are described next.

6.3 Opt-PSD Algorithm

We begin our computation of the projection subspace division by creating a *Division Graph* (DG). In this graph, a vertex is created corresponding to each element of $\overline{\mathbb{R}}$. Then, an edge is added between vertices corresponding to $\overline{r}_1$ and $\overline{r}_2$ if there exists a constraint c such that $\overline{r}_1 Mc$ and $\overline{r}_2 Mc$, (i.e. both the PRBs are related to a common constraint c), and the domains of $\overline{r}_1$ and $\overline{r}_2$ intersect. The resultant graph G is given as input to Algorithm 1, which returns the set of vectors $\mathbb{V}^*$ in the output. Leveraging

the vectors, the contents of the CPBs are computed using Equation 6. Then, the L^* relation is populated with the expression: $(p, \overline{r}) \in L^*, \quad \text{if} \quad v_p(\overline{r}) = 1, v_p \in \mathbb{V}^*$
The rest of the algorithm proceeds as follows:

- We iterate over the vertices of G. In the iteration for a PRB $\overline{r}$, a vector is initialized with '$\times$' for all the positions except that corresponding to $\overline{r}$, which is set to 1 (Line 3 of Algorithm 1). These initial vectors represent the top nodes of the Hasse Diagram. They are recursively further split in the while loop (Line 5), using a running list of vectors called $toBeSplit$.
- In each iteration of the while loop, an element v from $toBeSplit$ is popped and split using a $pivot$ vertex; the resultant elements are re-inserted in the list. A pivot PRB is distinguished as one which is included in v and co-occurs in a constraint c with another PRB (target) whose current assignment in the vector is $\times$. To compute the $pivot$ vertex in G, the $getPivot$ function is used, which selects the pivot based on the following conditions: (a) $v(pivot) = 1$, and (b) There exists a PRB $\overline{r}$ such that there is an edge between the vertices corresponding to $pivot$ and $\overline{r}$. Further, the value for $\overline{r}$ in the vector v is $\times$.
- The collection of all PRBs that satisfy condition (b) is denoted as the $targets$ set corresponding to $pivot$, and is returned by the $getPivot$ function. Now, v is split using the $Split$ function, which computes a powerset enumeration of the vector positions corresponding to PRBs in $targets$. This function also ensures that no redundant elements are added in the result set.

Algorithm 1: Optimal Projection Subspace Division

Input: Division Graph G
Output: Optimal Vectors-set $\mathbb{V}^*$
$toBeSplit \leftarrow \emptyset$;
$visited \leftarrow \emptyset$;
for $\overline{r}$ *in* $\overline{\mathbb{R}}$ **do**
  $visited \leftarrow visited \cup \overline{r}$ $v_{init} \leftarrow \{\times\}^m, v_{init}(\overline{r}) \leftarrow 1$;
  $toBeSplit \leftarrow \{v_{init}\}$;
  while $toBeSplit \neq \emptyset$ **do**
    $v \leftarrow toBeSplit.pop()$;
    $pivot, targets \leftarrow getPivot(G, v)$;
    if $pivot$ *exists* **then**
      $toBeSplit \leftarrow toBeSplit \cup Split(v, pivot, targets, visited)$;
    else
      $\mathbb{V}^* \leftarrow \mathbb{V}^* \cup \{v\}$;
    end
  end
end
return $\mathbb{V}^*$;

The correctness of `Opt-PSD` algorithm follows from the following:

- it starts from the top nodes of the Hasse diagram and recursively refines them. Therefore, it continues to cover all the elements of $\overline{\mathbb{R}}$.

Function `Split` (v, $pivot$, $targets$, $visited$) **:**
  $splitSet \leftarrow \emptyset$;
  for $\overline{r} \in targets$ **do**
    if $\overline{r} \in visited$ **then**
      $v_r \leftarrow 0$;
      remove $\overline{r}$ from $targets$;
    end
  end
  if $targets = \emptyset$ **then**
    return v;
  end
  $powerset \leftarrow$ generate powerset enumeration of $targets$;
  for $s \in powerset$ **do**
    $new\_v \leftarrow v$;
    $new\_v_r \leftarrow 1, \forall \overline{r} \in s$;
    $new\_v_r \leftarrow 0, \forall \overline{r} \in targets \setminus s$;
    $splitSet \leftarrow splitSet \cup new\_v_r$;
  end
  return $splitSet$;

- the PRBs that are related to a common constraint are split by restricted powerset enumeration ensuring that they are mutually disjoint.

Hence, the algorithm does restricted enumeration depending on vertex's neighbours, or in other words it takes into account which PRBs co-appear in a constraint.

Example Division

Consider the projection subspace of Amt in Example 1. $\overline{\mathbb{R}}^{Amt} = \{\overline{r}_1, \overline{r}_2, \overline{r}_3\}$. Let us see how the CPBs for projection subspace of Amt are created by `Opt-PSD`. The input DG for the example is shown in Figure 6.

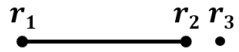


Figure 6: Example Division Graph

Initialization: $toBeSplit = \emptyset, \mathbb{V}^* = \emptyset$

Iteration 1: $\overline{r}_1$ is picked, $v_{init} = \langle 1 \times \times \rangle$ is added to $toBeSplit$, $toBeSplit = \{\langle 1 \times \times \rangle\}$. After popping, $v = \langle 1 \times \times \rangle$, $getPivot$ returns $pivot = 1, targets = \{2\}$ as vertex 1 is connected to vertex 2. The split function splits v by a restricted powerset enumeration on $targets$. $\{\langle 11\times \rangle, \langle 10\times \rangle\}$ is added to $toBeSplit$, $toBeSplit = \{\langle 11\times \rangle, \langle 10\times \rangle\}$. Both the elements in $toBeSplit$ are popped one by one and are added to $\mathbb{V}^*$ as they have no pivot. $toBeSplit = \emptyset, \mathbb{V}^* = \{\langle 11\times \rangle, \langle 10\times \rangle\}$.

Iteration 2: $\overline{r}_2$ is picked and the corresponding $v = \langle \times 1 \times \rangle$ is added to $toBeSplit$, $toBeSplit = \{\langle \times 1 \times \rangle\}$. After popping, $v = \langle \times 1 \times \rangle$ which return $pivot = 2, targets = \{1\}$ as vertex 2 is only

connected to vertex 1. On splitting, $\{\langle 01\times\rangle, \langle 11\times\rangle\}$ are added to $toBeSplit$. Both the elements are popped and $\langle 01\times\rangle$ is added to $\mathbb{V}^*$ as it does not have a pivot. $\langle 11\times\rangle$, being already present in $\mathbb{V}^*$, is not inserted again. $toBeSplit = \emptyset, \mathbb{V}^* = \{\langle 11\times\rangle, \langle 10\times\rangle, \langle 01\times\rangle\}$.

Iteration 3: $\overline{r}_3$ is picked with $\langle \times \times 1\rangle$ and added to $toBeSplit$. After popping, $v = \langle \times \times 1\rangle$, no pivot is found by $getPivot$ as vertex 3 is not connected to any other vertex. v is added to $\mathbb{V}^*$.

Finally, $\mathbb{V}^* = \{\langle 11\times\rangle, \langle 10\times\rangle, \langle 01\times\rangle, \langle \times \times 1\rangle\}$ (highlighted in Figure 5). Using Equation 6, it yielded in 4 CPBs for Amt, $\mathbb{P}^* = \{p_1, p_2, p_3, p_4\}$ (as discussed in Section 4) and $L^* = \{(p_1, \overline{r}_1), (p_1, \overline{r}_2), (p_2, \overline{r}_1), (p_3, \overline{r}_2), (p_4, \overline{r}_3)\}$.

The degree of the DG has a proportional impact on the number of CPBs constructed. To see this behaviour, the number of CPBs for `Opt-PSD` for a few general DGs are shown in Table 3.

Table 3: No. of CPBs in `Opt-PSD`

Division Graph	No. of CPBs
Empty Graph ($\overline{K_m}$)	m
Path Graph (P_m)	$\frac{1}{2}m(m+1)$
Cycle Graph (C_m)	$m^2 - m + 1$
Star ($K_{1,m-1}$)	$2^{m-1} + m - 1$
Complete Graph (K_m)	$2^m - 1$

6.4 Proof of Optimality

We now prove that `Opt-PSD` produces the optimal division. For a CPB $p \in \mathbb{P}$, consider the subset s of points:

$$s = \bigcap_{\overline{r}:v_p(\overline{r})=1} \overline{r} \setminus \bigcup_{\overline{r}':v_p(\overline{r}')=0,\times} \overline{r}'$$

Note that with this definition, $s \subseteq p$ and cannot overlap with any $p' \in \mathbb{P} \setminus \{p\}$. This restriction leads to the following lemma:

Lemma 3. *Given* $(\mathbb{P}, L)$ *returned by* `Opt-PSD`*,* $\forall p \in \mathbb{P}$*, there exists a point* $u \in p$ *such that* $u \notin p', \forall p' \in \mathbb{P} \setminus \{p\}$.

We use this observation to prove that `Opt-PSD` returns an optimal division, and further, that this optimal division is *unique*.

Lemma 4. `Opt-PSD` *returns the unique optimal division.*

Proof. We give a brief sketch of the proof here.

Let $(\mathbb{P}, L)$ be the division provided by `Opt-PSD`, and let there be another division $(\mathbb{P}', L')$ such that $|\mathbb{P}'| \leq \mathbb{P}$.

$$\implies \exists u \in p_1, v \in p_2 (\neq p_1) \text{ for some } p_1, p_2 \in \mathbb{P}, \text{ where } p_1 L \overline{r}_1, p_2 L \overline{r}_2,$$
$$\overline{r}_1, \overline{r}_2 \in \overline{\mathbb{R}}, \text{ such that } u, v \in p', p' L' \overline{r}_1, p' L' \overline{r}_2 \text{ for some } p' \in \mathbb{P}'.$$

Case (1) $\overline{r}_1 = \overline{r}_2 = \overline{r}$: Since $p_1 L \overline{r}$ and $p_2 L \overline{r}$,

$\implies \exists c \in C$ such that $\overline{r} M c, \overline{r}' M c,$ for some $\overline{r}' \in \overline{\mathbb{R}}$ and
$(p_1, \overline{r}') \in L, (p_2, \overline{r}') \notin L$ (wlog) (using Lemma 2)
$\implies v \notin \overline{r}',$ otherwise there would exist $p_3 \in \mathbb{P}$ such that $v \in p_3$;
$p_2 \cap p_3 \neq \emptyset$ and $p_3 L \overline{r}'$ would imply Condition 2 violation.
$\implies \exists p'' \in \mathbb{P}'$ such that $p'' L' \overline{r}', u \in p''$ and $v \notin p''$.
$Since, p' \cap p'' \neq \emptyset$ and $(p', c), (p'', c) \in M \circ L'$
Hence, contradiction (Condition 2 violation).

Case (2) $\overline{r}_1 \neq \overline{r}_2$:

(2a): $u \in p_1 \setminus p_2$ (or $v \in p_2 \setminus p_1$, wlog)
Since, $u \in p_1, p_1 L \overline{r}_2$, therefore $u \in \overline{r}_2$
$\implies \exists p_3 \in \mathbb{P}$ such that $u \in p_3$ and $p_3 L \overline{r}_2$
p_2, p_3, p' are such that $u \in p_3, v \in p_2, u, v \in p', p_2 L \overline{r}_2, p_3 L \overline{r}_2, p' L' \overline{r}_2$.
This is not possible using result of Case (1). Contradiction.

(2b): $u, v \in p_1 \cap p_2$
p_1, p_2 has at least one point each that is absent in all the other CPBs (using Lemma 3). Therefore, if u, v, which are present in $p_1 \cap p_2$ are merged in $\mathbb{P}'$, then $|\mathbb{P}'| > |\mathbb{P}|$. Contradiction.
Hence, `Opt-PSD` gives the optimal division. □

7 Constraints Formulation

As just discussed, Projection subspace division outputs a set of CPBs and a mapping function L. These form the input to the *Constraints Formulation* module, whose objective is to construct an LP that captures the projection constraints while ensuring that the solution corresponds to a physically constructible database.

Condition 1 of valid division ensures that each PRB $\overline{r} \in \overline{\mathbb{R}}^{\mathbb{A}}$ is completely covered by a set of CPBs. While Condition 2 ensures that all CPBs related to some $c \in \mathbb{C}$ are mutually disjoint. As a consequence, a constraint $c \langle f, \mathbb{A}, l, k \rangle$ can now be expressed as a summation of cardinalities of CPBs related to c through $M \circ L^{\mathbb{A}}$.

$$|\pi_{\mathbb{A}}(\sigma_f(\mathcal{T}))| = \sum_{p:(p,c) \in M \circ L^{\mathbb{A}}} |p| \tag{7}$$

Further, since each $\overline{r} \in \overline{\mathbb{R}}^{\mathbb{A}}$ is related to at least one $c \in \mathbb{C}$ through $M \circ L^{\mathbb{A}}$, the CPBs associated with $\overline{r} \in \overline{\mathbb{R}}^{\mathbb{A}}$ through $L^{\mathbb{A}}$ are also disjoint. Hence, the cardinality of $\overline{r} \in \overline{\mathbb{R}}^{\mathbb{A}}$ can be represented as a summation of the cardinalities of related CPBs.

$$|\overline{r}| = \sum_{p: p L^{\mathbb{A}} \overline{r}} |p| \tag{8}$$

The LP construction uses the above facts while constructing constraints. Specifically, the LP variables that are constructed, and their interpretations, are as follows:

x_r: total tuple cardinality in $r \in \mathbb{R}$, i.e. $|r|$
y_p: (distinct) tuple cardinality in $p \in \mathbb{P}^{\mathbb{A}}$, i.e. $|p|$ for PAS $\mathbb{A}$.

Given this framework, there are two classes of constraints, *Explicit Constraints* and *Sanity Constraints*, that constitute the input to the LP and are discussed in the remainder of this section.

7.1 Explicit Constraints

These are the LP constraints that are directly derived from the projection constraints. For each projection constraint, $c : \langle f, \mathbb{A}, l, k \rangle$, the following pair of constraints are added:

(a) Total Row Cardinality Constraint

$$\sum_{r:rMc} x_r = l \tag{9}$$

(b) Distinct Row Cardinality Constraint (using Equation 7)

$$\sum_{p:(p,c)\in M\circ L^{\mathbb{A}}} y_p = k \tag{10}$$

7.2 Sanity Constraints

These are the additional constraints necessary to ensure that the LP solution can be used for constructing a physical database instance. Here, there are three types of constraints:

Type 1: These constraints ensure that the row cardinality for each RB and CPB are non-negative in the LP solution. That is,

$$x_r \geq 0, \forall r \in \mathbb{R}, \quad \text{and} \quad y_p \geq 0, \forall p \in \mathbb{P}^{\mathbb{A}}, \text{ for all PAS } \mathbb{A} \tag{11}$$

Type 2: These constraints ensure that the total number of tuples for each RB is greater than or equal to the number of distinct tuples along each applicable PAS for that block. Using Equation 8, these constraints, for each RB r and each of its associated PAS $\mathbb{A}$, are expressed as follows:

$$\sum_{p:pL^{\mathbb{A}}\overline{r}} y_p \leq x_r \tag{12}$$

where $\overline{r} = \pi_{\mathbb{A}}(r)$.

Type 3: Even after satisfying the above sanity constraints, we can still have a situation where the total number of tuples for an RB may be positive while the number of distinct tuples along some projection subspace remains zero. To avoid this scenario, we add the following constraint for each RB r and each of its associated PAS $\mathbb{A}$:

$$x_r \leq |\mathcal{T}| \sum_{p:pL^{\mathbb{A}}\overline{r}} y_p \tag{13}$$

In the above, $\overline{r} = \pi_{\mathbb{A}}(r)$ and $|\mathcal{T}|$ is the cardinality of $\mathcal{T}$, which is an upperbound on x_r. We assume that $|\mathcal{T}|$ is given as an input PIC with no filter predicate.

We had already seen, in Section 4, the explicit constraints for our running example. The associated sanity constraints are shown in the box below:

Type 1	$x_1, x_2, x_3, x_{4a}, x_{4b} \geq 0$
	$y_1^{Amt}, y_2^{Amt}, y_3^{Amt}, y_4^{Amt} \geq 0$
	$y_1^{Qty}, y_2^{Qty}, y_3^{Qty}, y_4^{Qty}, y_5^{Qty}, y_6^{Qty} \geq 0$
Type 2, 3	$y_1^{Amt} + y_2^{Amt} \leq x_1 \leq \lvert\mathcal{T}\rvert(y_1^{Amt} + y_2^{Amt})$
	$y_1^{Amt} + y_3^{Amt} \leq x_2 \leq \lvert\mathcal{T}\rvert(y_1^{Amt} + y_3^{Amt})$
	$y_4^{Amt} \leq x_3 \leq \lvert\mathcal{T}\rvert y_4^{Amt}$
	$y_1^{Qty} + y_3^{Qty} \leq x_2 \leq \lvert\mathcal{T}\rvert(y_1^{Qty} + y_3^{Qty})$
	$y_2^{Qty} + y_6^{Qty} \leq x_3 \leq \lvert\mathcal{T}\rvert(y_2^{Qty} + y_6^{Qty})$
	$y_3^{Qty} + y_4^{Qty} \leq x_{4a} \leq \lvert\mathcal{T}\rvert(y_3^{Qty} + y_4^{Qty})$
	$y_2^{Qty} + y_5^{Qty} \leq x_{4b} \leq \lvert\mathcal{T}\rvert(y_2^{Qty} + y_5^{Qty})$

7.3 Sufficiency for Data Generation

For an RB and an associated PAS, the above sanity constraints ensure that any LP solution can always be used to generate data that conforms to it. Now, since RB is symmetric in nature, data across different PASs can be generated independently and concatenated together. Therefore, the constructed LP is sufficient for data generation.

7.4 Workload Scalability and Robustness

Inspired by graphical model-based table decomposition techniques that were proposed in [8], PiGen adopts an optimization of decomposing the denormalized table $\mathcal{T}$ into a collection of sub-tables based on which attributes co-appear in a constraint. Subsequently, these sub-tables undergo the various partitioning algorithms that were discussed in this paper. This decomposition helps to further reduce the number of variables in the LP. After the LP is solved, the solutions for the sub-tables are merged to get the corresponding synthetic denormalized table.

Other than the above optimization, to handle larger workloads, several heuristics can be adopted. One such heuristic is to not create all the CPBs in one go. Instead, first assume that all the PRBs are mutually exclusive and therefore, create only one CPB per PRB. If with this assumption, the obtained solution has minor errors in satisfying the constraints, prune the creation of other CPBs. If the errors are high, then progressively add more CPBs by now assuming that at most two PRBs intersect, and so on. Being an underdetermined system, there always exist a sparse solution to the LP – therefore, this algorithm is expected to converge quickly. However, from the solution quality perspective, using a sparse solution may not always be desirable, as was also shown in [21]. This is so because, sparse solutions create large holes in the data space, where there are no data points. This can lead to poor accuracy on unseen constraints. Constructing an approximation scheme that achieves better workload scalability while producing qualitatively robust solutions is an area of future research.

8 Data Generation

The LP solution gives the following information:

1. A list of RBs with their corresponding row cardinalities, and

2. For each RB and its associated PASs, a list of CPBs with their associated (distinct) row cardinalities.

Thus far, we have only associated statistical significance to each CPBs, specifying the presence or absence of their tuples in RBs. Now, we drill down to assign intervals for each CPB, thereby producing the summary tabulation for all RBs. The CPBs along each PAS are assigned intervals independently since each RB is symmetric along its associated PASs. The final summary that is produced can be used for either on-demand tuple generation, or for generating a complete materialized database instance. We discuss the summary construction and tuple generation procedures here.

8.1 Summary Construction

The summary construction module compactly stores information needed for efficient tuple generation. Each projection subspace is dealt with independently thanks to the projection isolation techniques. Consider the projection subspace corresponding to PAS $\mathbb{A}$ – here, the first step is to assign an interval to each CPB $p \in \mathbb{P}^{\mathbb{A}}$. A challenge in this assignment is that the domains of different CPBs may intersect. For instance, the domains of CPBs p_2^{Qty} and p_6^{Qty} intersect in PURCHASES. However, since CPBs related to a common projection constraint should not intersect, we assign disjoint intervals to these CPBs to ensure Condition 2. Hence, p_2^{Qty} and p_6^{Qty} are allocated disjoint intervals for PAS Qty as $(p_2^{Qty}, c_3), (p_6^{Qty}, c_3) \in M \circ L^{Qty}$. On the other hand, in the case of PAS Amt, p_2^{Amt} and p_4^{Amt} are not related to any c in $\mathbb{C}$, and therefore their data generation intervals can overlap.

As per above, a feasible interval assignment for PURCHASES is:

$p_2^{Amt} \leftarrow [1100, 2500)$	$p_2^{Qty} \leftarrow [20, 25)$
$p_4^{Amt} \leftarrow [500, 3000)$	$p_6^{Qty} \leftarrow [25, 40)$

The summary is maintained RB-wise, with the template structure shown in Figure 7. We see here that all the CPBs associated with the block, along with their distinct tuple cardinalities, are represented in the summary. Using α to denote the total number of associated PASs, an RB can be represented in $\alpha + 1$ components, with each component associated with a PAS having a distinct row-cardinality. For the attribute-set on which no projection is applied for the RB, shown as $\mathbb{A}_{left}$, the domain of the projection block is kept as is and no distinct tuple count is maintained. Lastly, each RB has an associated total cardinality. A populated instance of the template, and its interpretation, was discussed earlier in Section 4.5.

$\mathbb{A}_1$	$\mathbb{A}_2$	...	$\mathbb{A}_\alpha$	$\mathbb{A}_{left}$	
CPB_1: card.,	CPB_1: card.,	...	CPB_1: card.,	PB	RB Card.
CPB_2: card.,	CPB_2: card.,	...	CPB_2: card.,		
...	...	...	...		

Figure 7: Sample RB in Summary

8.2 Tuple Generation

Using the information in the summary, the tuples of the table are instantiated. Specifically, the algorithm iterates over each RB and generates the number of rows specified in the associated total cardinality

value. For an RB and an associated PAS $\mathbb{A}$, each CPB is picked and the corresponding partial tuples are generated. This gives a collection of partial tuples for $\mathbb{A}$ which may be less than the total cardinality. To make up the shortfall without altering the number of distinct values, we repeat the generated partial tuples until the total cardinality is reached. For the $\mathbb{A}_{left}$ component, which only has a single interval, any partial-tuple within its boundaries can be picked for repetition. Finally, partial-tuples across all projection spaces of the RB are concatenated to construct its output tuples.

Inter-Block Dependencies. We have to ensure that the partial-tuples associated with a CPB are identical for each of the associated RBs. To do so, we employ a deterministic algorithm that takes an interval and a cardinality as input, and produces a series of distinct points, equal to the cardinality, from the interval – this series is used in all the associated RBs. As a case in point, for the sample summary in Figure 3, the partial tuples generated for the CPB with interval $[20, 25)$ and distinct row cardinality 5 are identical in both r_3 and r_{4b}.

9 PiGen Pipeline

The end-to-end PiGen pipeline, which extends the Hydra framework to incorporate Projection, is shown in Figure 8. The modules that differ from Hydra are shown in green color.

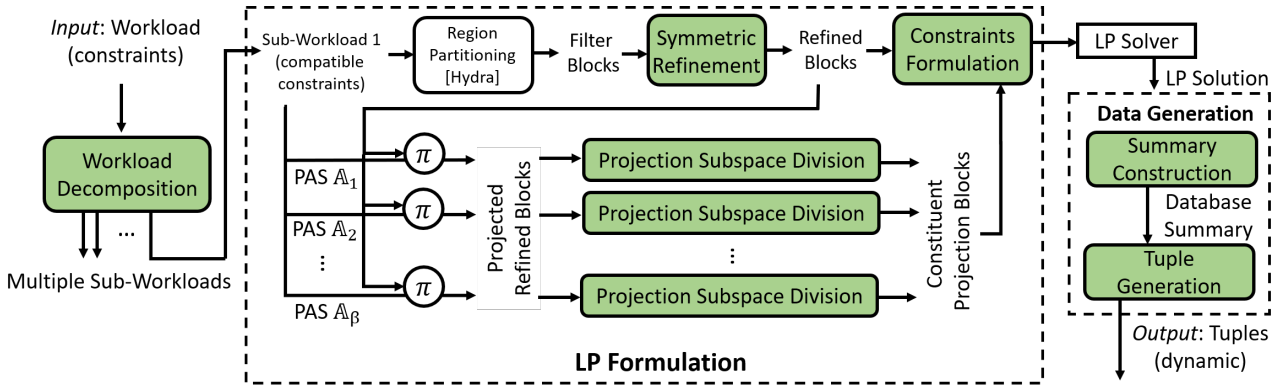


Figure 8: PiGen Algorithm Pipeline

PiGen takes a workload $\mathbb{W}$ of projection-inclusive constraints over a single denormalized table $\mathcal{T}$ as input. Let β be the total number of PASs across all the constraints, as indicated in Figure 8. From the constraints, PiGen produces data for $\mathcal{T}$. This is carried out by a sequence of core components, namely *Workload Decomposition*, *LP Formulation*, and *Data Generation* modules. **Workload Decomposition** splits $\mathbb{W}$ into a set of compatible sub-workloads. Subsequently, the rest of the pipeline, comprising of LP Formulation and Data Generation, is executed independently for each of these sub-workloads. The **LP Formulation** for a sub-workload $\mathbb{C}$ begins with **Region Partitioning** followed by **Symmetric Refinement** algorithm. This gives the set of RBs. For each PAS across all PICs, the PRBs are computed using the RBs. These PRBs and $\mathbb{C}$ are then used by the **Projection Subspace Division** module to construct the set of CPBs. Next, at the **Constraints Formulation** stage, an LP is constructed using variables representing the cardinalities of RBs and CPBs. This construction is then given as the input to the **LP Solver**. From the solution produced by the LP solver, a comprehensive table summary is constructed using the **Summary Construction** module. This summary is used by the **Tuple Generation** module to synthesize the data. It can generate tuples on-demand during query processing, thereby

eschewing the need for data materialization. Alternatively, if the user desires a materialized database instance, it can be generated from the summary and stored persistently.

Finally, PiGen leverages the graphical model-based table decomposition techniques proposed in [8] to construct the table in a piece-meal fashion and then stitch these constituent pieces together. Each sub-table consists of a subset of attributes determined by the attributes that co-appear in the PICs, thereby further reducing the LP complexity.

Having presented the mechanics of PiGen, we now take a step back and critique the approach on relevant aspects.

9.1 Workload Feasibility

Feasibility of a set of PICs implies that the PICs can be accurately satisfied by a single database instance. This notion can be classified into the following two scenarios:

Intra-PIC Feasibility This form of feasibility deals with PICs at an individual level. Specifically, a PIC $c : \langle f, \mathbb{A}, l, k \rangle$ is feasible iff:

$$0 < k \leq l \leq |\mathcal{T}| \text{ or } l = k = 0 \tag{14}$$

9.1.1 Proof

We give a data construction to prove feasibility: Generate values for $\mathbb{A}$ such that there are k unique value combinations (satisfying f) repeated over to make the total number rows to be l. Choose any values for the rest of the attributes such that complete tuple satisfies f. This will ensure the data always satisfies the constraint.

Inter-PIC Feasibility This is a stronger form of feasibility, where in addition to PICs being individually feasible, they are also required to be mutually compatible. For instance, consider the additional constraint, c_6, on the PURCHASES table:

$$c_6 : \langle Amt \leq 2000 \wedge Year \geq 2000, Qty, 400, 25 \rangle$$

We observe here that c_4 and c_6 cannot be satisfied together. Specifically, c_6 requires 25 distinct Qty values for the range $Amt \leq 2000 \wedge Year \geq 2000$, while c_4 requires that the number of distinct $(Qty, Year)$ pairs is 20 for a larger covering range, constituting an impossible situation.

Defining a set of necessary and sufficient conditions that ensure solution feasibility for various types of input constraints has been looked at in the database literature. For instance, [17] deals with *schematic* constraints on the participation cardinalities for the relationships between entities in the ER model, and provides necessary and sufficient conditions to determine whether database instances exist such that all entities and relationships are populated. However, giving a similar holistic solution in the *statistical* query-based constraints space, is still an open problem, although restricted versions have been attempted. Specifically, feasibility of projection cardinality constraints has been discussed in [12, 26, 16]. A class of constraints, called **BT** inequalities, were proposed in [12], which capture the necessary conditions to be satisfied by the projection output cardinalities. However, this constraint set is not sufficient, making it still possible that no actual database can satisfy these values. Subsequently, another class of constraints, called **NC** (non-uniform cover) inequalities, was proposed in [26]. While this constraint set creates sufficient conditions for database construction, the limitation is that satisfiability of these conditions is not guaranteed. Further, the feasibility space does not exhibit a convex behaviour, making it inexpressible as a set of linear constraints [16].

9.2 Solution Guarantees

We discuss the solution guarantees for feasible and infeasible workloads separately below.

Feasible Workload The input workload feasibility is true by definition when the PICs have been derived from an existing setup. In such scenarios, PiGen ensures, thanks to the explicit LP constraints, that the generated data satisfies the PICs with 100% accuracy. Further, the sanity constraints ensure the LP solution is always constructible. This leads us to the following lemma:

Lemma 5. *For a feasible and compatible set of PICs, PiGen always produces an instance of the table that satisfies all the constraints.*

Given an initially feasible workload, workload-decomposition can always produce sub-workloads that are both feasible and compatible. Therefore, for any initially feasible workload, the data produced by PiGen can cover all the input constraints. We formally prove Lemma 5 next.

Proof of Lemma 5

We briefly discuss the proof for Lemma 5, which is split into two parts: (a) The LP constructed for a feasible compatible workload $\mathbb{C}$ is always satisfiable; (b) Given any LP solution, data can be always be constructed from it, and this data will satisfy $\mathbb{C}$.

Part (a): Given workload feasibility, there exists at least one instance T of the table that satisfies $\mathbb{C}$. Further, due to compatibility, $\mathbb{C}$ is modeled in a single LP. Say T does not satisfy this LP. This implies T does not satisfy at least one of the Explicit or Sanity constraints. If T violates an Explicit constraint, then it does not satisfy at least one input PIC. This is because each input PIC is modeled using two Explicit constraints that ensure the data satisfies the PIC. Further, there cannot be a physical table that violates any Sanity constraint due to its inherent nature. Hence, T satisfies all the Sanity constraints as well. Therefore, by contradiction, we can conclude that T satisfies the LP – in fact, the LP gives the *necessary* conditions for data generation adhering to the workload. This implies that for feasible workloads, the LP is satisfiable.

Part (b): For a particular PAS $\mathbb{A}$, the Sanity constraints ensure that for each populated RB, the total tuple count in the RB is at least the number of distinct rows along $\mathbb{A}$, and the distinct row count is positive. Hence, the data along each projection subspace is generated easily. Further, since RB is symmetric in nature, data across its different projection subspaces can be generated independently and concatenated. Therefore, any LP solution is *sufficient* for data generation. Since, each PIC is modelled in the LP using the Explicit constraints, the generated data is compliant with $\mathbb{C}$.

Infeasible Workload Intra-PIC feasibility check can be trivially verified at the pre-processing stage by checking the adherence of constraints to Condition 14. However, if the input has inter-PIC infeasibility, the following two possibilities may arise: (a) It may so happen that Workload Decomposition, while resolving intersection PICs, may as a *collateral benefit*, also produce feasible sub-workloads. For example, by partitioning the workload $\{c_4, c_5, c_6\}$ into $\{c_4\}$ and $\{c_5, c_6\}$, the resulting sub-workloads become non-intersecting as well as feasible. In this scenario, PiGen produces one table for each sub-workload (using Lemma 5). (b) Alternatively, in case this beneficial effect of decomposition does not happen, then the LP constraints (discussed in Section 7) themselves become infeasible. Hence, the LP solver eventually flags this infeasibility. We have explicitly verified this to be the case for the Z3 solver with a few deliberately created infeasible constraint sets.

9.3 Solution Complexity

Computationally, the bottleneck of the pipeline lies in the LP solver. The LP complexity is primarily governed by the number of CPBs created, which is determined by the overlaps between the blocks intra-projection subspaces. The extent of overlaps is reflected by the outdegree of vertices in the Division Graph $G(V, E)$ introduced in Section 6. For adversarial cases, the number of CPBs can be as high as the number of connected induced subgraphs of G, which can go up to $2^{|V|}$.

Connection to Connected Induced Subgraph Problem

Assuming the domain of all the blocks are identical, then the number of CPBs is identical to the number of connected induced subgraphs in G. This can be proved by a straightforward *bijection* argument. That is, each induced connected subgraph (A) has a corresponding CPBs (B) in the solution and vice versa.

Further, $|V|$ itself is $\mathcal{O}(2^{|\mathbb{W}|})$. However, these worst-case exponential scenarios are relatively rare in practice, and our experience is that the count is usually well within the solver's computational limits. We quantitatively assess this aspect in our experimental evaluation (Section 10).

Further, for infeasible workloads, the only additional overheads incurred are the checks for intra-PIC feasibility. This verification takes constant time for an input PIC.

Lastly, the decision version of the general data generation problem is `NEXP-complete`, as shown in [8].

9.4 Limitations and Extensibility

While PiGen takes a substantive step towards addressing the primary challenges of projection modeling, there are some practical limitations wrt its current coverage and scope, as described next.

Multiple Summaries We would ideally like to produce a single summary instance that satisfies all the PICs. However, PiGen may have to produce multiple summaries, and hence multiple databases, to cater to constraint workloads that feature overlapping projection spaces. From a practical perspective, this multiplicity does not impose a substantive overhead due to the minuscule size of each summary. Further, PiGen attempts to reduce the number of sub-workloads to the minimum required to ensure compatibility.

Workload Scale Despite the proposed techniques provably gives minimal number of variables needed for expressing PICs, they can still be exponential if the input PICs have high overlaps intra projection subspaces. PiGen currently handles workloads of reasonable complexity as showcased in our experiments. However, for more complex scenarios, a promising recourse is to introduce *approximation*, where volumetric accuracy is marginally compromised to achieve solution tractability. For example, a plausible heuristic could be to not create all the CPBs in one go, but to create them greedily until the error limit is reached. Being a highly underdetermined system, there always exist a sparse solution to the LP – therefore, this iterative process is expected to converge quickly. However, from the solution quality perspective, using a sparse solution may not always be desirable, as was also shown in [21]. This is so because, sparse solutions create large holes in the data space, where there are no data points. This can have robustness limitations in assessing performance of unseen queries. Constructing an approximation scheme that achieves better workload scalability while producing qualitatively robust solutions is an area of future research.

Incremental Workloads Currently the entire constraint workload is assumed to be given as the input. An alternative scenario is where the constraints are incrementally provided. This may appear problematic since PiGen does not allow modifying the solution to satisfy additional constraints. However, its data-scale-free summary creation permits rebuilding the solution from scratch cheaply.

10 Experiments

In this section, we evaluate the empirical performance of a Java-based implementation of PiGen. The popular Z3 solver [6] is invoked by the tool to compute the solutions for the LP formulations. Our experiments cover the accuracy, time and space overheads and scalability aspects of PiGen, and are conducted using the PostgreSQL v9.6 engine [3] on a vanilla HP Z440 workstation.

Workload Construction In presenting the experimental results, we initially focus on fully compatible workloads. Subsequently, in Section 10.5, we discuss the corresponding performance for workloads featuring intersection. A variety of real world and synthetic benchmarks were used in designing the workloads. For representative large fact tables from each of the benchmarks, a workload of compatible PICs was derived by executing a set of queries. The denormalized versions of these tables were considered for constructing PICs. The details of the compatible workloads are as follows:

TPC-DS Suite: This suite comprises of four workloads, corresponding to the four TPC-DS tables [4] subject to the maximum number of projection operations in the benchmark – namely, STORE_SALES (SS), CATALOG_SALES (CS), WEB_SALES (WS), and INVENTORY (INV).

Census Workload: Here, the **Census** dataset framework used in [14] is extended to additionally feature projections apart from the extant filter cardinality constraints. In particular, a single workload was constructed on the PERSONS (P) table.

IMDB Suite: This suite is designed from the JOB [2] benchmark based on the **IMDB** dataset. It comprises of three workloads, corresponding to the three tables subject to the maximum projection operations – namely, MOVIE_KEYWORD (MK), CAST_INFO (CI), and MOVIE_COMPANIES (MC).

The complexity of these various workloads is quantitatively characterized in Table 4. Note that they feature a substantial degree of both inter-projection complexity (up to 10 projection subspaces and 6 dimension subspaces) and intra-projection complexity (maximum degree of the Division Graph vertices goes as high as 72).

Baselines We compare PiGen against the **DataSynth** and **Hydra** frameworks which both support strict cardinality constraints. For DataSynth, projection constraints need to be restricted to single attribute tables, whereas in Hydra, only the filter constraints are considered in the generation process. We deliberately omit the evaluation of systems dealing with parameterized cardinality constraints such as Touchstone [18] and MyBenchmark [19]). This is due to the organic differences, highlighted in Section 2 between their problem framework and ours, which render quantitative comparisons to be infructuous.

Table 4: Workload Complexity

Dataset	Table	# PICs	# PASs	PAS Length		Vertex Degree	
				Avg.	Max.	Avg.	Max.
TPC-DS	SS	16	8	1.4	5	3.95	10
	CS	15	10	2.2	5	4.74	15
	WS	16	8	2	6	5.7	16
	INV	6	3	1.5	4	0.92	4
Census	P	220	3	1.67	2	1.33	72
IMDB	MK	16	4	1.25	2	5.68	14
	CI	14	3	2.67	3	3.7	17
	MC	19	4	1.5	2	3.75	15

10.1 Constraint Accuracy

When PiGen was run on the aforementioned workloads, the generated data satisfied all the constraints with **100% accuracy**. To appreciate the complexity present in these successfully modeled constraints, we present a representative sample constraint applied on the denormalized relation of STORE_SALES from TPC-DS below:

$$\mathbf{c} : \langle \mathbf{f}, \mathbb{A}, \mathbf{31921358}, \mathbf{15061} \rangle$$

$\mathbf{f} : d\_year = 2002 \ \wedge$
$(i\_category \in$ ('Jewelry','Women') $\wedge \ i\_class \in$ ('mens watch','dresses')) $\vee$
$(i\_category \in$ ('Men','Sports') $\wedge \ i\_class \in$ ('sports-apparel','sailing')) and
$\mathbb{A} : \{i\_category, i\_brand, s\_store\_name, s\_company\_name, d\_moy\}$.

Note that there are several attributes in the projection set, and both conjunctive and disjunctive predicates in the filter condition.

When the same experiments were carried out with Hydra, we found that typically over 90% of the constraints had a relative error of greater than 90%.A sample accuracy graph that bears testimony to this behavior is shown in Figure 9, covering all four tables of the TPC-DS workload suite. These observations highlight that it is non-trivial to satisfy projection cardinalities in the synthetic database without explicit modeling and catering to these constraints, as done by PiGen.

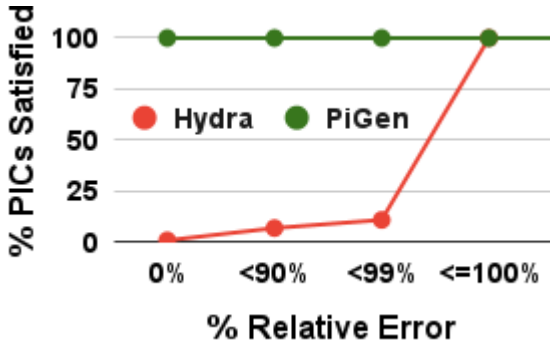


Figure 9: Constraint Accuracy

Turning our attention to DataSynth, we also generated a customized workload from the TPC-DS benchmark, comprising of only single attribute projection and filter constraints to suit DataSynth's

restricted environment. For a single attribute case, there is only one projection subspace possible. Further, two distinct tuples cannot overlap in projection subspace either. Therefore, the inter projection subspace and intra projection subspace challenges do not surface. Even for this simplified scenario, we found several cases, where the LP solution obtained from DataSynth was *inconstructible*. An example illustration showcasing this fundamental problem is shown below:

Consider a toy example with the following pair of projection-inclusive constraints (PICs) on the ITEM table from TPC-DS:

$$PIC\ 1 : \langle 4 \leq i\_class\_id < 12, i\_class\_id, 6876, 8 \rangle$$
$$PIC\ 2 : \langle 8 \leq i\_class\_id < 16, i\_class\_id, 4490, 8 \rangle$$

DataSynth's algorithm divided the domain of i_class_id attribute into five intervals and further assigns total row cardinality and distinct row cardinality to each of these intervals. The obtained boundaries and the two cardinalities for each interval is tabulated in the table below. We can see from the table that for intervals I_1 and I_2, the total row cardinality is positive, while the distinct row cardinality is 0. Since to populate an interval, at least one (distinct) tuple is necessary, therefore, this solution can not produce a valid instantiated table.

For this scenario, DataSynth produced the following interval-based solution:

Table 5: LP Solution from DataSynth

Interval	Range	Total, Distinct Row Card.
I_1	$i\_class\_id < 4$	**11124,0**
I_2	$4 \leq i\_class\_id < 8$	**2386,0**
I_3	$8 \leq i\_class\_id < 12$	4490,8
I_4	$12 \leq i\_class\_id < 16$	0, 0
I_5	$i\_class\_id > 16$	0,0

Due to this clear inability of both DataSynth and Hydra to produce data that satisfies projection-compliant constraints, we restrict our attention to PiGen in the rest of this section.

10.2 Generated Data

We now show a concrete example of how the data generated by PiGen satisfies the input PICs. Consider the following PIC from the CENSUS workload on the PERSONS table:

$$\langle 18 \leq Age \leq 85 \wedge Relationship = \text{'Spouse'} \wedge PUMA = 822, (Age, Sex), 205, 4 \rangle$$

A snippet of the generated table is shown in Table 6. Here, the first four rows in the (Age, Sex) columns are repeated in round-robin fashion, while the remaining attributes have a fixed constant value, for producing the first 205 rows. Then, the subsequent rows (206th row onwards) in the table are assigned values that do not satisfy the above constraint.

10.3 Time and Space Overheads

Having established the accuracy credentials of PiGen, we now turn our attention to the associated computational and resource overheads. To begin with, the summary construction times and sizes for various summary tables are reported in Table 7. We see here that the time to produce the summary is

Table 6: Sample Rows produced for PERSONS Table

Age	Sex	Relationship	PUMA	Tenure
18	M	Spouse	822	Rented
25	F	Spouse	822	Rented
36	M	...	...	...
68	M	...	...	...
Repeated in Round Robin		Spouse	822	Rented
(Row # 206) 76	F	Parent	100	Owned
...	...	...	...	...

Table 7: Overheads

Table	Summary	
	Time	Size
SS	21 min	58 kB
CS	32 min	117 kB
WS	15 min	64 kB
INV	2 s	13 kB
MK	2 min	15.5 kB
CI	41 s	13.6 kB
MC	3.6 min	27.7 kB
P	30 min	416 kB

Table 8: Block Profiles

Table	Cardinality of		
	FBs	RBs	CPBs
SS	74	88	132662
CS	139	141	165936
WS	119	132	73929
INV	11	16	41
MK	30	32	30083
CI	278	301	14386
MC	187	203	42835
P	1193	1529	7170

in a few tens of minutes. From a deployment perspective, these times appear acceptable since database testing is usually an offline activity. Moreover, the summary sizes are miniscule, just a few 100s of kilobytes at most.

Drilling down into the summary production time, we find that virtually all of it is consumed in the LP solving stage. In fact, the collective time spent by the other stages was less than *ten seconds* in all the cases. These results highlight the need for minimizing the number of LP variables, since the solving time is largely predicated on this number. To obtain a quantitative understanding, we report the sizes of the intermediate results at various pipeline stages in Table 8 – specifically, the table shows the number of FBs, RBs, and CPBs created by PiGen. We see here that there is huge jump in the number of regions from the initial FB to the final CPBs, testifying that $\mathbb{C}$ has considerable overlap among its constraints, and therefore represents a "tough-nut" scenario wrt projection. An exception to this observation is the PERSONS table from Census dataset, where even though the maximum degree for a vertex in the Division graph was 72 (Table 4), the overlaps between PICs are limited as also indicated by the average degree which is less than 2.

We also show the improvement of `Opt-PSD` over `Pow-PSD` by additionally reporting the number of CPBs in case of `Pow-PSD`. Lastly, the speedup achieved by `Opt-PSD` over `Pow-PSD` is shown in the last column of the table. Typically, for larger inputs, the speedup achieved is also high. As a case in point, for store_sales table, `Opt-PSD` completed 70 times faster than `Pow-PSD`. In absolute terms also, while `Pow-PSD` took days to produce the summary, `Opt-PSD` completed the process in a few minutes.

We also evaluated the time taken to flag infeasibility by PiGen for the cases where the input workload itself has infeasible PICs. In our experience, this situation was usually caught within a few minutes. As a case in point, on adding an infeasible constraint to the 220 PICs set for CENSUS data, the error was

Table 9: No. of Blocks and Comparison against Pow-PSD

Table	# FBs	#RBs	#PRBs Opt-PSD	#PRBs Pow-PSD	Multiplicative Speed-up
ss	74	88	132662	524404	70
cs	139	141	165936	524336	16

flagged in 3 minutes.

The summarized table can be used to generate tuples either in-memory during query processing, or to produce materialized instances. The time to generate the tuples from the summary in-memory is reported in Table 10, and we see that even a huge table such as SS, having close to 3 billion records, is generated within a few minutes.

Table 10: Tuple Generation Time

Table	# Rows	Tuple Gen. Time	Table	# Rows	Tuple Gen. Time
SS	2.9 bn	4 min	WS	0.72 bn	8 seconds
CS	1.4 bn	1.5 min	INV	0.78 bn	9 seconds

10.4 Scalability Profile

The scalability aspect of PiGen was discussed at length in Section 9, and we now provide quantitative observations with regard to data and workload scale.

Data Scale The time and space overheads incurred to produce table summaries are intrinsically *data-scale-free*, i.e., they do not depend on the generated size. We explicitly verified this property by running PiGen over 10 GB, 100 GB and 1 TB versions of TPC-DS.

Workload Scale The time and space requirements with increasing number of PICs is shown in Figures 10(a) and 10(b), respectively, for the Census workload. The figures highlight that the memory consumption is relatively stable and manageable (few GB) across the spectrum, but that time scalability can be a limitation for workloads beyond a certain complexity (Figure 10(a) is on a log scale).

10.5 Workload Decomposition

We now turn our attention to intersecting workloads, which require the pre-processing step of workload decomposition. To model this scenario, we added intersecting PICs to the TPC-DS workload suite, with the final workloads having the following PIC distributions: SS (52 PICs), CS (28 PICs), WS (29 PICs), and INV (8 PICs). In particular, we have evaluated the PiGen results on $\mathbb{W}$ for two decomposition strategies: (a) Instance-based Decomposition, and (b) Template-based Decomposition, which are discussed below.

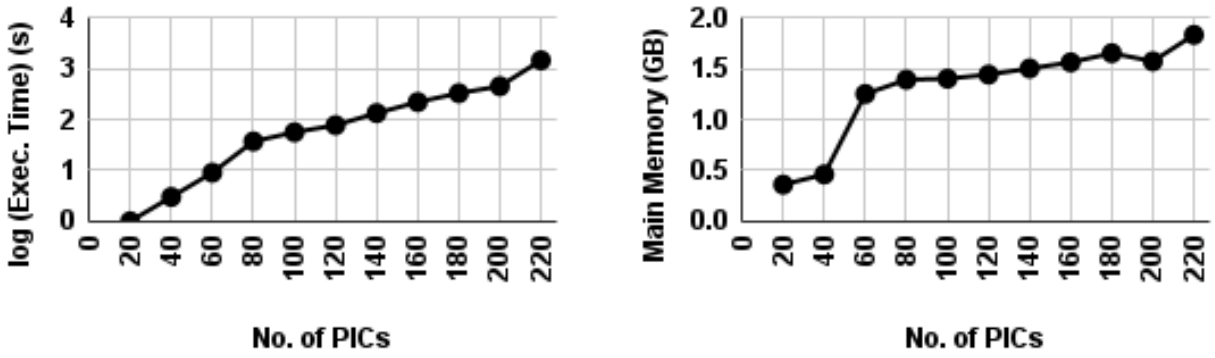


Figure 10: (a) Execution Time (b) Memory Usage

10.5.1 Instance-based Decomposition (ID)

Here the decomposition algorithm uses Definition 5 of a conflicting pair, and for this framework, the number of workloads obtained for the four tables are shown in Table 11. We observe that despite using an approximate vertex coloring algorithm (Section 5.2), a partitioning of $\mathbb{W}$ into at most 6 sub-workloads sufficed for ensuring internal compatibility. Interestingly, the aggregate summary generation times are extremely small, completing in just a few seconds, and much lower than the corresponding numbers for $\mathbb{C}$ in Table 7. At first glance, this might appear surprising given that $\mathbb{W}$ is more complex in nature – the reason is that due to workload decomposition, an array of databases is produced for $\mathbb{W}$ with low individual production complexity, whereas a single unified database is produced for $\mathbb{C}$. From a testing perspective, it is preferable to generate the minimum number of databases, and therefore we would always strive to have as little decomposition as possible.

Table 11: Workload Decomposition - ID

Table	Sub-Workload Sizes	Aggregate Summary Time	Aggregate Summary Size
SS	13,11,8,7,7,6	14 s	135 kB
CS	14,5,5,4	12 s	69 kB
WS	12,10,7	7 s	58 kB
INV	6,2	3 s	16 kB

10.5.2 Template-based Decomposition (TD)

Here, the decomposition algorithm assumes conflicting pairs are defined at a template level. That is, two constraints conflict if their PASs partially intersect. The reason we consider TD is to remove any coincidental performance benefit that may have been obtained thanks to the specific filter predicate constants present in the original workload. Table 12 shows the number of workloads obtained for the four tables with this artificially expanded definition of conflict. We observe that even here, just 8 sub-workloads are sufficient for producing compatibility. Finally, again thanks to decomposition, both the summary generation times and the summary sizes are extremely small.

Finally, we also verified the quality of the approximation algorithm for decomposition. That is, how far is the obtained number of sub-workloads from the actual minimum count. To assess this, we implemented the exponential algorithm that computes the true minimum number of sub-workloads

Table 12: Workload Decomposition - TD

Table	Sub-Workload Sizes	Aggregate Summary Time	Aggregate Summary Size
SS	10,10,8,8,5,5,4,3	70 s	109 kB
CS	9,7,4,4,4	14 s	117 kB
WS	9,9,6,5	7 s	41 kB
INV	6,2	2 s	16 kB

and in the cases where this exhaustive algorithm could be evaluated, we found that the approximation algorithm returned the same count as the optimal.

11 Conclusions

Synthetic data generation from a set of cardinality constraints has been strongly advocated in the contemporary database testing literature. PiGen expands the scope of the supported constraints to include, for the first time, the general Projection operator. The primary challenges in this effort were tackling dependencies within a projection subspace and across different projection subspaces. By using a combination of workload decomposition and symmetric refinement, dependencies across various projection subspaces were handled. Within a projection subspace, union was converted to summation via division of the space. Further, an optimal division strategy was presented to construct efficient LP formulations of the constraints. The experimental evaluation on real-world and synthetic benchmarks indicated that PiGen successfully produces generation summaries with viable time and space overheads.

Currently, PiGen deems any exact solution to the LP as satisfactory for database generation. This choice could be materially improved in two ways: 1) By using approximation algorithms that sacrifice constraint accuracy to a limited extent to achieve better workload scalability; and 2) By preferentially directing the LP solver towards solutions with reduced sparsity so as to improve the robustness of the generated database to future unseen queries.

References

[1] Dagstuhl Seminar 21442. Ensuring the Reliability and Robustness of Database Management Systems. dagstuhl.de/en/program/calendar/semhp/?semnr=21442

[2] JOB Benchmark. github.com/gregrahn/join-order-benchmark

[3] PostgreSQL. postgresql.org/docs/9.6

[4] TPC-DS. tpc.org/tpcds/

[5] TPC-H. tpc.org/tpch/

[6] Z3. github.com/Z3Prover/z3

[7] A. Alexandrov, K. Tzoumas, and V. Markl. Myriad: Scalable and Expressive Data Generation. *PVLDB*, 5(12):1890-1893, 2012.

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

[9] A. Arasu, R. Kaushik, and J. Li. DataSynth: Generating Synthetic Data using Declarative Constraints. *PVLDB*, 4(12):1418-1421, 2011.

[10] C. Binnig, D. Kossmann, and E. Lo. Reverse Query Processing. *Proc. of 23rd ICDE Conf.*, 2007, pgs. 506-515.

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

[12] B. Bollobás and A. Thomason. Projections of Bodies and Hereditary Properties of Hypergraphs. *Bulletin of the London Mathematical Society*, 27(5):417-424, 1995.

[13] N. Bruno and S. Chaudhuri. Flexible Database Generators. *Proc. of 31st VLDB Conf.*, 2005, pgs. 1097-1107.

[14] A. Gilad, S. Patwa, and A. Machanavajjhala. Synthesizing Linked Data Under Cardinality and Integrity Constraints. *Proc. of ACM SIGMOD Conf.*, 2021, pgs. 619-631.

[15] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. *Proc. of ACM SIGMOD Conf.*, 1994, pgs. 243-252.

[16] I. Leader, Z. Randelovic, and E. Raty. Inequalities on Projected Volumes. *arXiv:1909.12858*

[17] M. Lenzerini, and P. Nobili. On The Satisfiability of Dependency Constraints in Entity-Relationship Schemata. *Proc. of 13th VLDB Conf.*, 1987, pgs. 147–154.

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

[19] E. Lo, N. Cheng, W. W. Lin, W.-K. Hon, and B. Choi. MyBenchmark: generating databases for query workloads. *The VLDB Journal*, 23(6):895-913, 2014.

[20] T. Rabl, M. Danisch, M. Frank, S. Schindler, and H. Jacobsen. Just can't get enough - Synthesizing Big Data. *Proc. of ACM SIGMOD Conf.*, 2015, pgs. 1457-1462.

[21] A. Sanghi, Rajkumar S., and J. R. Haritsa. Towards Generating HiFi Databases. *Proc. of 26th DASFAA Conf.*, 2021, pgs. 105-112.

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

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

[24] E. Shen, and L. Antova. Reversing statistics for scalable test databases generation. *Proc. of DBTest Workshop*, 2013, pgs. 1-6.

[25] A. Silberschatz, H. F. Korth, and S. Sudarshan. Database System Concepts, McGraw-Hill, New York, Seventh Edition, 2020.

[26] Z. Tan, and L. Zeng. On the Inequalities of Projected Volumes and the Constructible Region. *SIAM Journal on Discrete Mathematics*, 33(2):694-711, 2019.