Projection-Compliant Database Generation

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology
IN
Faculty of Engineering

BY

Shadab Ahmed

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

July, 2021

Declaration of Originality

I, **Shadab Ahmed**, with SR No. **04-04-00-10-42-19-1-16671** hereby declare that the material presented in the thesis titled

Projection-Compliant Database Generation

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2019-2021**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: 10th July, 2021 Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof. Jayant R. Haritsa Advisor Signature

DEDICATED TO

the future IISc students.

May they have a good time.

Acknowledgements

I would like to express my gratitute to my advisor Prof. Jayant R. Haritsa for giving me the opportunity to work on this project. I am grateful for his guidance throughout this project.

I would like to thank Anupam Sanghi for mentoring and helping me every step of the way. His collaboration has helped this work to reach a good end. I would also like to thank my lab mates for their help and for making my time in lab fun. I am grateful to Tarun Kumar Patel and Subhodeep Maji for helping me with the project on numerous occasions.

I would also like to express my gratitude to the Department of Computer Science and Automation for providing a good and friendly environment for learning. The staff has been very helpful.

I am thankful to my family for their support and their belief in me.

Lastly, I am grateful to my friends. I am grateful for the many life lessons that I have learnt from them. I am grateful to them for helping me enjoy life outside of academics. I am also thankful to my friend, Hemanta Makwana, for giving me emotional support that got me through the difficult times.

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 work, 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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database software vendors often need to generate synthetic databases for a variety of applications [15, 5], 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.

Cardinality Constraints

The use of *declarative formalisms* to express data characteristics has been persuasively advocated in contemporary testing frameworks [5, 14, 18]. 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:

$$|\pi_{\mathbb{A}}(\sigma_f(T_1 \bowtie T_2 \bowtie ... \bowtie T_N))| = k \tag{1.1}$$

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) [8].

Data Generation using Cardinality Constraints

Generating synthetic data that adheres to a collection of cardinality constraints was first proposed in the pioneering work of **DataSynth** [5, 6]. This initial effort was later extended in **Hydra** [18, 19] to incorporate dynamism and scale in the generation process. The generic procedure in these frameworks is as follows: For each table T in the database schema, a corresponding *denormalized* table $\mathcal{T}$ is constructed, with the schema comprising the non-key attributes from the source table and the tables to which it is connected through referential constraints. This construction allows replacing the join expression (restricted to PK-FK joins) in each constraint with a single denormalized table. Next, the data space of $\mathcal{T}$ is partitioned into a set of disjoint *filter-blocks*[1] (FBs), determined by the filter predicates in the cardinality constraints. Specifically, an FB b represents a collection of data points that satisfy a particular set of filters. Further, a variable x_b is created for each b, representing its row-cardinality in the synthetic database. Next, a feasibility problem is constructed, where each constraint is expressed as a *linear equation* in these variables. The solution of the problem is used to construct the denormalized table. Finally, from the denormalized tables, the base tables are extracted while ensuring referential integrity.

As a concrete example of this procedure, consider the following scenario: **Example:** A table PURCHASES with non-key columns Qty and Amt has the following three filter constraints:

$$
\begin{aligned}
|(\sigma_{f_1}(\textsc{Purchases}))| &= \ \ 500, \ \ f_1 = (Qty < 20) \wedge (1100 \leq Amt < 2500) \\
|(\sigma_{f_2}(\textsc{Purchases}))| &= 1000, \ \ f_2 = (Qty \geq 20) \wedge (500 \leq Amt < 3000) \\
|(\sigma_{f_3}(\textsc{Purchases}))| &= 3000, \ \ f_3 = (Qty \geq 10)
\end{aligned}
$$

Figure 1.1(a) shows the 2D data space of the Qty and Amt attributes. On this space, the above filter constraints are represented using regions with colored solid-line boundaries. For partitioning the data space, Hydra adopts *region-partitioning* algorithm, which produces FBs such that all the data points that exclusively satisfy a particular set of filters are put in the same FB. For the above example constraints, the algorithm produces the four disjoint FBs: b_1, b_2, b_3, b_4, depicted with dashed-line boundaries. The corresponding linear program (LP) constructed on these FBs is shown in Figure 1.1(b). Here, x_1, x_2, x_3, x_4 correspond to the count variables for FBs b_1, b_2, b_3, b_4, respectively. A possible solution to the LP is: $< x_1 = 500, x_2 = 0, x_3 = 1000, x_4 = 2000 >$. A distinguishing feature in Hydra is that it produces summarized tables in the output, where a single point per FB is picked and the entire row-cardinality of the FB is assigned to that point. Therefore, the table summary corresponding to

[1] The individual sets in a partition are called *blocks*.

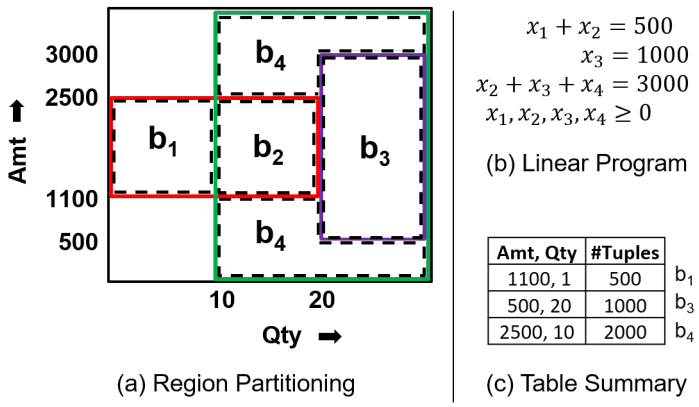


Figure 1.1: Hydra Data Generation

the example LP solution is shown as in Figure 1.1(c). The summarized tables, collectively called the *database summary*, can be used to *dynamically* generate tuples on-demand during query execution, thereby eschewing the need for data materialization.

Incorporating Projection

The above generation process accurately and efficiently satisfies the filter constraints applied on $\mathcal{T}$. However, a critical limitation is that it lacks support for the *projection* operator. Projection forms the core of the DISTINCT, GROUP BY, and UNION SQL constructs, and therefore producing a synthetic database that is projection-compliant would be of considerable value to database vendors. As a case in point, a thorough assessment of a new memory manager's ability to handle hash aggregate/sort operations is predicated on accurate modeling of projection cardinalities. Consequently, in this work, we investigate the explicit incorporation of Projection into the data generation framework. In particular, our focus is on the *duplicate-eliminating* version 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.)

Projection-inclusive Constraints To represent a projection-inclusive cardinality constraint c for 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, k represents the row-cardinality after projecting the filtered table, which is of cardinality l. For instance, the following constraints could represent the post-projection scenario for the three filters from our earlier example:

$$c_1 : \langle f_1, Amt, 500, \mathbf{5} \rangle,\ c_2 : \langle f_2, Amt, 1000, \mathbf{3} \rangle,\ c_3 : \langle f_3, Qty, 3000, \mathbf{9} \rangle$$

As a case in point, constraint c_1 denotes that applying the f_1 predicate on the PURCHASES table produces 500 rows in the output, which is further reduced to 5 rows after projecting on the Amt

column.

Technical Challenges

We now highlight the challenges involved in handling projection-inclusive constraints:

Union Cardinality: While the FBs are mutually disjoint, their projections onto a subspace may *overlap*. Therefore, handling projection constraints requires computing the cardinality of the *union* of FBs obtained after projecting onto the subspace spanned by a given projection attribute-set (PAS).

For instance, leveraging the fact that projection distributes over union [21], we can rewrite projection constraint c_1 in the example as follows, with regard to the FBs in Figure 1.1(a):

$$|\pi_{Amt}(b_1) \cup \pi_{Amt}(b_2)| = 5$$

Here, the union does not translate to the simple summation that sufficed for handling filters. For instance, consider the two points: $u : (Amt = 1500, Qty = 3)$ and $v : (Amt = 1500, Qty = 16)$ from the FBs b_1 and b_2, respectively. The union of projections of u and v along Amt yields a single point – namely, $Amt = 1500$.

Inter-Projection Subspace Dependencies: When an FB b is subjected to multiple projections, the data generation for each projection subspace may be dependent on the others. So, for a pair of PASs $\mathbb{A}_1$ and $\mathbb{A}_2$, sourced from constraints c_1 and c_2, respectively, we have the inclusion property:

$$\pi_{\mathbb{A}_1 \cup \mathbb{A}_2}(b) \subseteq \pi_{\mathbb{A}_1}(b) \times \pi_{\mathbb{A}_2}(b) \tag{1.2}$$

For instance, considering the example FB b_4, $Amt = 2700$ and $Qty = 25$ can belong to $\pi_{Amt}(b_4)$ and $\pi_{Qty}(b_4)$, respectively, but $(Amt = 2700, Qty = 25)$ lies outside the boundary of b_4. Moreover, $\mathbb{A}_1$ and $\mathbb{A}_2$ may be partially intersecting as well. Expressing a general set of projection constraints as linear constraints, while ensuring the solution is physically constructible, is often infeasible – this is because the set of constructible solutions does not form a convex polytope [13].

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 that help attain the desired objective are: (a) *Isolating Projections*, (b) *Projection Subspace Division*, and (c)

Deterministic Instantiation – these principles are discussed in detail in Chapter 2. PiGen has been implemented as a substantive extension of the base Hydra platform. Therefore, it generates summarized tables as its output and the entire pipeline (from constraints processing to summary construction) is data-scale-free. Further, a detailed evaluation on a workload of constraints, derived from TPC-DS benchmark, demonstrates that PiGen accurately and efficiently models Projection outcomes. As a case in point, for a suite of constraints-workloads, comprising over a hundred projection constraints in total, PiGen generated data that satisfied all these constraints, as well as the attendant filter constraints, with *perfect accuracy*. Moreover, the entire summary production pipeline completed in viable time and space overheads.

Organization The remainder of this report is organized as follows: The key design principles of PiGen are highlighted in Chapter 2, and an overview of its data generation pipeline is presented in Chapter 3. Subsequently, the internals of the core components in the pipeline are described in Chapters 4 through 7. The experimental framework and performance results are reported in Chapter 8, while the prior literature is reviewed in Chapter 9. Finally, our conclusions and future research avenues are summarized in Chapter 10.

Chapter 2

Design Principles

We present here the three design principles incorporated in PiGen to address the technical challenges of modeling projection constraints. The PURCHASES table example scenario of the Introduction is used as the running example to explain their operations.

2.1 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 1.1. 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 2.1(a) – it is easy to see that r_{4a} and r_{4b} are symmetric. This refinement allows for the values along different projection subspaces to be generated independently. (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 2.1(a)).

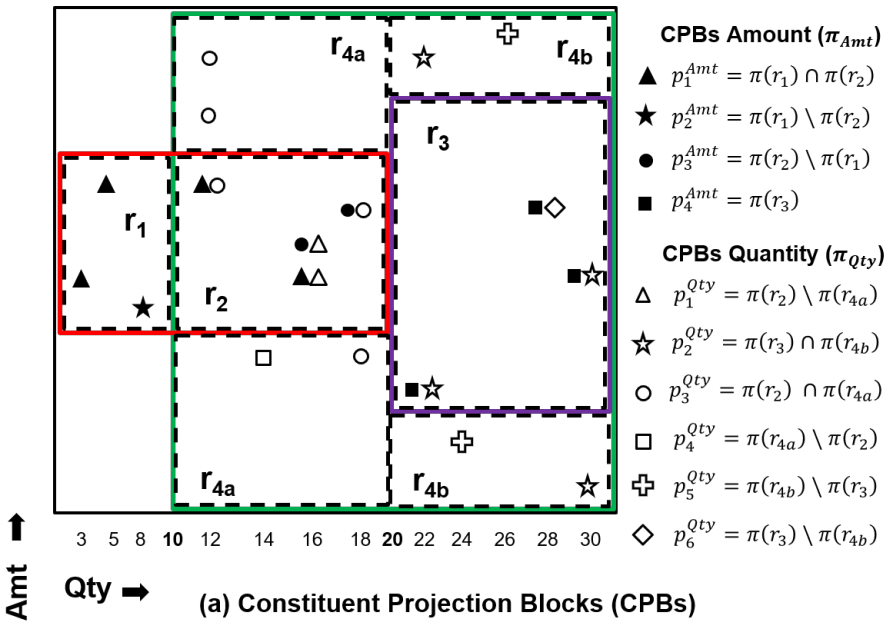
The above refinement, however, does not scale when the projections applied on an FB are along partially overlapping PASs. Therefore, to eliminate such situations, we resort to *decomposing* the workload into sub-workloads using a vertex coloring-based strategy. Further, for each such sub-workload, a summarized table is produced. 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.

2.2 Projection Subspace Division

To deal with the cardinality of union, the domain of each PAS is divided into a set of *constituent-projection-blocks* (CPBs). This construction ensures that projection constraints can be expressed as *summations* over the cardinalities of these CPBs.

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 2.1(a). Each CPB has a semantic meaning associated with it. For example, p_1^{Amt} semantically represents the set of Amt values that are 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 2.1(a) also shows, for the sample table shown in Figure 2.1(b), with its unique tuples enumerated, the CPB (s) to which each of these tuples belongs.



(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 2.1: Projection Subspace Division

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 the

cardinalities of CPBs p_j^{Amt} and p_k^{Qty}, respectively, then the constraints for our example are expressed by the following suite of linear equations:

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

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

2.3 Deterministic Instantiation

Deterministic instantiation of tuples is predicated on well constructed summary. To construct the summary, the domain of each PAS is divided into a set of intervals and then the CPBs are assigned these intervals. This gives the summarized table. A sample summary for the PURCHASES table, with respect to the LP solution:

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

is shown in Figure 2.2 with an additional attribute $Year$ to illustrate a multi-dimensional projection. The summary has a tabulation for each populated RB, comprising of a column for each PAS acting on the RB. In each of these columns, intervals in the projection subspace corresponding to that column is maintained along with their distinct counts. Additionally, another column stores intervals for all the non-projection columns for that RB which contains only intervals but no counts – an example is $Year$ in r_3. 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 two-dimensional interval `[1,10), [1990,2000)`, and the remaining 8 from the two-dimensional interval `[1,10), [2010,2020)`."

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

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 2.2: PiGen Table Summary

by Hydra (Figure 1.1 (c)). The key difference lies in that Hydra neither maintains intervals nor distinct value counts. Specifically, it constructs only one distinct tuple for each FB in the summary, and assigns the entire cardinality to that single tuple.

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 overall PiGen pipeline. and describe the internal details of its major components. The notations used in this delineation are summarized in Table 2.1 for quick reference. Also, for ease of presentation, we will assume that the data type of all columns is *continuous numeric*, but the extension to other types is straightforward.

Table 2.1: Notation Table

Notation	Meaning
$\mathcal{T}$	(Denormalized) Table
$\mathbb{U}$	Set of all attributes in $\mathcal{T}$
c	An input constraint $\langle f, \mathbb{A}, l, k \rangle$, where f is the filter predicate, l is the filtered table row cardinality, k is the row cardinality after projection along PAS $\mathbb{A}$
$\mathbb{W}$	Input constraints workload
$\mathbb{C}$	A compatible constraints workload
PAS	Projection Attribute Set
FB	Filter Block
CPB	Refined Block
PRB	Projected Refined Block
RB	Constituent Projection Block
b	A filter-block (FB)
r	A refined-block (RB)
$\mathbb{R}$	Set of all RBs
M	A relation between $\mathbb{C}$ and $\mathbb{R}$ (Definition 3.2)
$\overline{r}$	projected-refined-block (PRB) wrt r and some PAS
$\overline{\mathbb{R}}^{\mathbb{A}}$	Set of all PRBs for a PAS $\mathbb{A}$
p	An constituent-projection-block (CPB)
$\mathbb{P}^{\mathbb{A}}$	Set of all CPBs for a PAS $\mathbb{A}$
$L^{\mathbb{A}}$	A relation between $\mathbb{P}^{\mathbb{A}}$ and $\overline{\mathbb{R}}^{\mathbb{A}}$ (Definition 5.1)
$(\mathbb{P}^*, L^*)$	Optimal Projection Subspace Division
$\mathbb{V}^*$	Set of mapping vectors corresponding to $(\mathbb{P}^*, L^*)$
x_r	LP variable associated with RB r
y_p	LP variable associated with CPB p
$\mathbb{D}^{\mathbb{A}}$	Data subspace spanned by attribute-set $\mathbb{A}$

Chapter 3

PiGen Pipeline

The end-to-end PiGen pipeline, which extends the Hydra framework to incorporate Projection, is shown in Figure 3.1. The modules that differ from Hydra are shown in green color.

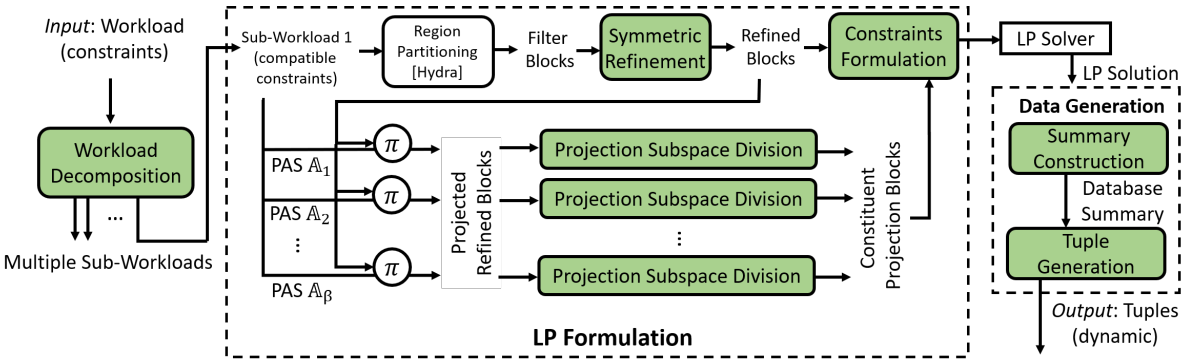


Figure 3.1: PiGen Algorithm Pipeline

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 3.1. 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 the input workload into a set of *compatible* sub-workloads, and the rest of the pipeline is then executed independently for each of these sub-workloads. Specifically, an LP is first constructed which expresses the input (compatible) constraints as linear constraints, and the solution is used by the Data Generation module to produce the tuples. We discuss each of these stages in the remainder of this section.

3.1 Workload Decomposition

The input constraints workload is split into sub-workloads with compatible constraints, since this is a pre-requisite for the LP formulation downstream in the pipeline. Formally, a sub-workload should not contain constraints that result in *conflicting pairs*, where conflict is defined as follows:

Definition 3.1 *A pair of constraints* ($c_1 : \langle f_1, \mathbb{A}_1, k_1, l_1 \rangle, \quad c_2 : \langle f_2, \mathbb{A}_2, k_2, l_2 \rangle$) *conflict 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.

Given an original workload $\mathbb{W}$, the set of conflicting pairs CP is computed first. Subsequently, this module aims to construct the *minimum* number of sub-workloads $\mathbb{W}_1, \mathbb{W}_2, .., \mathbb{W}_n$ such that:

1. $\mathbb{W}_1 \cup \mathbb{W}_2 \cup ... \cup \mathbb{W}_n = \mathbb{W}$.
2. No element of CP is present in any $\mathbb{W}_i$.

Minimization of number of sub-workloads is desirable to ensure larger set of constraints can share a common database. This problem is NP complete (reduction from vertex coloring). Therefore, we adopt a heuristic based on greedy vertex coloring. The algorithm iterates over the constraints, and in each iteration, the constraint c with minimum conflicts in CP is picked and assigned to an existing sub-workload $\mathbb{W}_i$, if doing so does not introduce a conflict. If multiple options are available, then 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.

3.2 LP Formulation

For each sub-workload $\mathbb{C}$ (where $\mathbb{C} = \mathbb{W}_i$, for some $i \in [n]$) derived from the above procedure, FBs based on the filter predicates are constructed using the **Region Partitioning** algorithm of [18], as discussed in Chapter 1. In fact, the algorithm ensures the count of FBs is minimum.

The FBs are refined into a set of RBs, to facilitate isolation of projection subspaces. This is done by the **Symmetric Refinement** module (details in Chapter 4). It ensures that the all the resultant RBs are *symmetric* along each PAS applicable on it. The set of RBs is denoted as $\mathbb{R}$, and is used for expressing the input constraints in the LP. To explain this process, we first define the following relation between the sets $\mathbb{R}$ and $\mathbb{C}$.

Definition 3.2 *An RB $r \in \mathbb{R}$ is related by relation M to a constraint $c \in \mathbb{C}$ containing filter predicate f, iff all the points in r satisfy f. That is,*

$$rMc \Leftrightarrow \sigma_f(r) = r \tag{3.1}$$

For each RB $r \in \mathbb{R}$, its projection on $\mathbb{A}$ is shown as $\overline{r}$, i.e. $\overline{r} = \pi_{\mathbb{A}}(r)$ and is referred to as a *projected-refined-block* (PRB). Further, for brevity, we overload the same relation M to establish an association between PRB $\overline{r}$ and a constraint c. That is, $\overline{r}Mc \Leftrightarrow rMc$. The set of all PRBs for a PAS $\mathbb{A}$ is shown as $\overline{\mathbb{R}}^{\mathbb{A}}$.

A constraint $c \in \mathbb{C}$ can be expressed as a union of the RBs and PRBs related to it by M. Specifically, $c : \langle f, \mathbb{A}, l, k \rangle$ is expressed as:

$$\Big|\bigcup_{r:rMc} r\Big| = l, \quad \Big|\bigcup_{r:rMc} \pi_{\mathbb{A}}(r)\Big| = \Big|\bigcup_{\overline{r}:\overline{r}Mc} \overline{r}\Big| = k \tag{3.2}$$

As discussed earlier, the constraint to ensure total cardinality l can easily be expressed by replacing the unions with summations as the RBs are mutually disjoint. However, since PRBs may share data points, to express the constraint requiring distinct cardinality k of the projection result, the projection subspace needs to be divided into a set of *constituent-projection-blocks* (CPBs).

Let $\mathbb{D}^{\mathbb{A}}$ represent the subspace spanned by PAS $\mathbb{A}$. The objective of **Projection Subspace Division** module (details in Chapter 5) is to partition $\mathbb{D}^{\mathbb{A}}$ associated with each PAS $\mathbb{A}$ that occurs in $\mathbb{C}$. Specifically, each such subspace $\mathbb{D}^{\mathbb{A}}$ is divided into a group of CPBs such that the constraints are expressible as linear equations.

Next, at the **Constraints Formulation** stage (details in Chapter 6), an LP is constructed using variables representing the row cardinalities of RBs and CPBs which is solved by the **LP solver**.

3.3 Data Generation

From the LP solution, a comprehensive table summary is constructed that can be used for on-demand tuple generation during query processing, thereby eschewing the need for data materialization. Alternatively, if the user intends to generate a materialized database instance, that can also be generated from the summary. The details of this module are discussed in Chapter 7.

Chapter 4

Symmetric Refinement

The refinement for each FB b is done independently. Given an FB and its associated PASs, this module refines b into a group of RBs, such that each RB is symmetric along the input PASs. That is, for each domain point of an RB r along an applicable PAS, the projection of r along the remaining attributes is identical. Hence, it follows the concept translation symmetry, and is formally defined as follows:

Definition 4.1 *A block r in the data space of a $\mathbb{U}$-dimensional table $\mathcal{T}$ is called symmetric along an attribute-set $\mathbb{A}$ 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 sets $\mathbb{A}_1, \mathbb{A}_2, ..., \mathbb{A}_\alpha$ iff

$$D(r) = D(\pi_{\mathbb{A}_1}(r)) \times D(\pi_{\mathbb{A}_2}(r)) \times ... \times D(\pi_{\mathbb{A}_\alpha}(r)) \times D(\pi_{\mathbb{U}\setminus(\mathbb{A}_1\cup\mathbb{A}_2\cup...\cup\mathbb{A}_\alpha)}(r)) \tag{4.1}$$

The Cartesian product implies that for a symmetric block, the data can be *independently* generated for each of the PAS. Hence, post-refinement, the different projection spaces can be processed independently, as shown in Figure 3.1.

4.1 Refinement Algorithm

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 the domain of $\mathbb{A}$ 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 constraints. 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 a refined block – the block is 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 4.1 *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 constraints. 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. For example, block b_4 in Example 1 has two projection subspaces along the Qty and Amt attributes, and refinement along either of these dimensions ensures symmetry along both 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 resultant PRBs, and usually more overlaps

of PRBs 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 with the maximum FB overlaps with b is chosen as the next subspace to be refined in the iterative sequence.

Impact of Conflicting Constraints. When partially overlapping PASs, say $\mathbb{A}_1$ and $\mathbb{A}_2$, are applied on an RB 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 Chapter 3.1, that the input workload is initially itself decomposed into non-conflicting sub-workloads.

Chapter 5

Projection Subspace Division

Projection subspace division aims at dividing $\mathbb{D}^{\mathbb{A}}$, the data subspace spanned by $\mathbb{A}$, into a collection $\mathbb{P}^{\mathbb{A}}$ of CPBs, where each element $p \in \mathbb{P}^{\mathbb{A}}$ is 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.

5.1 Valid Division

A valid division is defined as follows:

Definition 5.1 *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:*

$$\overline{r} = \bigcup_{p : pL^{\mathbb{A}}\overline{r}} p, \quad \forall\, \overline{r} \in \overline{\mathbb{R}}^{\mathbb{A}} \tag{5.1}$$

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

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

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 5.1 *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 5.1, 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:

$$p = \bigcap_{\overline{r}:(p,\overline{r})\in L} \overline{r} \setminus \bigcup_{\overline{r}':(p,\overline{r}')\notin L} \overline{r}' \tag{5.2}$$

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 5.1 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:

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

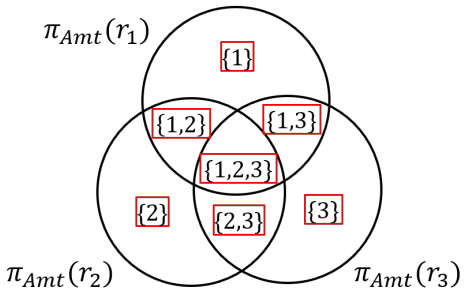


Figure 5.1: Partitioning in Projected Space

5.2 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 5.2 *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,

$$pL\overline{r} \implies p \subseteq \overline{r} \tag{5.3}$$

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 5.3, we can say that

$$\begin{aligned} p_1 \cap \overline{r} &= \emptyset, \quad \text{where } \overline{r} \in \overline{\mathbb{R}}(p_2) \setminus \overline{\mathbb{R}}(p_1) \\ p_2 \cap \overline{r} &= \emptyset, \quad \text{where } \overline{r} \in \overline{\mathbb{R}}(p_1) \setminus \overline{\mathbb{R}}(p_2) \end{aligned} \tag{5.4}$$

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 5.4, 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:

$$p = \bigcap_{\overline{r}: v_p(\overline{r})=1} \overline{r} \setminus \bigcup_{\overline{r}': v_p(\overline{r}')=0} \overline{r}' \tag{5.5}$$

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 5.2 (for simplicity, the elements of $\mathbb{V}$ are shown instead of $\mathbb{Q}$).

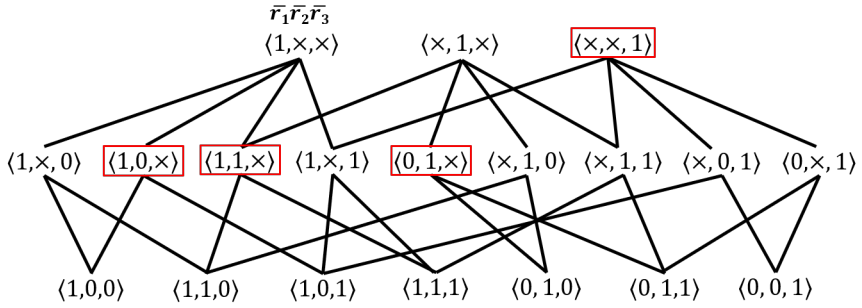


Figure 5.2: Hasse Diagram

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.

5.3 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 5.5. 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
      $\mathbb{V}^* \leftarrow \mathbb{V}^* \cup \{v\}$;
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 $\overline{\mathbb{R}}$.
- 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.

```
Function Split(v, pivot, targets, visited):
    splitSet ← ∅;
    for r̄ ∈ targets do
        if r̄ ∈ visited then
            v_r ← 0;
            remove r̄ from targets;
    if targets = ∅ then
        return v;
    powerset ← generate powerset enumeration of targets;
    for s ∈ powerset do
        new_v ← v;
        new_v_r ← 1, ∀r̄ ∈ s;
        new_v_r ← 0, ∀r̄ ∈ targets \ s;
        splitSet ← splitSet ∪ new_v_r;
    return splitSet;
```

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 5.3.

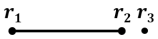


Figure 5.3: Example Division Graph

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 5.2). Using Equation 5.5, it yielded in 4 CPBs for Amt, $\mathbb{P}^* = \{p_1, p_2, p_3, p_4\}$ (as discussed in Chapter 3) and $L^* = \{(p_1, \overline{r}_1), (p_1, \overline{r}_2), (p_2, \overline{r}_1)$

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 5.1.

Table 5.1: 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)	$2^m - 1$

5.4 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 5.2 *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 5.3 `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 $p_2 L \overline{r}$,

$$\implies \exists c \in C \text{ such that } \overline{r} M c, \overline{r}' M c, \text{ for some } \overline{r}' \in \overline{\mathbb{R}} \text{ and}$$
$$(p_1, \overline{r}') \in L, (p_2, \overline{r}') \notin L \text{ (wlog) (using Lemma 5.1)}$$
$$\implies v \notin \overline{r}', \text{ otherwise there would exist } p_3 \in \mathbb{P} \text{ such that } v \in p_3;$$
$$p_2 \cap p_3 \neq \emptyset \text{ and } p_3 L \overline{r}' \text{ would imply Condition 2 violation.}$$
$$\implies \exists p'' \in \mathbb{P}' \text{ such that } p'' L' \overline{r}', u \in p'' \text{ and } v \notin p''.$$
$$Since, p' \cap p'' \neq \emptyset \text{ and } (p', c), (p'', c) \in M \circ L'$$
$$\text{Hence, contradiction (Condition 2 violation).}$$

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 5.2). 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. □

Chapter 6

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 $M \circ L^{\mathbb{A}}$.

$$|\pi_{\mathbb{A}}(\sigma_f(\mathcal{T}))| = \sum_{p:(p,c)\in M\circ L^{\mathbb{A}}} |p| \tag{6.1}$$

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.

$$|\overline{r}| = \sum_{p:pL^{\mathbb{A}}\overline{r}} |p| \tag{6.2}$$

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.

6.1 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

$$\sum_{r:rMc} x_r = l \tag{6.3}$$

(b) Distinct Row Cardinality Constraint (using Equation 6.1)

$$\sum_{p:(p,c)\in M\circ L^{\mathbb{A}}} y_p = k \tag{6.4}$$

6.2 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,

$$x_r \geq 0, \forall r \in \mathbb{R}, \quad \text{and} \quad y_p \geq 0, \forall p \in \mathbb{P}^{\mathbb{A}}, \text{ for all PAS } \mathbb{A} \tag{6.5}$$

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 6.2, these constraints, for each RB r and each of its associated PAS $\mathbb{A}$, are expressed as follows:

$$\sum_{p:pL^{\mathbb{A}}\overline{r}} y_p \leq x_r \tag{6.6}$$

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 $\mathbb{A}$:

$$x_r \leq |\mathcal{T}| \sum_{p:pL^{\mathbb{A}}\overline{r}} y_p \tag{6.7}$$

where $\overline{r} = \pi_{\mathbb{A}}(r)$ and $|\mathcal{T}|$ is the cardinality of $\mathcal{T}$. (Note that in general we can replace $|\mathcal{T}|$ with any sufficiently large positive integer).

We had already seen, in Chapter 3, 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	$y_1^{Amt} + y_2^{Amt} \leq x_1 \leq \lvert\mathcal{T}\rvert(y_1^{Amt} + y_2^{Amt})$
	$y_1^{Amt} + y_3^{Amt} \leq x_2 \leq \lvert\mathcal{T}\rvert(y_1^{Amt} + y_3^{Amt})$
	$y_4^{Amt} \leq x_3 \leq \lvert\mathcal{T}\rvert y_4^{Amt}$
	$y_1^{Qty} + y_3^{Qty} \leq x_2 \leq \lvert\mathcal{T}\rvert(y_1^{Qty} + y_3^{Qty})$
	$y_2^{Qty} + y_6^{Qty} \leq x_3 \leq \lvert\mathcal{T}\rvert(y_2^{Qty} + y_6^{Qty})$
	$y_3^{Qty} + y_4^{Qty} \leq x_{4a} \leq \lvert\mathcal{T}\rvert(y_3^{Qty} + y_4^{Qty})$
	$y_2^{Qty} + y_5^{Qty} \leq x_{4b} \leq \lvert\mathcal{T}\rvert(y_2^{Qty} + y_5^{Qty})$

6.3 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.

6.4 Workload Scalability and Robustness

Inspired by graphical model-based table decomposition techniques that were proposed in [5], 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 [17]. 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.

Chapter 7

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.

7.1 Summary Construction

The summary construction module compactly stores information needed for efficient tuple generation. It first assigns intervals to each of the populated CPBs from which data for the CPB is eventually generated. A challenge in interval assignment is that the domains of different CPBs may intersect. For example, in the running example, the domains of CPBs p_2^{Qty} and p_6^{Qty} intersect. However, since CPBs that are related to a common projection constraint should not intersect, we assign disjoint intervals to such 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, for PAS Amt, p_2^{Amt} and p_4^{Amt} are not related to any c in $\mathbb{C}$, and therefore their data generation intervals need not be disjoint. Finally, turning our attention to $\mathbb{A}_{left}$, that is, the attribute-set on which no projection is applied, the block boundaries are

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

Figure 7.1: Sample RB in Summary

kept as is and no distinct tuple count is maintained. Further, the number of distinct values can range anywhere from 1 to the block cardinality without violating the constraints.

With the above process, a possible interval assignment for the running example 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, as per the template structure shown in Figure 7.1. We see here that all the CPBs associated with the block, along with their distinct tuple cardinalities, are represented. 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 has a distinct row-cardinality. Lastly, each RB has an associated total cardinality. For a populated instance of the template, and its interpretation, we refer the reader to the example database summary previously shown in Figure 2.2.

7.2 Tuple Generation

Using the information in the summary, database tuples are instantiated. 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 has only a single interval, any partial-tuple within its boundaries can be picked for repetition. Finally, all partial-tuples of the RB are concatenated to construct its output tuples.

Inter-Block Dependencies. The CPBs are associated with a group of RBs through the relation L. 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 set of distinct points, equal to the cardinality, from the interval, and use this set in all the associated RBs.As a case in point, for the sample summary in Figure 2.2, the partial tuples generated for the CPB with interval $[20, 25)$ and distinct row cardinality 5 will be used to populate both r_3 and r_{4b}.

Chapter 8

Experiments

In this section, we evaluate the empirical performance of PiGen, which has been implemented in a Java tool incorporating the concepts presented in the previous sections. The popular Z3 solver [3] is invoked by the tool to compute the solutions for the LP formulations. Our experiments cover the accuracy, time and space overheads aspects of PiGen.

Database Environment The standard 1 TB TPC-DS [2] decision-support benchmark warehouse is used in our experiments. It is hosted on a PostgreSQL v9.6 engine [1], with a vanilla HP Z440 workstation serving as the hardware platform.

Workload Construction To construct the input workload, we first executed a large set of queries derived from the benchmark. The queries were chosen to cover both fact tables and dimension tables in the warehouse. Here, we report on the four (denormalized) tables that were subject to the maximum number of projection operations, namely, the STORE_SALES (SS), CATALOG_SALES (CS) and WEB_SALES (WS) INVENTORY (INV) tables.

We created two workloads, $\mathbb{C}$ and $\mathbb{W}$, of projection-inclusive constraints spread over these four tables. $\mathbb{C}$ has all mutually compatible constraints, while $\mathbb{W}$ is a superset of $\mathbb{C}$, featuring additional constraints that result in conflicts. The distribution of constraints over the four tables for these two workloads is enumerated in Table 8.1.

Table 8.1: Distribution of Constraints

Table	**# Constraints ($\mathbb{C}$)**	**# Constraints ($\mathbb{W}$)**
SS	16	52
CS	15	28
WS	16	29
INV	6	8

In presenting the experimental results, we initially focus on the compatible workload $\mathbb{C}$. Subsequently, using $\mathbb{W}$, we discuss the corresponding performance for workloads featuring conflicts.

8.1 Accuracy

The constraints in the $\mathbb{C}$ workload cover a wide variety of complexities, with their cardinalities varying from a few rows to several million rows. Further, the PAS lengths vary from one to six. A representative sample constraint from the workload is:

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

applied on the denormalized relation of STORE_SALES where

$\mathbb{A} : \{i\_category, i\_brand, s\_store\_name, s\_company\_name, d\_moy\}$ and

$\mathbf{f} : d\_year = 2002 \ \wedge$

$(i\_category \in$ ('Jewelry','Women') $\wedge \ i\_class \in$ ('mens watch','dresses')) $\vee$

$(i\_category \in$ ('Men','Sports') $\wedge \ i\_class \in$ ('sports-apparel','sailing'))

When PiGen was run on these inputs, the generated data satisfied all the constraints with **100%** accuracy – this was explicitly confirmed by running the original queries on the synthetic database and monitoring the operator outputs.

8.2 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 the four summary tables are reported in Table 8.2. We see here that the time to produce the summary is 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.

Table 8.2: Overheads

Table	Summary Time	Summary Size
SS	21 min	58 kB
CS	32 min	117 kB
WS	15 min	64 kB
INV	2 sec	13 kB

Table 8.3: Block Profiles

Table	FB	RB	CPB
SS	74	88	132662
CS	139	141	165936
WS	119	132	73929
INV	11	16	41

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 four 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 8.3 – 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.

Note that the time and space overheads incurred are intrinsically *data-scale-free*, i.e., they do not depend on the generated size. We explicitly verified this property by running the PiGen algorithm over 10 GB and 100 GB versions of the TPC-DS database.

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 8.4, and we see that even a huge table such as SS, having close to 3 billion records, is generated within a few minutes.

Table 8.4: 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

8.3 Workload Decomposition

We now turn our attention to conflicting workloads, which require the pre-processing step of workload decomposition. 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.

8.4 Instance-based Decomposition (ID)

Here the decomposition algorithm uses Definition 3.1 of a conflicting pair, and for this framework, the number of workloads obtained for the four tables are shown in Table 8.5. We observe that despite using an approximate vertex coloring algorithm (Chapter 3.1), 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 8.2. 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 8.5: 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

8.5 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 8.6 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.

Table 8.6: 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

Chapter 9

Related Work

Over the past three decades, a variety of novel approaches have been proposed for synthetic database generation. The initial efforts (e.g. [12, 10]) 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 [20, 16, 4]. In more recent times, generation techniques driven by constraints on query outputs have been analyzed. A particularly potent effort in this class was **RQP** [7], 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 [8, 15, 5, 18, 17, 14, 11] fall in this category. Among these, **QAGen** [8], **MyBenchmark** [15] and **TouchStone** [14] take parameterized constraints as input, i.e. 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. A stricter notion of constraints was considered in [5, 18, 17, 11], where instantiated constraints in the form of cardinality constraints are given as part of the input itself, and the generated data is expected to conform to them. While these techniques handled constraints with filter and join operators satisfactorily, their support for the projection operator was stylized to special cases. For instance, **DataSynth** [5] proposed a projection generator that catered to individual columns. In contrast, in PiGen, we consider a general class of projection-inclusive constraints that bring in a host of new technical challenges, as outlined in the Introduction.

Complementary to the studies by the database community, the mathematical literature includes work such as [9, 22, 13], 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 [9], 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 [22]. 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 [13]. 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.

Chapter 10

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 union cardinality, projection subspace dependencies and block dependencies. 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. Finally, block dependencies were catered to through deterministic tuple instantiation techniques. The experimental evaluation on a TPC-DS platform 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 queries outside of the current workload.

Bibliography

[1] PostgreSQL. https://www.postgresql.org/docs/9.6

[2] TPC-DS. http://www.tpc.org/tpcds/

[3] Z3. https://github.com/Z3Prover/z3

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

[5] A. Arasu, R. Kaushik, and J. Li. Data Generation using Declarative Constraints. In *ACM SIGMOD Conf.*, 2011.

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

[7] C. Binnig, D. Kossmann, and E. Lo. Reverse Query Processing. In *23rd ICDE Conf.*, 2007.

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

[9] B. Bollobás and A. Thomason. Projections of Bodies and Hereditary Properties of Hypergraphs. *Bulletin of the London Mathematical Society*, 27 (1995).

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

[11] A. Gilad, S. Patwa, and A. Machanavajjhala. Synthesizing Linked Data Under Cardinality and Integrity Constraints In *ACM SIGMOD Conf.*, 2021.

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

[13] I. Leader, Z. Randelovic, Eero Raty. Inequalities on Projected Volumes. *arXiv:1909.12858*

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

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

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

[17] A. Sanghi, Rajkumar S., and J. R. Haritsa. Towards Generating HiFi Databases. In *26th DASFAA Conf.*, 2021.

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

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

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

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

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