HYDRA: A Dynamic Approach to Database Regeneration

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Anupam Sanghi

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

December, 2022

Declaration of Originality

I, **Anupam Sanghi**, with SR No. **04-04-00-10-12-17-1-15034** hereby declare that the material presented in the thesis titled

HYDRA: A Dynamic Approach to Database Regeneration

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2017-2022**.
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: 15/12/2022 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 HARITSA Advisor Signature

DEDICATED TO

My Parents

for their unconditional support

Acknowledgements

I would like to express my heartfelt gratitude to my advisor, Prof. Jayant R. Haritsa, for introducing me to the world of research and providing me with an extraordinary learning experience. He has been a constant source of inspiration to me, not only as a passionate researcher but also as a person who embodies humility. I am fortunate to have had the opportunity to work with him during my M.E. and PhD. Our eight years of association, filled with lengthy technical and non-technical discussions, have guided me in ways I could not have imagined. His unparalleled ability to tell a story with a dash of humor is something I will cherish the most.

I am also deeply grateful to Prof. Srikanta Tirthapura from Iowa State University. His guidance has taught me how to approach and solve problems progressively. I am grateful for his constructive feedback, which has been essential in shaping this thesis.

I am extremely thankful to all my collaborators - Shadab Ahmed, Raghav Sood, Rajkumar S., Tarun Patel, Prashik Rawale, Dharmendra Singh, Subhodeep Maji, and Manish Jayswal - for an extremely enjoyable and fulfilling experience. Hydra is incomplete without all of its heads, and I thank each of them for their valuable contributions.

I sincerely thank my thesis examiners, Prof. S. Sudarshan from IIT Bombay and Dr. Manuel Rigger from NUS, for their detailed and thoughtful review. I am also grateful to my comprehensive committee – Dr. Satish Govindarajan, Prof. Shalabh Bhatnagar, Prof. Srikanth Iyer, and Dr. Kunal Chaudhury, for their extremely helpful feedback and suggestions at various stages of my PhD journey. I am also deeply thankful to Dr. Prasad Deshpande (Google) for his invaluable guidance and support. I wish to extend a special thanks to the supportive faculty at the Dept. of Computer Science and Automation (CSA), including Prof. Chiranjib Bhattacharyya, Prof. L. Sunil Chandran, Prof. Shirish Shevade, Dr. Rahul Saladi, and Dr. Bhavana Kanukurthi, for their guidance and encouragement during the course.

I have been fortunate to have gotten a chance to present my work at various forums and receive insightful suggestions. Among these, I would like to especially thank the attendees of the Dagstuhl seminar 21442, including Prof. Hannes Mühleisen (CWI Amsterdam), Dr. Danica Porobic (Oracle), and Dr. Alexander Böhm (Google) for providing me with an industrial

perspective on my work. I am also deeply thankful to Dr. Diptikalyan Saha and the AI team at IBM Research India and Dr. Karthik Ramachandra and the SQL team at Microsoft India for providing the opportunity to present my work and receive their invaluable feedback.

I would like to extend a special thanks to Dr. Arvind Arasu (Microsoft) and Prof. Jian Li (Tsinghua Univ.) for their guidance during my initial exploration of the previous research in this field. I also thank Prof. Ashwin Machanavajjhala and Shweta Patwa from Duke Univ. for providing useful benchmarks for experimental evaluations. I am also grateful to Ashoke S., Kalyan S., Rajeev Rastogi, and Prasanna V., my colleagues during my one year at Huawei Technologies. Their guidance was immensely useful in building the foundation of Hydra.

I cannot be thankful enough to my labmates at the Database Systems Lab at IISc, the DSLites. They created a welcoming and supportive environment that encouraged both productivity and camaraderie, and I will greatly miss working with them. In addition to my collaborators, I was fortunate to have Dr. Anshuman Dutt, Dr. Srinivas Karthik, Manish Kesarwani, and Shivani Tripathi as strong pillars, who were always there to offer support during any technical or non-technical difficulty. I have many fond memories of the fun sessions with Rafia and Kuntal, as well as the numerous discussions and outings with Sanket, Gourav, and Urvashi. This list would be incomplete without mentioning Kapil, Dhrumil, Vishal, and Santhosh, who supported me during the toughest stages of my journey.

I am immensely grateful to IBM Research for providing me with a PhD fellowship, which was incredibly helpful in supporting my research. I would also like to sincerely thank Microsoft Research for their generous travel support, which enabled me to attend top international database conferences. In addition, I would like to acknowledge ACM India and the institute's GARP sponsorship for aiding my travel. Without their support, it would not have been possible for me to share my research with the wider community. I am also extremely thankful to the office staff of the CSA Dept. who have generously helped me throughout the M.E. and PhD courses.

My experience at IISc was truly special, thanks to the amazing friends I met here. My masters' gang - Divya, Jay, Raman, Lucky, Parita, Kuntal, Nitesh, Anshu, and Geetanjali - have been like a second family to me for the past eight years. I would also like to express my love and gratitude to Anwesha, Sruthi, Vishal, Nimisha, Rahul, Shweta, and Anand for making my PhD life joyful and providing me with enough memories to cherish for a lifetime. I am also grateful to Shadab, Hemanta, Subhodeep, Sakya, and Akshay, who made my stay at IISc memorable and lively. I have been fortunate enough to have so many friends who have been a part of different stages of my life, including Rupali, Riya, Yougansh (sir), and Ashish who are always available to listen to my rants.

I am blessed to have been taught by some amazing teachers throughout my academic journey,

including those at Air Force School, Gwalior, and Jaypee Institute of Information Technology, NOIDA. Their guidance and encouragement have had a lasting impact on me.

Lastly, but most importantly, I am deeply thankful to my wonderful family, who have been my greatest support and source of inspiration. My biggest well-wisher, my grandmother, has always blessed me with affection. My elder sister, Richa, is my first teacher, and I am extremely grateful to her for giving me a strong foundation. My sister Shruti is my first friend, and I thank her for being my biggest cheerleader. My parents have been unwavering in their support and mean the world to me. I dedicate this thesis to them.

Abstract

Database software vendors often need to generate synthetic databases for a variety of applications, including (a) Testing 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 toward 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.

Several data generation frameworks have been proposed for OLAP over the past three decades. The early efforts focused on ab initio generation based on standard mathematical distributions. Subsequently, there was a shift to database-dependent regeneration, which aims to create a database with similar statistical properties to a specific client database. However, these mechanisms could not mimic the customer query-processing environments satisfactorily. The contemporary school of thought generates workload-aware data that uses query execution plans from the customer workloads as input and guarantees volumetric similarity. That is, the intermediate row cardinalities obtained at the client and vendor sites are very similar when matching query plans are executed. This similarity helps to preserve the multi-dimensional layout and flow of the data, a prerequisite for achieving similar performance on the client's workload. However, even in this category, the existing frameworks are hampered by limitations such as the inability to (a) provide a comprehensive algorithm to handle the queries based on core relational algebra operators, namely, Select, Project, and Join; (b) scale to big data volumes; (c) scale to large input workloads; and (d) provide high accuracy on unseen queries.

In this work, motivated by the above lacunae, we present HYDRA, a data regeneration tool that materially addresses the above challenges by adding functionality, dynamism, scale, and robustness. Firstly, extended workload coverage is provided through a comprehensive solution for modeling select-project-join relational algebra operators. Specifically, the constraints are represented as a linear feasibility problem, in which each variable represents the volume of a partitioned region of the data space. Our partitioning scheme for filter constraints permits the

regions to be non-convex and ensures the minimum number of regions, thereby hugely reducing the problem complexity as compared to the rectangular grid-partitioning advocated in the prior literature. Similarly, our projection subspace division and projection isolation strategies address the critical challenge of capturing unions, as opposed to summations, in incorporating projection constraints. Finally, by creating referential constraints over denormalized equivalents of the tables, Hydra delivers a comprehensive solution that also handles join constraints.

Secondly, a unique feature of our data regeneration approach is that it delivers a database summary as the output rather than the static data itself. This summary is of negligible size and depends only on the query workload and not on the database scale. It can be used for dynamically generating data during query execution. Therefore, the enormous time and space overheads incurred by prior techniques in generating and storing the data before initiating analysis are eliminated. Our experience is that the summaries for complex Big Data client scenarios comprising over a hundred queries are constructed within just a few minutes, requiring only a few MBs of storage.

Thirdly, to improve accuracy towards unseen queries, Hydra additionally exploits metadata statistics maintained by the database engine. Specifically, it adds an objective function to the linear program to pick a solution with improved inter-region tuple distribution. Further, a uniform distribution of tuples within regions is modeled to obtain a spread of values. These techniques facilitate the careful selection of a desirable database from the candidate synthetic databases, and also provide metadata compliance.

The proposed ideas have been evaluated on the TPC-DS synthetic benchmark, as well as real-world benchmarks based on the Census and IMDB databases. Further, the Hydra framework has been prototyped in a Java-based tool that provides a visual and interactive demonstration of the data regeneration pipeline. The tool has been warmly received by both academic and industrial communities.

Publications based on this Thesis

- Projection-Compliant Database Generation
  Anupam Sanghi, Shadab Ahmed, and Jayant Haritsa
  Proc. of 48th Intl. Conf. on Very Large Data Bases (VLDB), Sydney, Australia, September 2022
  published as PVLDB Journal, 15(5), January 2022

- Towards Generating HiFi Databases
  Anupam Sanghi, Rajkumar S., and Jayant Haritsa
  Proc. of 26th Intl. Conf. on Database Systems for Advanced Applications (DASFAA), Taipei, Taiwan, April 2021

- HYDRA: A Dynamic Big Data Regenerator
  Anupam Sanghi, Raghav Sood, Dharmendra Singh, Jayant Haritsa, and Srikanta Tirthapura
  Proc. of 44th Intl. Conf. on Very Large Data Bases (VLDB), Rio de Janeiro, Brazil, August 2018
  published as PVLDB Journal, 11(12), August 2018

- Scalable and Dynamic Regeneration of Big Data Volumes
  Anupam Sanghi, Raghav Sood, Jayant Haritsa, and Srikanta Tirthapura
  Proc. of 21st Intl. Conf. on Extending DataBase Technology (EDBT), Vienna, Austria, March 2018

Reports

- Data Generation using Join Constraints
  Anupam Sanghi, Shadab Ahmed, Prashik Rawale, and Jayant Haritsa
  Tech. Report TR-2022-01, DSL/CDS, IISc, 2022, dsl.cds.iisc.ac.in/publications/report/TR/TR-2022-01.pdf

Publications based on this Thesis

- Data Generation using Projection Constraints
  Anupam Sanghi, Shadab Ahmed, and Jayant Haritsa
  Tech. Report TR-2021-03, DSL/CDS, IISc, 2021, dsl.cds.iisc.ac.in/publications/report/TR/TR-2021-03.pdf

- High Fidelity Database Generators
  Anupam Sanghi, Rajkumar S., and Jayant Haritsa
  Tech. Report TR-2021-01, DSL/CDS, IISc, 2021, dsl.cds.iisc.ac.in/publications/report/TR/TR-2021-01.pdf

- Scalable and Dynamic Workload Dependent Data Regeneration
  Anupam Sanghi, Raghav Sood, Jayant Haritsa, and Srikanta Tirthapura
  Tech. Report TR-2017-01, DSL/CDS, IISc, 2021, dsl.cds.iisc.ac.in/publications/report/TR/TR-2017-01.pdf

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In industrial practice, a common requirement for enterprise database vendors is to adequately test their database engines. The enormous size and complexity of these software make the testing process challenging. The importance of addressing these challenges is also showcased by a recent Dagstuhl seminar [3].

The testing process commonly requires generating synthetic databases and query workloads. Traditionally, the vendors have relied on using synthetic benchmarks such as TPC-H [11] and TPC-DS [12]. However, these benchmarks cover only a limited input testing space [25], due to lack of flexibility with respect to the schema, queries, and data distribution.

Several database generators have been proposed from both industry and academia (e.g. MUDD [68], PSDG [40], PDGF [62], Myriad [16]) that do an ab initio data generation. However, here as well, there is limited control on the input/output of the intermediate operators of a query during a test, which can be an important requirement while testing individual DBMS components. As a result, the client applications may reflect unforeseen problems. Therefore, vendors need a way to generate representative data and workloads that accurately mimic the data processing environments at customer deployments. While, in principle, clients could transfer their original data and workloads to the vendor for the intended evaluation purposes, this is often infeasible due to privacy and liability concerns. Moreover, even if a client is willing to share the data, transferring and storing the data at the vendor's site may have impractical space and time overheads, especially in this Big Data era. Therefore, an important requirement is to be able to dynamically regenerate representative databases while executing queries during the testing process.

Several frameworks have been proposed in the last two decades for generating client-specific synthetic databases. These frameworks can be classified into two main classes: (a) Database-Dependent Regeneration, and (b) Query-Dependent Regeneration. We will discuss these classes

next.

Database-Dependent Regeneration

The frameworks in this category (e.g. DBSynth [61], RSGen [66], DScaler [82]) derive coarse statistics from the original data and ensure the synthetic data also obeys these statistics. For example, RSGen generates data using one-dimensional histograms with respect to various columns in the database. The key limitation here is that the accuracy is restricted to simple queries, typically dealing with single attribute filters.

More recently, a new breed of generators in this category has emerged that makes use of machine learning models such as auto-encoders and generative adversarial networks. However, these models are focused on synthesizing a single table [36]. Therefore, evaluation of even simple queries involving joins between multiple tables cannot be satisfactorily performed.

Query-Dependent Regeneration

Here the objective is to ensure that the generated data ensures *volumetric similarity* for a given set of input queries. That is, assuming a common choice of query execution plans at the client and vendor sites, the output row cardinalities of individual operators in these plans are very similar in the original and synthetic databases. This similarity helps to preserve the multi-dimensional layout and flow of the data, a prerequisite for achieving similar performance on the client's workload. There are two sub-classes in this category based on the nature of constraints:

Parameterized Constraints: Here the queries are parameterized with regard to the constants in the query, and the generators (e.g. MyBenchmark [55], TouchStone [52]) additionally also provide parameter values for which the intermediate cardinalities are matched.

Value Constraints: Here predicate constants are pre-specified in the input as part of the workload. The generators (e.g. DataSynth [17], Linked-Data [37]) ensure that the intermediate cardinalities are matched for these constants.

Consider the SQL query shown in Figure 1.1(b). Here the predicate constants are specified in the query and therefore result in value constraints. The parameterized equivalent of this query is as follows:

```
Select Age, Average (Score) From Register Reg, Student Std
Where Reg.RollNo = Std.RollNo and Year = param1 and Age >= param2
```

In the above query, *param1* and *param2* are variables.

While the parameterized constraints can be preferable from a privacy perspective, value constraints allow construction of databases that are closer to the original and, therefore, more robust to unseen queries. The existing frameworks across both categories suffer from one or more of the limitations, such as the inability to (a) provide a comprehensive algorithm to handle queries based on core relational algebra operators, namely, select, project, and join; (b) scale to big data volumes; (c) scale to large input workloads; and (d) provide high accuracy on unseen queries.

In this thesis, motivated by the above lacunae, we present **HYDRA**, a new (value) query-dependent data generation framework, that materially addresses the above challenges by adding functionality, dynamism, scale, and robustness. The extended workload coverage is obtained by providing a comprehensive solution to support SPJ queries. At its core, the constraints are modeled using linear feasibility problem, where each variable represents the volume of a region of the data space. These regions are computed using a series of partitioning strategies. Hydra further exploits metadata statistics maintained by the database engine in order to improve accuracy towards unseen queries.

A unique feature of our data regeneration approach is that it delivers a database summary as the output rather than the static data itself. This summary is of negligible size and depends only on the query workload and not on the database scale. It can be used for dynamically generating data during query execution. Therefore, the enormous time and space overheads incurred by prior techniques in generating and storing data before initiating analysis are eliminated. Specifically, the summaries for complex Big Data client scenarios comprising over a hundred queries are constructed within just a few minutes, requiring only a few MBs of storage.

As a proof of concept, the Hydra framework has been prototyped in a Java based-tool and evaluated on both synthetic and real-world benchmarks. Our work is closest to DataSynth [17, 18] tool from Microsoft. DataSynth also expresses the value constraints in the form of linear feasibility program whose solution is used to construct the synthetic database. We also show a comparison of Hydra with DataSynth on various facets in our experimental evaluation.

1.1 Volumetric Similarity

Synthetic databases have been targeted toward 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. The contemporary school of thought [25, 55, 17, 52, 37, 80] is generating query-dependent databases that use query execution plans from the customer workloads

as input and provide volumetric similarity. That is, the intermediate row-cardinalities obtained at the client and vendor sites are very similar when matching query plans[1] are executed. This similarity helps to preserve the multi-dimensional layout and flow of the data, a prerequisite for achieving similar performance on the client's workload. Therefore, in this thesis, we have primarily focused on data synthesis that achieves volumetric similarity. Extensions to other important metrics such as Duplication Distribution and Presortedness in Chapter 9.

Now, we will discuss the preliminaries of volumetric similarity. Subsequently, we will present some of its prominent applications.

1.1.1 Preliminaries

We start with background information on the key foundations – Annotated Query Plans [25] and Cardinality Constraints [17] – that form the basis for achieving volumetric similarity, and are therefore relevant to our design of the Hydra system.

Annotated Query Plans. Consider a toy university database with the schema shown in Figure 1.1(a). A sample client query on this schema is shown in Figure 1.1(b), with the corresponding query execution plan in Figure 1.1(c). Note that this execution plan has the output edge of each operator annotated with the associated row cardinality[2] in green color – for instance, there are 3000 rows resulting from the join of filtered Register and Student tables and 10 rows qualify after the final projection as a result of Group By clause. Such a plan is referred to as an "Annotated Query Plan" (AQP) in [25].

We wish to highlight that in the context of volumetric similarity, Group By clause behaves same as Distinct SQL operation. Distinct is expressed as duplicate eliminating projection relational algebra operation and therefore we have used projection as an umbrella term to include Group By as well.

Cardinality Constraints. A unified and declarative mechanism for representing AQP data characteristics, called *cardinality constraints* (CCs), was proposed in [17]. The canonical representation of the constraint 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* (PAS), i.e. the set of attributes on

[1] ensured through "plan forcing" [13] or "metadata matching" [73]

[2] These are the actual row cardinalities as evaluated during the client's execution. Using compile-time estimates instead can lead to a qualitatively poor synthetic database, since the estimates are routinely highly erroneous, as emphasized in the query optimization literature.

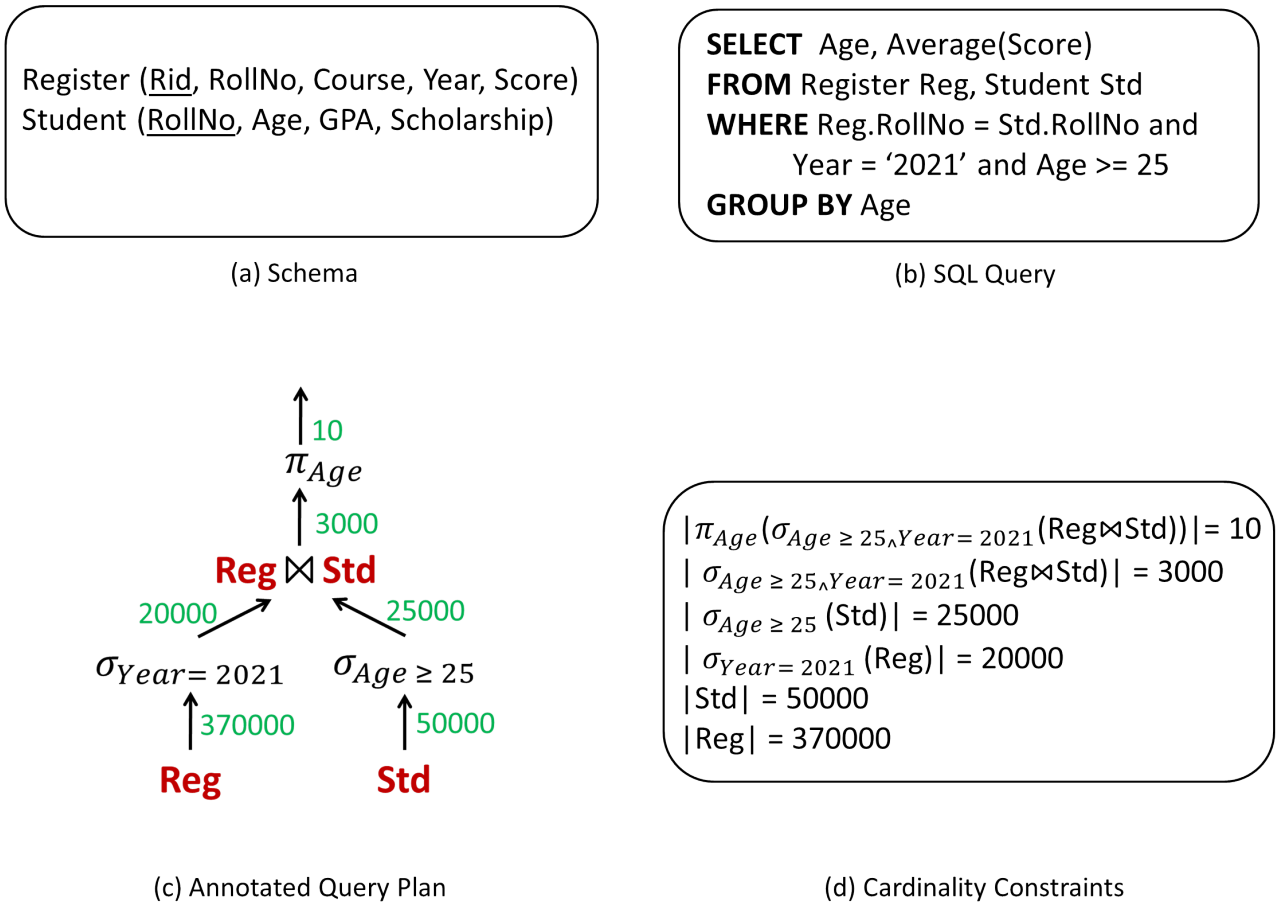


Figure 1.1: Example Scenario

which the projection is applied; and k is a count representing the output row-cardinality of the relational expression. For example, the CCs expressing the AQP of Figure 1.1(c) are shown in Figure 1.1(d).

To ensure volumetric similarity, the schematic information and the set of AQPs with respect to all the queries in the workload are taken as input from the client site. The synthetic data produced is aimed at closely meeting the CCs derived from the AQPs.

1.1.2 Applications

By ensuring volumetric similarity, Hydra can be beneficial for a variety of use-cases. We enumerate some of these next.

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 a hash operator is predicated on accurate modeling of input row cardinality; and (c) Given an operator of interest, evaluating its impact on the performance of downstream operations. For instance, 12 out of the 22 queries in the TPC-H benchmark require a sort operation immediately following a projection-based operator. Therefore, in these cases, the projection output cardinality affects the performance of sort operator.

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 mimicking intermediate row cardinalities can be of particular utility since cardinality estimation techniques are known to have difficulty in accurately modeling of operator outputs, especially for the higher operators in the plan tree. For instance, knowledge of left and right intermediate table cardinalities can be used by the DBA to choose the better plan between nested loops vs hash join based plans and subsequently force it.

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 database synthesized using cardinality constraints is a viable option. 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, modeling this as a CC and ensuring synthetic data satisfies it can be useful.

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. Volumetric similarity helps in achieving greater fidelity to the application's framework.

1.2 Summary of Contributions

In this thesis, we present HYDRA, a data regeneration tool that materially advances state of the art on a variety of facets such as functionality, dynamism, scale, and robustness. Specifically, the key contributions of Hydra are enumerated below.

Extended Workload Coverage. Hydra provides a comprehensive solution to support queries based on SPJ relational algebra operators. Specifically, the constraints are modeled using linear feasibility problem (LP)[1], in which each variable represents the volume of a region of the data space. These regions of data space are computed using a series of partitioning strategies. For example, to encode the filter constraints, our *Region Partitioning* approach divides the data space into the provably minimum number of regions. Our *projection subspace division* and *projection isolation* strategies address the critical challenges in incorporating projection constraints. By modeling *referential constraints* over denormalized equivalents of the tables, Hydra delivers a comprehensive solution that also additionally handles join constraints.

Apart from the scope of queries, by ensuring minimum region count in the output of the partitioning algorithm, Hydra covers a larger number of input queries. As a case in point, on comparing the Region Partitioning strategy of Hydra against the more fine-grained *Grid Partitioning* approach used in DataSynth, the former reduces the LP complexity by many orders of magnitude.For instance, an LP with a few *thousand* variables in Hydra requires more than a *billion* variables in DataSynth– in fact, in this case, the LP is solved in a less than a minute on the Hydra formulation, while the solver crashes on the DataSynth formulation.

Database Summary and Dynamic Database Regeneration. Hydra introduces the concept of dynamic database regeneration by constructing a minuscule database summary that can on-the-fly regenerate databases of arbitrary size during query execution. This approach is imperative for Big Data systems, where working with materialized solutions entails impractical time and space overheads. Specifically, dynamic generation eliminates the need to store data on the disk and its subsequent load by the engine – instead, all data is created and delivered on demand. An orthogonal benefit is that the generation rate can be strictly controlled, thereby addressing *velocity*, one of the several V's associated with Big Data [2].

[1]LP term has been used to be consistent with terminology used in literature.

Data Accuracy, Scalability and Efficiency. Hydra provides perfect volumetric similarity on the input query workload, a property not seen in the prior work. Moreover, Hydra does so without compromising on the efficiency of the generation pipeline. In fact, the algorithm proposed for database summary production is *data-scale-free*. That is, it depends only on the query workload and not on the database scale. Specifically, the summaries for complex Big Data client scenarios comprising over a hundred queries are constructed within just a few minutes, on a vanilla computing platform, requiring only a few MBs of storage.

Improved Robustness. Hydra provides a mechanism that is expected to provide better accuracy on *unseen* queries – the mechanism leverages the metadata statistics maintained by the database engine. Specifically, it adds an *objective function* to the linear program to pick a solution with improved inter-region tuple distribution. Further, a uniform distribution of tuples within regions is generated to get a spread of values. In a nutshell, these techniques facilitate careful selection of a more representative database from the candidate synthetic databases and also provide *metadata compliance*.

Enhanced Evaluation. We have evaluated the proposed ideas using both synthetic benchmarks, such as TPC-DS, and real-world benchmarks based on Census (from [37]) and IMDB databases [45, 5]. Therefore, our evaluation is more comprehensive than that in prior techniques, which have largely been evaluated on simpler and small-sized query workloads operating on modest databases.

Prototype Tool. As a proof of concept, the Hydra framework has been prototyped in a Java based-tool, running to over 50K lines of code, and is currently operational on the PostgreSQL v9.6 engine [7]. It has an intuitive user interface that facilitates modeling of enterprise database environments, delivers feedback on the regenerated data, and tabulates performance reports on the regeneration quality. The entire tool, including the source, is downloadable [1]. The tool has been warmly received by both the academic and industrial communities.

1.3 Thesis Overview

In this section we present an overview of the thesis with a brief description of the various chapters included. The flow of information from the client to vendor is shown in Figure 1.2. The client ships the schema information, and the query workload with its corresponding AQPs

[1] https://dsl.cds.iisc.ac.in/projects/HYDRA/index.html

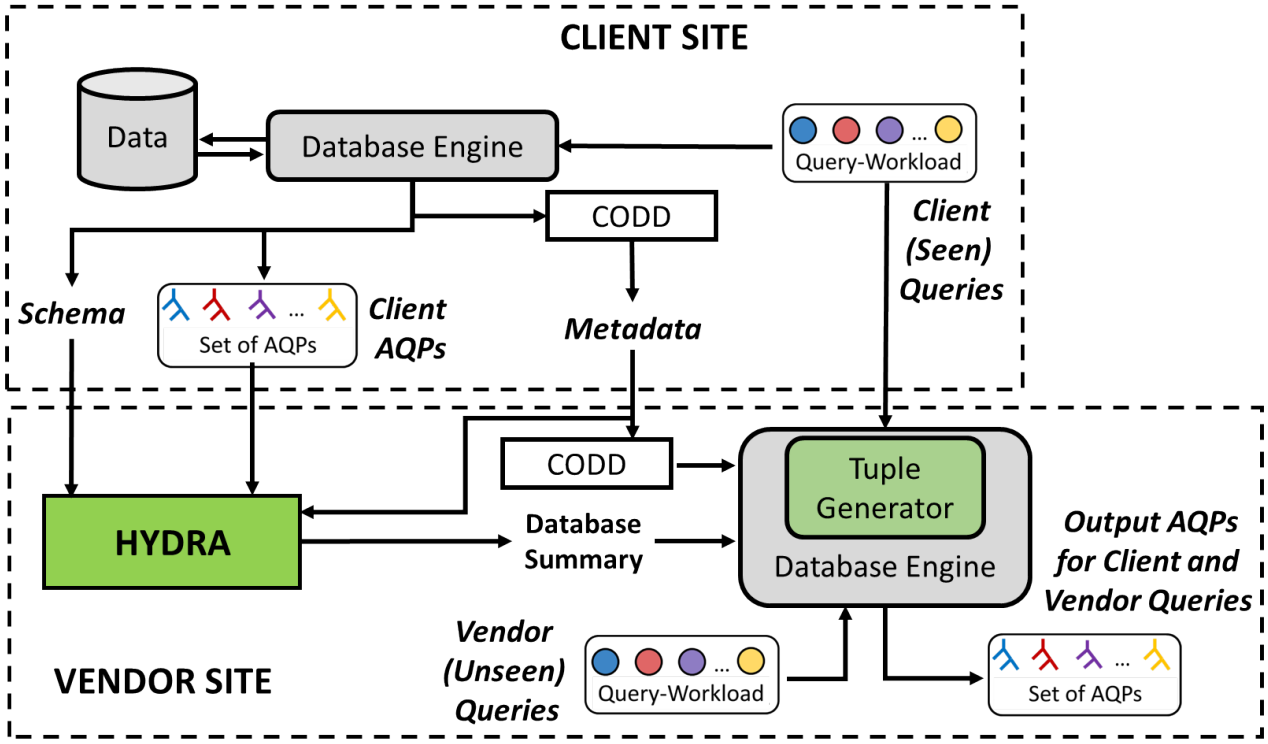


Figure 1.2: Data Flow

obtained from the database engine. The statistical metadata from the database catalogs is captured with the help of CODD metadata processor tool [19] tool. Using this information, Hydra generates synthetic data at the vendor cite.

As discussed earlier, at its core Hydra expresses the volumetric constraints using LP. Each variable in the LP accounts for the cardinality of a specific region of the data space. We use an LP solver to obtain a feasible solution to the problem. [1]

Using LP solution, Hydra produces a collection of database summaries. Database summary is used to produce data on demand during query execution using the Tuple Generator module that is implemented inside the database engine. The produced data ensures volumetric similarity on the client queries. In addition to this, the vendor can also evaluate unseen queries on the produced synthetic database.

We shall now present the organization of the thesis followed by a summary of the core technical chapters.

[1] An alternate strategy could have been to construct an SQL query for each region and have another iteration of client interaction where the client provides the actual values for these queries. While this strategy has the potential to produce more robust databases, it requires significant extra work from the client-side and may not be always desirable from a privacy point of view.

1.3.1 Organization

The thesis Introduction and Related Work are presented in Chapters 1 and 2, respectively. The Problem Framework is presented in Chapter 3. The mechanism for synthesizing data using the input CCs is explained progressively in Chapters 4 through 6. Specifically, we begin by discussing the single table synthesis method, where initially we assume only filter operators in the CCs in Chapter 4, and subsequently include projection operator in Chapter 5. Further, Chapter 6 discusses multiple table synthesis, which includes join operators in the CCs. Next, the discussion on improving robustness to unseen queries is presented in Chapter 7. The complete implementation and prototype details are provided in Chapter 8. We discuss the extensions of Hydra in Chapter 9 and finally, conclude in Chapter 10.

1.3.2 Problem Framework (Chapter 3)

We summarize the basic problem statement, assumptions made and the output delivered.

Statement. Given an SPJ query-workload $\mathcal{W}$, with its corresponding set of AQPs $\mathcal{Q}$, derived from an original database with schema $\mathcal{S}$ and statistical metadata $\mathcal{M}$, the objective is to generate a synthetic database $\mathcal{D}$ such that it conforms to $\mathcal{S}$ and $\mathcal{Q}$. That is, the AQPs obtained from the original database match, wrt the cardinality annotations, the AQPs obtained on $\mathcal{D}$.

Assumption. We assume that $\mathcal{W}$ comprises of only PK-FK joins. Further, we assume that the filters and projections are applied only on non-key columns. For simplicity, we assume that $\mathcal{Q}$ is collectively feasible, that is, there exists at least one legal database instance conforming to $\mathcal{Q}$. In the regeneration usecase, this assumption holds trivially since the constraints are produced from an original client database.

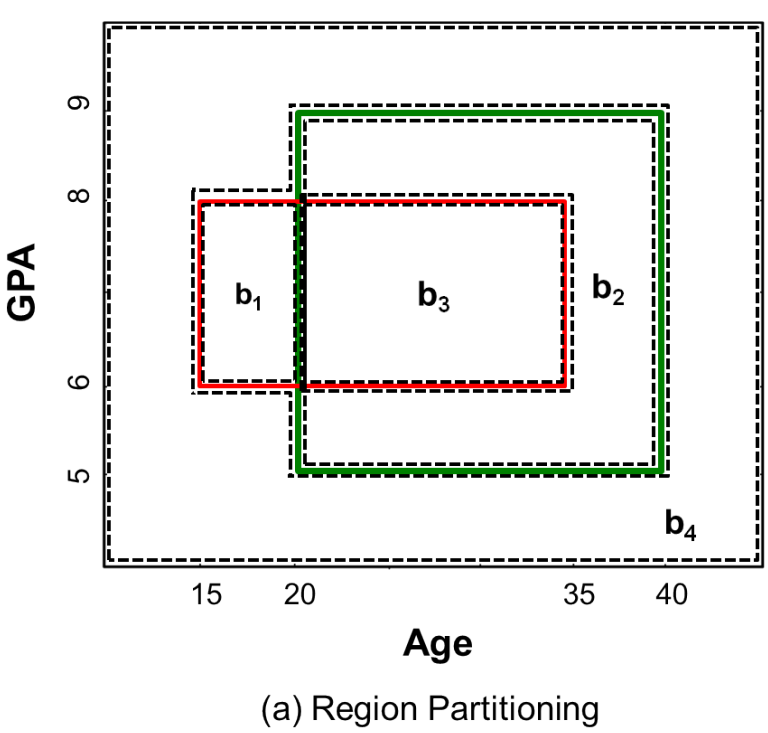
Output. Given $\mathcal{S}$, $\mathcal{M}$, $\mathcal{W}$ and $\mathcal{Q}$, Hydra outputs a collection of database summaries $\mathbb{S}$. Each summary $s^{\mathcal{D}} \in \mathbb{S}$ can be used to deterministically produce the associated database $\mathcal{D}$. The databases produced are such that: (a) all of them conform to $\mathcal{S}$, and (b) for each query in $\mathcal{W}$, its corresponding AQP in $\mathcal{Q}$ matches with the AQP obtained on at least one output database instance.

In principle, we would like to have a single summary that represents the entire workload. However, in order to handle the problem of conflicting projections (Section 1.3.4), we are forced to take recourse to multiple summaries.

1.3.3 Filter Constraints (Chapter 4)

To model filter constraints, the data space of the target table is logically partitioned into a set of *Filter-Blocks*. Each filter-block satisfies the condition that every data point in it satisfies the

same subset of filter predicates. The row cardinality of each filter-block is represented using a variable, and is then used to construct a linear feasibility problem. The resultant system is usually highly under-determined and therefore, to reduce the complexity of solving it, our proposed Region Partitioning technique partitions the data space into the minimum number of filter-blocks.



(a) Region Partitioning

$$x_1 + x_3 = 40000$$
$$x_2 + x_3 = 45000$$
$$x_1 + x_2 + x_3 + x_4 = 50000$$
$$x_1, x_2, x_3, x_4 \geq 0$$

(b) Linear Program

Student	
Age, Scholarship, GPA	#Tuples
20, 20000, 5	5000
20, 20000, 6	40000
15, 20000, 5	5000

(c) Table Summary

Figure 1.3: Filter Constraints based Regeneration

To make the above concrete, consider the following two filter CCs on *Std* table.

$$c_1 : |\sigma_{15 \leq Age < 35 \wedge 6 \leq GPA < 8}(Std)| = 40000$$
$$c_2 : |\sigma_{20 \leq Age < 40 \wedge 5 \leq GPA < 9}(Std)| = 45000$$

For simplicity, Figure 1.3 shows only the 2D data space comprising the Age and GPA 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 four disjoint filter-blocks: b_1, b_2, b_3, b_4, whose domains are depicted with dashed-line boundaries.

The corresponding LP constructed on these filter-blocks is shown in Figure 1.3(b). Here, x_1, x_2, x_3, x_4 correspond to the count variables for filter-blocks b_1, b_2, b_3, b_4, respectively. A

possible solution to the LP is: $\langle x_1 = 0, x_2 = 5000, x_3 = 40000, x_4 = 5000\rangle$. A table summary corresponding to the example LP solution is shown in Figure 1.3(c) where a single point is picked for each populated filter-block and is assigned multiplicity equal to the cardinality of the filter-block.

The complete details of the above procedure, including the partitioning, proof of optimality, summary construction mechanism and several optimizations are discussed in Chapter 4.

1.3.4 Projection Constraints (Chapter 5)

To model projection constraints, the key technical challenges faced are as follows:

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 CCs, we have the inclusion property:

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

That is, it may be possible that a tuple lying within the boundaries of $\pi_{\mathbb{A}_1}(b)$ and $\pi_{\mathbb{A}_2}(b)$, may be outside the boundary of b itself. Therefore, the projections along different subspaces cannot be dealt with independently.

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 groups b_1 and b_2, the direct expression for computing their 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, unlike filters, here union does not translate to a simple summation.

The key design principles incorporated in Hydra to handle the above challenges are the following:

Isolating Projections. To circumvent inter-projection subspace dependencies, we first "isolate" the projections. Specifically, the following set of steps are taken in this process: (discussed in detail in Chapter 5)

A *Symmetric Refinement* strategy is adopted that refines an filter-block into a set of disjoint *Refined Blocks* such that each resultant refined-block exhibits translation symmetry along each applicable projection subspace. That is, for each domain point of a refined-block r along a particular PAS, the projection of r along the remaining attributes is identical.

For instance, consider b_2 in Figure 1.3(a). Clearly, it is asymmetric along Age – specifically, compare the spatial layout in the range $20 \leq Age < 35$ with that in $35 \leq Age < 40$. After refinement, this block breaks into r_{2a} and r_{2b} as shown in Figure 1.4 – it is easy to see that r_{2a} and r_{2b} are symmetric. This refinement allows for the values along different projection subspaces to be generated independently.

The above refinement, however, does not scale when the projections applied on a region are along partially overlapping attribute-sets. To eliminate such situations, we resort to a *Workload Decomposition* strategy to split 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. 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 generated is minimized.

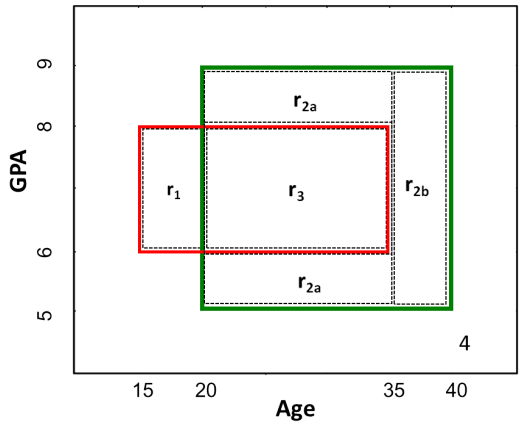


Figure 1.4: Symmetric Refinement

Age	Scholarship, GPA	#Tuples
[20, 40): 5	(S) [20000, 50000), (G) [5,6): 12 (S) [20000, 50000), (G) [8, 9): 08	5000

Age	Scholarship	GPA	#Tuples
[20, 35): 3	[10000,30000): 4 [30000, 80000): 5	[6,8)	40000

Scholarship	Age, GPA	#Tuples
[30000, 80000): 5	(A) [15,20) U [35,40), (G) [9, 10): 6	5000

Figure 1.5: Table Summary

Projection Subspace Division. This technique divides each projection subspace into regions that allow modeling the unions into a summation of the cardinality of a subset of the regions obtained. For instance, by using a naive projection subspace division strategy to divide the subspace with respect to Age attribute from our running example, we can express $|\pi_{Age}(r_{2a} \cup r_{2b} \cup r_3)|$ (projection of c_2 along Age) as the following summation of cardinalities of four projection regions: $|(\pi_{Age}(r_{2a}) \setminus \pi_{Age}(r_3))| + |(\pi_{Age}(r_{2a}) \cap \pi_{Age}(r_3))| + |(\pi_{Age}(r_3) \setminus \pi_{Age}(r_{2a}))| + |\pi_{Age}(r_{2b})|$.

Hydra's strategy for projection subspace division gives the minimum number of such projection regions. These are called *constituent projection blocks* (CPBs). An LP variable is constructed for each refined-block and CPB to express the CCs as a summation of LP variables.

A sample table summary produced is shown in Figure 1.5. Each segment of the summary corresponds to a populated refined-block. For example, the first refined-block's tabulation can be interpreted as "generate 5000 tuples, such that there are 5 distinct values of Age in the interval `[20,40)`, and 20 distinct value pairs of $\{Scholarship, GPA\}$ such that 12 are from the 2D interval `[20000,50000), [5,6)`, and the remaining 8 from the 2D interval `[20000,50000), [8,9)`." For attributes that do not feature in any projection subspace, no associated distinct cardinalities are maintained. The complete details of projection handling are discussed in Chapter 5.

1.3.5 Join Constraints (Chapter 6)

To handle joins in the CCs, three additional steps are taken:

Denormalization. For each table T to be constructed, a corresponding *view* V_T is synthesized first. This view captures the denormalized equivalent of T (excluding the key columns). Processing on views helps in generating correlations that are compatible with the various join cardinalities in the input. The views constructed for our running example are as follows:

$$V_{Reg}(Course, Year, Score, Age, Scholarship, GPA), \quad V_{Std}(Age, Scholarship, GPA)$$

These views allow rewriting the join expression on a single view. For example, the first two CCs from Figure 1.1(d) can now be rewritten as:

$$|\pi_{Age}(\sigma_{Age\geq 25 \wedge Year=2021}(V_{Reg}))| = 10, \quad |\sigma_{Age\geq 25 \wedge Year=2021}(V_{Reg})| = 3000$$

Post-rewriting, the data space of each of these views is partitioned using the region partitioning and symmetric refinement algorithms.

Referential Constraints. To obtain original tables back from their denormalized equivalents, the views need to obey referential integrity. We know that *referential integrity* constraint between a fact table F containing foreign key $F.fk$ referencing dimension table D with primary key $D.pk$ is expressed as $\pi_{F.fk}(F) \subseteq \pi_{D.pk}(D)$. The equivalent expression of this constraint in terms of the views is the following:

$$\pi_{\mathbb{B}}(V_F) \subseteq V_D$$

where $\mathbb{B}$ is the set of columns in V_D, and hence is borrowed in V_F. Therefore we include additional constraints in the LP formulation that ensure these referential dependencies. Specifically, these *referential constraints* ensure that for each interval of the borrowed columns, the number of distinct values present in V_F is at most equal to the number of distinct values present in V_D.

To add referential constraints, the regions of V_F and V_D need to be aligned. Therefore, as a precursor, an *Align Refinement* stage may be required, to ensure that each refined-block in V_D is either contained or is disjoint with a refined-block in V_F along the subspace spanned by $\mathbb{B}$. The output of this stage is the set of *Aligned Refined Blocks* (ARBs).

Referential constraints only ensure cardinality subsumption. For subset property, we also need to further enforce value subsumption. This is done in the final stage of Key-Range Curation as described next.

Key-Range Curation. This final stage is responsible for the curation of FK values in F. Specifically, for each ARB a in V_F, to construct its equivalent in F, a range of FK values is assigned to it. This assignment is done using a range of PK values associated to a set of blocks in V_D, such that:

1. The chosen V_D blocks are contained within the boundaries of a after projecting along $\mathbb{B}$.
2. The tuples associated with the selected PK values have the desired number of distinct values along the PAS prescribed by the projection applied on the a.

In this way, we get the summary for each table, which is used for dynamic data regeneration. The complete details of join operator handling is discussed in Chapter 6.

1.3.6 Adding Robustness (Chapter 7)

The aforementioned suite of techniques ensure volumetric similarity on *seen* queries provided by the client. Generalization to new queries may be a requirement at the vendor site as part of the ongoing evaluation exercise. To address this expectation, the following design choices are added in Hydra:

Optimization Problem. There can be numerous feasible solutions to the LP designed by Hydra. Even though all these ensure volumetric similarity on seen queries, it is easily possible that several of these solutions are far from the solution that is representative of the client database. Hence, volumetric similarity for unseen queries can incur enormous errors. To address this, we construct finer regions by additionally using metadata stats. Subsequently, an LP is formulated, by adding an *Objective Function* that picks up a feasible solution that is close to the estimated solution derived from the stats. This also helps ensure metadata compliance.

Intra-Region Tuple Distribution. Within each populated region obtained from the LP solution, we try to obey the tuple distribution based on the information extracted from metadata stats. At the finer granularity where volumes cannot be estimated any further, we resort to uniform distribution.

We evaluate the efficacy of these additional modules by comparing their volumetric similarity with the basic Hydra on unseen queries. Our results indicate a substantive improvement – specifically, the volumetric similarity on filter constraints of unseen queries was better by more than 30 percent, as measured by the UMBRAE model-comparison metric [32]. Further, the produced database also is metadata compliant. Finally, the produced data also has a more realistic look. A sample set of rows produced by Hydra for a few columns of the TPC-DS table Item is shown in Figure 1.6.

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

Figure 1.6: Sample Output (Item Table)

The complete details of robustness addition are discussed in Chapter 7.

1.3.7 Prototype Implementation (Chapter 8)

The above mentioned modules are combined to produce an end-to-end data generation pipeline. A pictorial view is presented in Figure 1.7 – in this picture, the green boxes represent the various modules of Hydra. The key acronyms and notations used in the thesis are enumerated in Tables 3.1 and Table 3.2, respectively.

Client Site. The information flow from the client to the vendor is as follows: At the client site, Hydra fetches the schema information, and the query workload with its corresponding AQPs obtained from the database engine. The statistical metadata from the database catalogs is captured with the help of CODD [19] tool. All this information is then shipped to the vendor site.

Vendor Site. Using the schema, views are constructed using the Denormalization module. The query-workload is passed through a Workload Decomposition modules, which returns the set of non-conflicting sub-workloads. The AQPs wrt each sub-workload is converted to equivalent CCs using a Parser. Subsequently, the rest of the pipeline, comprising of Data Space Partitioning, LP Formulation and Summary Construction, is executed independently for each of these sub-workload CCs.

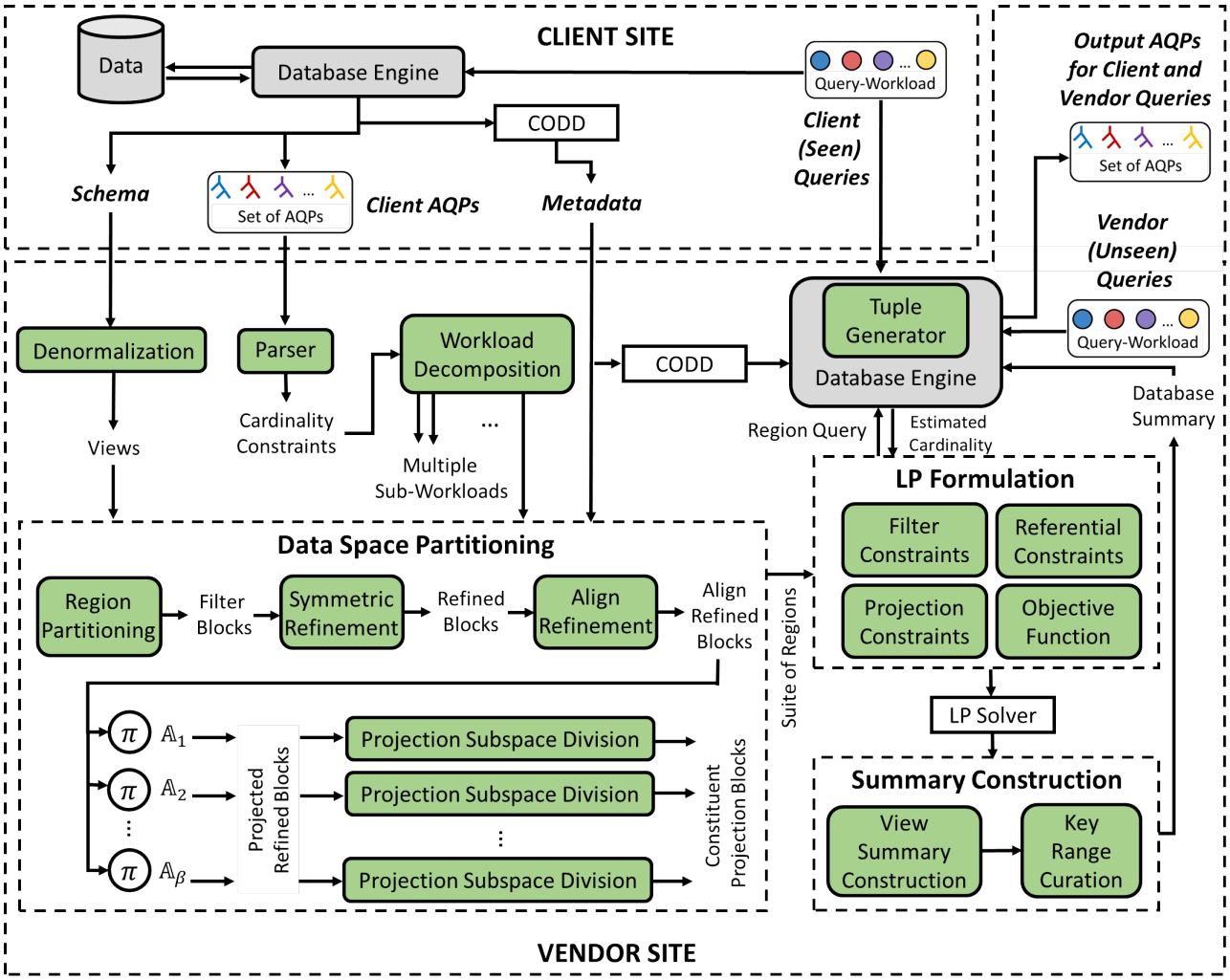


Figure 1.7: Hydra Architecture

The Data Space Partitioning for a View and sub-workload begins with Region Partitioning followed by Symmetric Refinement algorithm. This gives the set of refined-blocks. Further, due to Align Refinement across Fact and Dimension tables, the refined-blocks are split and give *Align Refined Blocks* (ARBs) in the output. The ARBs after being projected along the applicable PASs, give the *Projected Refined Blocks* (PRBs). These PRBs and sub-workload are then used by the Projection Subspace Division module to construct the set of CPBs.

Next, at the LP Formulation stage, an LP is constructed using variables representing the cardinalities of ARBs and CPBs. Specifically, Filter Constraints and Projection Constraints are modeled for each view. Subsequently, Referential Constraints are added between each pair of Fact and Dimension table. Finally, based on the cardinality estimation module of the database engine, an estimate of the size of various blocks is obtained. Using these estimates, an Objective

Function is added to the LP. This construction is then given as the input to the LP Solver. We have used the popular Z3 solver [14] from Microsoft for this purpose.

From the solution produced by the LP solver, a comprehensive table summary is constructed using the Summary Construction module. Specifically, it constructs View Summary and then replaces borrowed attributes in the fact tables with the corresponding foreign key columns using the Key Curation module.

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

The complete pipeline, the implementation detail and the visual interface are discussed in Chapter 8.

1.3.8 Extensions (Chapter 9)

The techniques discussed in the thesis focus on ensuring volumetric similarity. In Chapter 9 we go beyond volumetric similarity and discuss two other data characteristics – namely, Duplication Distribution and Presortedness, which are important for mimicking client data processing environments. Specifically, we discuss their mathematical characterization, mechanism of extraction and preliminary ideas on modeling them for data generation.

1.4 Summary

Synthetic data generation from a set of cardinality constraints has been strongly advocated in the contemporary database testing literature. Hydra provides a comprehensive solution to support SPJ constraints. Further, Hydra introduces the concept of dynamic data regeneration using minuscule database summaries. These summaries are produced in a data-scale-free manner and is hence extremely efficient and scalable. Hydra further provides a mechanism to obtain reasonable accuracy on unseen constraints. The experimental evaluation on real-world and synthetic benchmarks indicated that Hydra successfully produces generation summaries with viable time and space overheads.

Chapter 2

Related Work

Over the past few decades, a rich corpus of literature has developed on synthetic database construction. There are two broad streams of research on the topic, one dealing with the *ab initio* generation of new databases using standard mathematical distributions (e.g. [39, 28]), and the other with *regeneration* of an arbitrary existing database. In the latter category, there are two approaches, one of which uses only schematic and statistical information from the original database (e.g. [61, 66]). The other uses both the original database and the query workload to achieve statistical fidelity during evaluation (e.g [53, 17]) – our work on Hydra falls into this class. In this section, we review recent literature on this spectrum of research categories.

2.1 Ab Initio Generation

The frameworks in this category deal with the *ab initio* generation of new databases using standard mathematical distributions. For example, [39] was one of the early work in the area. The focus here was to build parallel algorithms to generate data sets with different distributions and dense unique sequences in linear time. Thereafter, a special purpose language called Data Generation Language (DGL) was proposed in [28]. It is used in generating synthetic data distributions by utilizing the concept of iterators. While the work supports a broad range of dependencies between relations, the construction of dependent tables always requires access to the referenced table, creating a bottleneck on the data generation speed. A graph based generation tool was proposed in [41], that models dependencies in a graph and uses a depth-first traversal to generate dependent tables.

Two subsequent tools that offer similar capabilities are MUDD [68] and PSDG [40]. They generate all related data at the same time thereby providing parallelism in the data generation process. MUDD proposed algorithms to efficiently generate dense-unique-pseudo-random

sequences and derive nonuniform distributions. Further, it gives the concept of separating the data generation from data distribution definition. This enables users to quickly change data distributions to suit individual needs. Like MUDD, PSDG decouples data generation details from data description. It generates a constant output for a given input file regardless of the degree of parallelism. The data is described and constrained using an XML-based language called Synthetic Data Description Language (SDDL). While SDDL includes constructs to express intra-row, inter-row, and inter-table dependencies, it supports restrictive data distributions. Further, even with the benefits of parallelism, due to the requirement of generating all the referenced tables, the above techniques can be rendered inefficient, especially if the referenced tables are large in size or if some table is not required to be generated during a testing exercise.

A faster way of generating references is recomputing them in the distributed setting. It helps in getting rid of the I/O cost incurred to satisfy referential constraints across tables that are present across different nodes. PDGF [62] was designed with this goal of achieving scalability and decoupling. In PDGF, the user specifies two XML configuration files, one for the data model and the other for the formatting instructions. The generation strategy is based on the exploitation of determinism in pseudo random number generators (PRNG), which enables regeneration of the same sequences, hence eliminating the scan overheads. The multi-layer seeding strategy used in PDGF makes it possible to generate data with cyclic dependencies as well. However, this induces high computation cost for generating the keys. Also, PDGF comes with a set of fixed generators for different datatypes and with some basic distribution functions. A scheduler is responsible for dividing the work among physical nodes and the worker threads on each node. The work is divided between nodes and workers equally making it suitable for homogeneous clusters. A similar generator is Myriad [16], which implements an efficient parallel execution strategy leveraged by extensive use of PRNGs with random access support. With these PRNGs, Myriad distributes the generation process across the compute nodes and ensures that they can run independently from each other, without imposing any restrictions on the data modeling language.

Finally, a rule-based probabilistic approach, based on an extension of Datalog was proposed in [22], which is capable of generating data characterized by parameterized classical discrete distributions.

A common setback in all these frameworks is that they use standard distributions to generate the data. And it is not always feasible to assign such distributions to real-world data, especially over multivariate spaces. Further, the generated database tends to give empty results over complex queries since the subtle correlations between attributes are often not captured. Therefore, mimicking customer environments remains a challenge.

2.2 Database-Dependent Regeneration

We now turn our attention to database-dependent regeneration techniques. DBSynth[61], an extension to PDGF, builds data models from an existing database by extracting schema information and sampling. If sampling is permissible, histograms and dictionaries of text-valued data are built. Also, if the textual data contains multiple words, then Markov chain generators are used. These help in analyzing the word combination frequencies and probabilities. However, if sampling cannot be done, DBSynth falls back to random values. Finally, after the model construction is complete, PDGF is invoked to generate the corresponding data.

Like DBSynth, RSGen[66] takes a metadata dump, including 1-D histograms, as the input, and generates database tables along with a loading script as the output. It uses a bucket based model at its core, which is able to generate trillions of records with minimum memory footprint. However, the proposed technique works well only for queries with single attribute range predicate.

UpSizeR [71] is a graph-based tool that uses attribute correlations extracted from an existing database to generate an equivalent synthetic database. A derivative work, Rex [29] produces an extrapolated database given an integer scaling factor and the original database, while maintaining referential constraints and the distributions between the consecutive linked tables. UpSizeR clusters the values in each FK, does co-clustering, then generates FK values per co-cluster.

In contrast to UpSizeR, Dscaler [82] replicates per-tuple correlation patterns for key attributes using the concept of correlation database. This facilitates non-uniform scaling[1] and greater similarity. However, obtaining these per-tuple correlations themselves is typically hard. Moreover, all these techniques only generate the *key* attributes, whereas the non-key values are sampled from the original database using these key values. Hence, the approach becomes impractical in Big Data and security-conscious environments. Finally, Dscaler fails to retain volumetric similarity for some common query classes.

A different trajectory of Database-Dependent Regeneration has been followed by the ML community. The techniques here were presented in a comprehensive study by Fan et al. [36]. They broadly classified the frameworks into *statistical models* and *neural models* as follows:

Statistical Models. The aim is to model the input data as a joint multivariate distribution and generate synthetic data by sampling from this distribution. The dependence between variates is captured using techniques such as copulas [51, 60], Bayesian networks [81], Gibbs sampling [58], and Fourier decompositions [21]. Other than these, synopses-based

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

approaches such as wavelets and multi-dimensional sketches, build compact data summaries which can be then used for estimating joint distributions [30, 77]. However, statistical models may have limitations on effectively balancing privacy and data utility [36].

Neural Models. The frameworks here use deep generative models to approximate the distribution of the original data. The techniques adopted are based on autoencoders [38], variational autoencoders (VAE) [72], and more recently generative adversarial networks (GANs) [31, 78, 59, 20, 33, 56].

The study performed thorough evaluation of the techniques and found that GAN are promising for relational data synthesis.

Despite the appeal, applying these techniques for our usecases is not straightforward because of the following issues: (a) They focus on single table synthesis. Therefore, evaluation of even simple queries involving joins between multiple tables cannot be satisfactorily performed; (b) Training a GAN requires original data which may itself not be provided by the client; (c) These generators largely focus on development of ML models over the synthetic databases. Establishing their utility for ensuring volumetric similarity is an area for future work.

Apart from the above, there is extensive work in the area of data privacy. The work in this area (e.g. [35, 43, 50, 48, 49, 69, 76]) has largely focused on generating query answers that do not expose features of the underlying private data, rather than generate the data itself [37]. Although, there is some work in the area of privacy-preserving data publishing (e.g. [15, 27, 34, 47]). Here, the focus is on sharing data utility in a way that preserves sensitive information. Our work differs from this line of work as we assume no access to the client database instance. This lack of client data access could stem from reasons including (a) Client may not be comfortable in sharing the data; (b) There could be regulatory protocols such as GDPR (General Data Protection Regulation [4]) – in fact, such constraints have gained much traction in the recent times. For example, the recent work by Beedkar et al. [23] studies data processing under such constraints; (c) Data transfer overheads from client to vendor, especially in the current Big Data era.

2.3 Query-Dependent Regeneration

In more recent times, generation techniques driven by constraints on query outputs have been analyzed. Here the aim is to generate a database given a workload of queries such that volumetric similarity is achieved on these queries. A particularly potent effort in this class was Reverse Query Processing (RQP) [24], 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 [53, 55, 17, 52, 37, 80] fall in this category. They can be classified into two groups based on the nature of constraints.

Parameterized Constraints: Here the queries are parameterized with regard to the constants in the query, and the generators (e.g., QAGen [25], MyBenchmark [55], TouchStone [52]) additionally also provide parameter values for which the intermediate cardinalities are matched.

Value Constraints: Here predicate constants are pre-specified in the input as part of the workload. The generators (e.g., [17], [37], [80]) ensure that the intermediate cardinalities are matched for these constants.

We already discussed examples of both type of constraints in the previous chapter. While the parameterized constraints may be preferable from a privacy perspective, value constraints 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. Therefore, our work is based on this category. We further discuss the various query-dependent regeneration frameworks in greater detail.

2.3.1 Reverse Query Processing (RQP)

RQP [24] gets a query and a result as input and returns a possible database instance that could have produced that result for that query. Specifically, the problem addressed is as follows: "Given a SQL Query Q, the Schema S_D of a relational database (including integrity constraints), and a Table R (called RTable), the goal of RQP is to find a database instance D such that: $R = Q(D)$, and D is compliant with S_D and its integrity constraints."

Generally, there are many different database instances that can be generated for a given Q and R. Amongst them, RQP tries to generate a small database instance.

The overall architecture of RQP is shown in Figure 2.1. An input query is (reverse) processed by the following components: (The description of the components is largely sourced from [24].)

Parser: The SQL query is parsed into a query tree. This parsing is carried out in exactly the same way as in a traditional SQL processor. What makes RQP special is that that query tree is translated into a reverse query tree T_Q. In the query tree T_Q, each operator of the relational algebra is translated into a corresponding operator of the reverse relational algebra[1].

[1]In a strict mathematical sense, the reverse relational algebra is not an algebra and its operators are not operators because they allow different outputs for the same input.

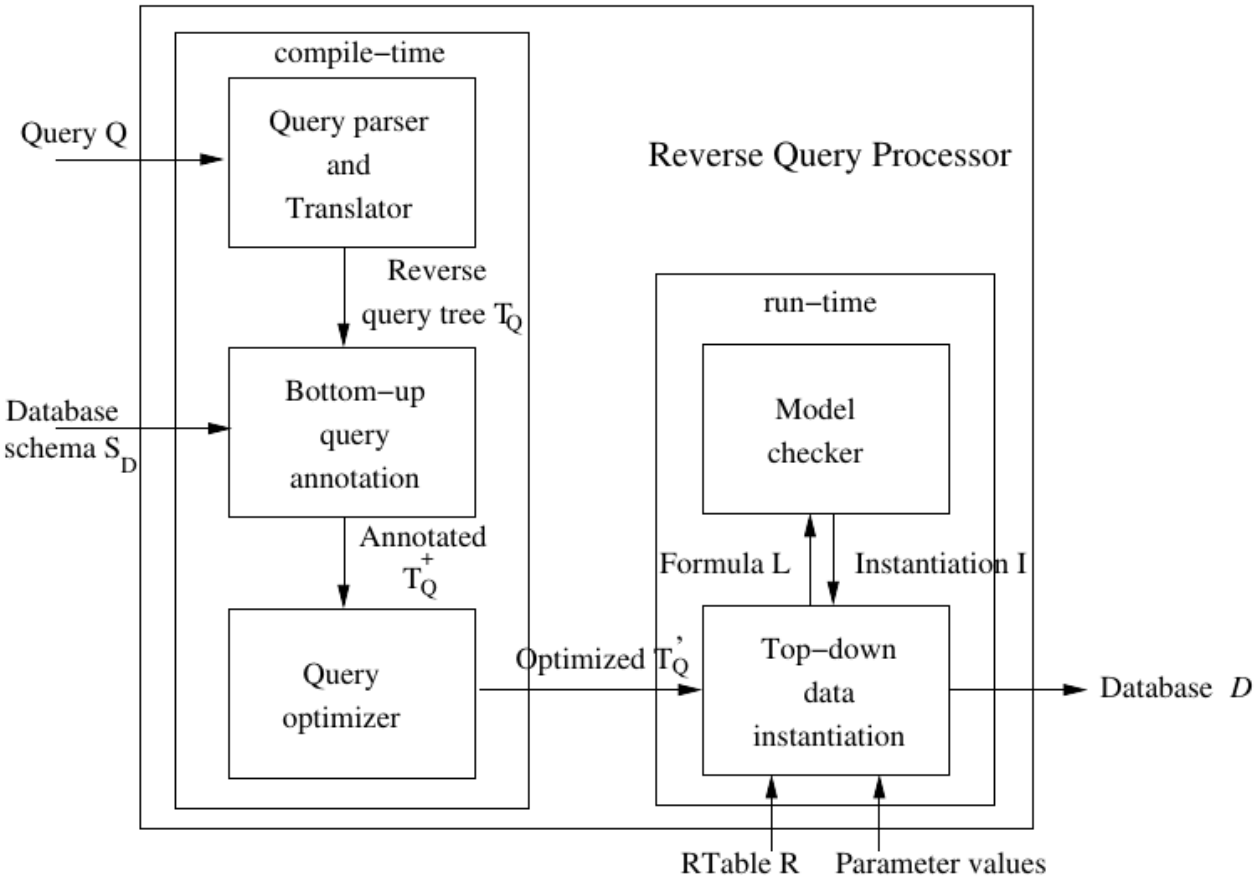


Figure 2.1: Architecture of RQP [24]

Bottom-up Query Annotation: The second step is to propagate schema information (types, attribute names, functional dependencies, and integrity constraints) to the operators of the query tree. Furthermore, properties of the query (e.g., predicates) are propagated to the operators of the reverse query tree. As a result, each operator of the query tree is annotated with constraints that specify all necessary conditions of its result, to give T_Q^+. This, for instance, guarantees that a top-level operator of the reverse query tree does not generate any data that violates one of the database integrity constraints.

Query Optimization: In the last step of compilation, the annotated query tree T_Q^+ is transformed into an equivalent optimized reverse query tree T_Q' that is expected to be more efficient at run-time.

Top-down Data Instantiation: At run-time, the annotated reverse query tree is interpreted using R as input. Just as in traditional query processing, there is a physical implemen-

tation for each operator of the reverse relational algebra that is used for reverse query execution. The result of this step is a valid database instance D. As part of this step, a model checker is used in order to generate the data.

We would like to emphasize that RQP differs from the frameworks that focus on preserving volumetric similarity since RQP focuses on matching the final query result as opposed to cardinalities at intermediate nodes in the query execution plan.

2.3.2 Query Aware Generation (QAGen)

The idea of using cardinalities from a query plan tree for data regeneration was first introduced in QAGen [25, 53]. It deals with generating synthetic data for a single query plan only. The general architecture of QAGen is shown in Figure 2.2. The various modules in the pipeline are described next: (The modules' description is largely sourced from [25].)

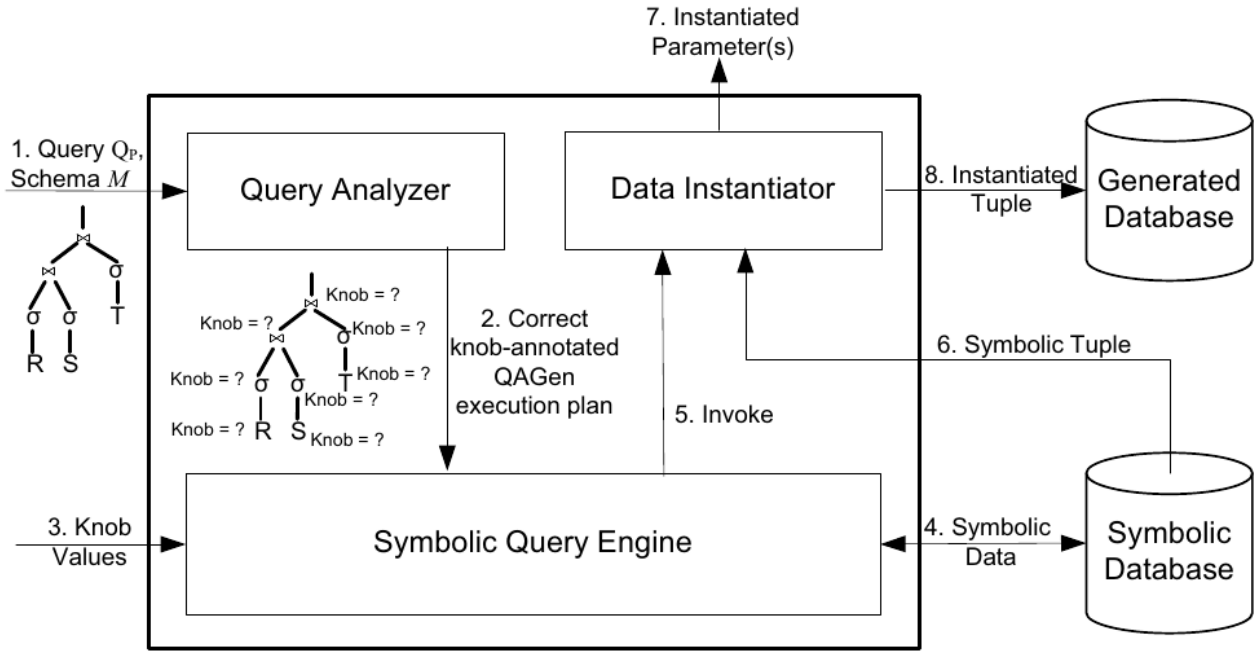


Figure 2.2: Architecture of QAGen [25]

Query Analyzer. First, the database schema M and the input query Q_P are taken as input by the Query Analyzer. The output of this process is a *knob-annotated* query execution plan. A knob can be regarded as a parameter of an operator that controls the output. A basic knob that is offered by QAGen is the output cardinality constraint. This knob

allows a user to control the output size of an operator. However, whether a knob is applicable depends on the operator and its input characteristics.

Symbolic Query Processing (SQP). After parsing, the next phase is of SQP. The goal here is to capture the user-defined constraints on the query into the target database. For this, QAGen integrates the concept of symbolic execution from software engineering into traditional query processing. Symbolic execution is a well known program verification technique, which represents values of program variables with symbolic values instead of concrete data and manipulates expressions based on those symbolic values. Borrowing this concept, QAGen first constructs a symbolic database containing a set of symbols instead of concrete data. Since the symbolic database provides an abstract representation for concrete data, this allows controlling the output of each operator of the query. The input query is executed by a *Symbolic Query Engine* just like in traditional query processing; i.e., each operator is implemented as an iterator, and the data flows from the base tables up to the root of the query tree. Each operator manipulates the input symbolic data according to the operator's semantics and the user-defined constraints, and incrementally imposes the constraints defined on the operators to the symbolic database. After this phase, the symbolic database is then a query-aware database that captures all requirements defined by the test case of the input query (but without concrete data).

Data Instantiation. This phase reads in tuples from the symbolic database that are prepared by the SQP phase and subsequently, a *Constraint Satisfaction Program* (CSP) is invoked to identify values for symbols that satisfy all the constraints, and instantiates the symbols. The instantiated tuples are then inserted into the target database.

On the positive side, QAGen is capable of handling complex operators as they use a general CSP, but the performance cost is huge since the number of CSP calls substantially increases with the database size. Further, it requires operating on a symbolic database of matching size to the original database, and processing of the entire database during the algorithm execution. This makes it impractical for Big Data environments. Finally, as mentioned earlier, QAGen supports only one query plan in the input.

2.3.3 MyBenchmark

This single query input limitation of QAGen was addressed in a follow-up tool called MyBenchmark [54, 55], which creates a symbolic database on a per query basis and at the end tries to heuristically merge the various databases into a small number of databases. Specifically,

the problem addressed is as follows: "Given a database schema H, a set of annotated queries $Q = \{Q_1, Q_2, \ldots, Q_n\}$ (the operator(s) in Q_i are annotated with cardinality constraint(s) C_i), MyBenchmark generates m ($m \leq n$) databases $D_1, D_2, \ldots, D_m$ and m sets of parameter values $P_1, P_2, \ldots, P_m$, such that (1) all databases D_j ($1 \leq j \leq m$) conform to H, and (2) the resulting cardinalities C_i of executing Q_i on one of the generated databases D_j, using the parameter values P_j, approximately meet C_i (the degree of approximation defined is based on the relative error between actual cardinalities and annotated cardinalities)."

If $m = n$, that essentially means MyBenchmark is the same as QAGen where one database is generated per query. MyBenchmark aims to minimize m, the number of generated databases.

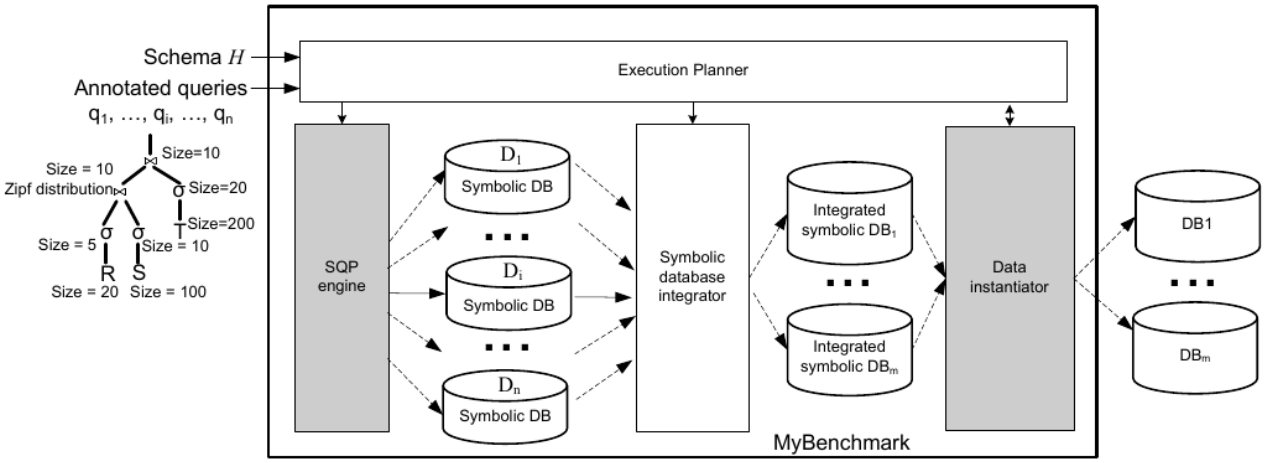


Figure 2.3: Architecture of MyBenchmark [55]

Figure 2.3 shows the architecture for MyBenchmark. The data flow through this pipeline, as described in [55], is as follows: "To generate m databases for n annotated queries $Q_1, Q_2, \ldots, Q_n$, MyBenchmark first uses QAGen's SQP engine as a black-box component to process each annotated query separately (without data instantiation) and generates n Symbolic Databases $D_1, D_2, \ldots, D_n$. Each symbolic database D_i guarantees that $Q_i(D_i)$ satisfies the constraints annotated on Q_i. Then, a *Symbolic Database Integrator* is used to integrate these databases. The integration algorithms are designed to minimize the number of symbolic tuples with contradicting constraints and the number of generated databases. Finally, the *Data Instantiator* of QAGen is used to instantiate each integrated symbolic database with concrete values."

This concept of constructing multiple databases for a single query workload is similar to our Workload Decomposition module. Our module constructs different sub-workloads which do not have overlapping projections, which also finally results in multiple database summaries. However, we differ on the key practical issue that MyBenchmark generates a database on a per

query basis to begin with, which results in enormous time and space overheads. In contrast, our approach is not only always in the summary space, but also data-scale-free.

2.3.4 Touchstone

Touchstone [52, 75] targeted the scalability limitations of its predecessors and achieved fully parallel data generation, linear scalability, and lean memory consumption for supporting the generation of enormous query-aware test databases.

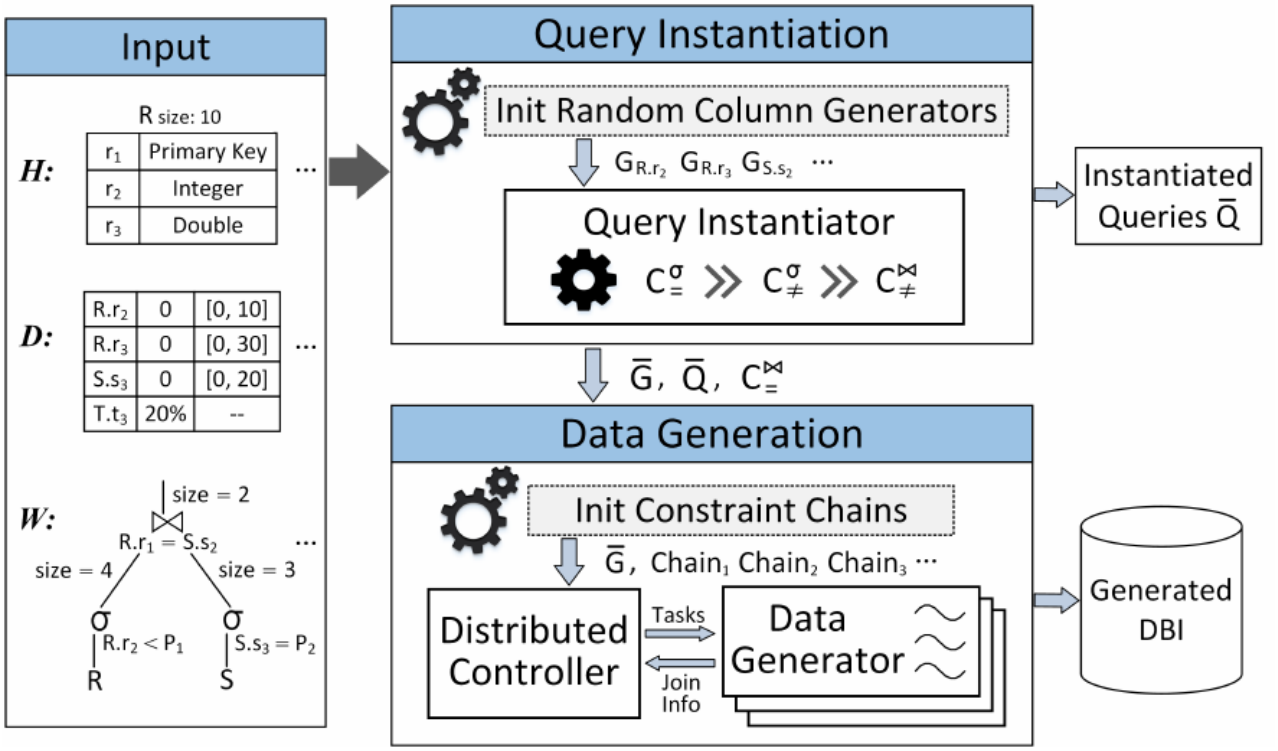


Figure 2.4: Architecture of Touchstone [52]

The input to Touchstone contains the database schema H, data characteristics of non-key columns D, and parameterized cardinality constraints. The infrastructure of Touchstone is divided into two major components, which are responsible for query instantiation and data generation, respectively, as shown in Figure 2.4. The details of the pipeline is as follows: (The pipeline description is largely borrowed from [52])

Touchstone builds a group of random column generators $\mathrm{G} = \{G_1, G_2, \ldots, G_n\}$, which determine the data distributions of all non-key columns to be generated. A random column generator G_i in G is capable of generating values for the specified column while meeting the required data characteristics in expectation.

Given the input workload characteristics W, Touchstone instantiates the parameterized queries by adjusting the related column generators if necessary and choosing appropriate values for the variable parameters in the filter and non-equi-join predicates. The instantiated queries $\overline{Q}$ are output to the users for reference, while the queries $\overline{Q}$ and the adjusted column generators $\overline{G}$ are fed into the data generation component.

Given instantiated queries $\overline{Q}$ and constraints over the equi-join operators, Touchstone decomposes the query trees annotated with constraints into constraint chains, in order to decouple the dependencies among columns, especially for PK-FK pairs. Data generation component generally deploys the data generators over a distributed platform. The random column generators and constraint chains are distributed to all data generators for independent and parallel tuple generation.

While TouchStone significantly improved the efficiency of data generation, it does not support projection based operators in the query. Further, equality constraints over filters involving multiple columns are also not supported. Also, an important thing to note is that the techniques adopted here do not translate to the value constraints case. For example, column generator usage and parameter values adjustment exploit the flexibility of dealing with query parameters.

2.3.5 DataSynth

DataSynth [17, 18] was the first work which identified the declarative property of cardinality constraints and its ability to specify data characteristics. Given a set of value cardinality constraints as input, the paper proposed algorithms based on the LP solver and graphical models to instantiate tables that satisfy those constraints.

DataSynth also constructs a denormalized table (variant) for each relation in the database first. The core idea here is also to express the constraints using an LP, where each variable represents the volume of a region of the data space. The regions are constructed using a grid-partitioning approach. In this approach, the domain of each attribute is intervalized based on the constants appearing in the CCs. This gives a grid structure which is aligned with the interval boundaries for each attribute. For each cell in the grid, a variable is created that represents the number of data rows present in that cell. For example, Figure 2.5 shows the grid partitioning for the example constraints considered in the Introduction. Each rectangular box (formed with dotted lines) forms a region in DataSynth and subsequently a variable in the LP. Therefore, for this example, it results in 25 variables. The solution of the LP is used to generate a denormalized table using a sampling based approach.

While DataSynth laid the foundation for Hydra, the latter has significantly extended the solution on multiple facets:

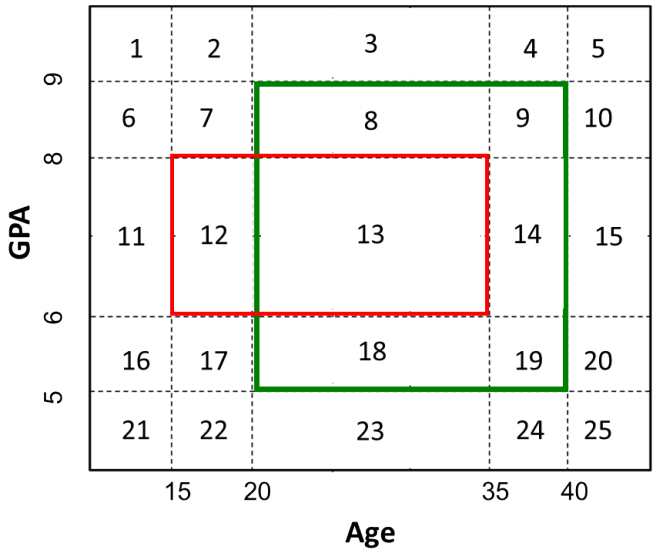


Figure 2.5: Grid Partitioning in DataSynth

Constraints Scope. The algorithm adopted in DataSynth handled constraints with filter and join operators satisfactorily, however, the support for the projection operator has been minimal, restricted to a few extreme cases. Specifically, proposed projection generator catered to *single-column* tables. Here, due to the single-column restriction, there are by definition no critical challenges of intra/inter projection subspace dependencies. Hydra gives a comprehensive solution to handle SPJ constraints.

LP Complexity. By replacing the grid-partitioning approach with region-partitioning, Hydra significantly reduces the complexity of the LP. For example, for the above example, grid partitioning led to 25 variables, while region partitioning of Hydra (shown in Figure 1.3) resulted in only four variables. This differences magnifies as the number of constraints grow. As a case in point, an LP with more than a billion variables in DataSynth got reduced to an LP with a few thousand variables in Hydra – in fact, in this case, the LP solver crashes on the DataSynth formulation, but runs to completion in less than a minute on the Hydra formulation.

Database Instantiation Overheads and Accuracy. In DataSynth, the sampling approach and its subsequent processes require iterating on the entire database multiples times. This is an computationally expensive, especially for Big Data systems. Further, the sampling technique also induces errors in volumetric fidelity. Hydra replaces the sampling strategy with a deterministic Align-Merge algorithm that not only eliminates the inaccuracies, but also is extremely efficient, thanks to its data-scale-free nature.

2.3.6 Linked Data Synthesis

Recently, a framework was proposed in [37] with the focus on generating key columns, such that, the generated data obeys referential integrity. Specifically, the input is a relation R_1 with an unknown foreign key dependence on a relation R_2, i.e., the FK column in R_1 is missing, and a set of CCs and Integrity Constraints (ICs). Further, for the ICs, they define a type of Denial Constraints (DCs), called Foreign Key DCs, that applies to R_1 and forbids tuples from having the same FK value under specified conditions.

The solution operates in two phases as shown in Figure 2.6. We briefly describe these next.

Phase 1. The goal here to construct the view $R_1 \bowtie R_2$ based on the CCs. This view is initialized with a copy of R_1 without the FK column, along with an empty column per non-key column in R_2. The algorithm for generating the view is inspired from the grid partitioning approach of DataSynth. Further, when the CCs do not have intersections and disjunctions, an optimization is used, where the containment and disjointedness of CCs are modeled as a Hasse diagram.

Phase 2. Here the generated view is used to construct the FK column so that the DCs are satisfied. For this, the concept of conflict hypergraphs and hypergraph coloring is used. Specifically, the tuples in R_1 are taken as vertices and an edge between a pair of vertices is added if the corresponding tuples will violate a DC if assigned the same foreign key. Here an approximate greedy algorithm for coloring is used and therefore leads to some artificial tuple addition in R_2.

Using the techniques in this paper for our settings has the following concerns:

1. The work assumes that the tables without the key column are given as input. This may not be acceptable to the clients due to privacy concerns.

2. Projection CCs are not considered here as well.

3. The focus is on satisfying constraints only on the foreign key table (R_1). Therefore, as a consequence of ensuring referential integrity, significant spurious tuple may be added to the referenced table (R_2).

4. The techniques have high computational overheads.

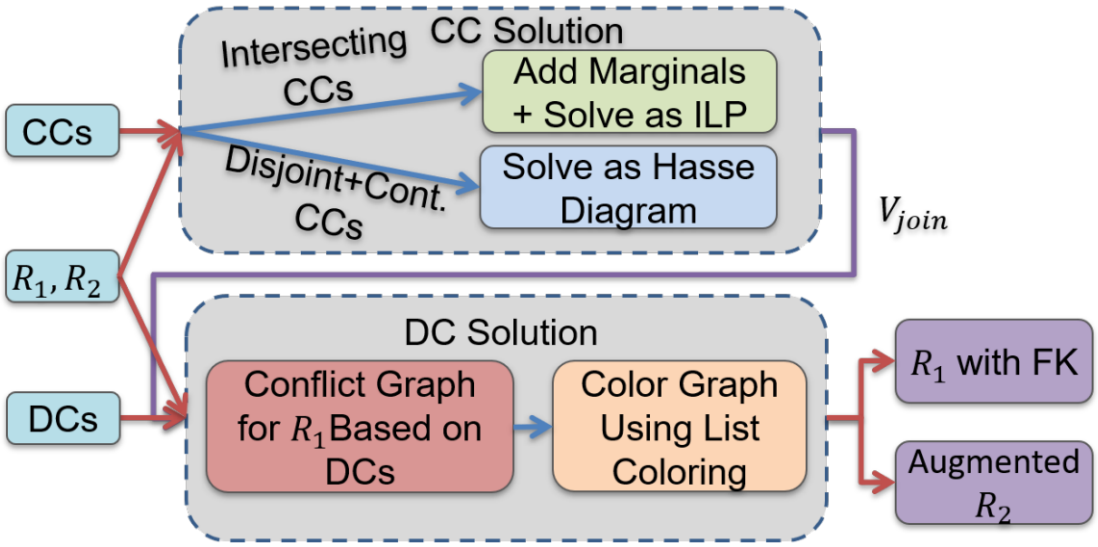


Figure 2.6: Linked Data Architecture [37]

2.3.7 Supervised Autoregressive Models (SAM)

We discussed several neural frameworks in Section 2.2. These frameworks used the original data in the input to train their models. An exception to these models is the recent work by Yang et al. [80], where a Supervised deep Autoregressive Model-based method for database generation (SAM) is proposed. SAM trains the model of joint data distribution of the entire database from a query workload, rather than the data directly. With an autoregressive model of the joint data distribution, it is straightforward to generate a database instance of a single relation. To address the challenges of multi-relation databases, the proposed approach includes inverse probability weighting and scaling algorithms to obtain unbiased base relation samples from full outer join samples, and Group-and-Merge algorithm to assign join keys to the generated relations.

Figure 2.7 shows the workflow of SAM, consisting of two stages – learning and generation. During the learning stage, batches of (query, cardinality) pairs are collected. From these, SAM learns the joint data distribution of the original database using differentiable progressive sampling. During generation, SAM first samples full outer join tuples from the training model. Further, inverse probability weighting is used to produce samples for each base relation. Finally, join keys are assigned using the Group-and-Merge algorithm.

While SAM advances ML based regeneration techniques, the proposed approach raises the following concerns:

1. Since SAM is probabilistic in behaviour, it is likely to make errors in matching output

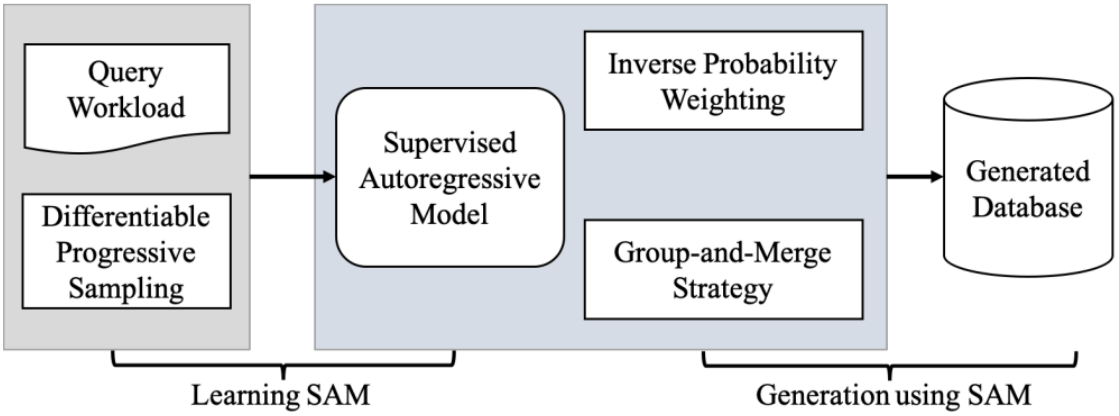


Figure 2.7: SAM Architecture [80]

row cardinalities, especially wrt fine-grained equality predicates. This is not the case with Hydra as it is deterministic in nature.

2. The framework's accuracy with respect to volumetric similarity is predicated on training the model with large query workloads.

3. Modeling has been done for the Select and Join operators, but support for projection based operators has not yet been established.

2.4 Miscellaneous

Hydra focuses on mimicking the client environments at the vendor site to aid in performance testing of the system subsequently. This falls under client-dependent testing approach, which is the most focused testing approach. However, an alternative is randomized creation of databases and queries - for instance, the optimizer testing approach by Waas and Galindo-Legaria [74] employs stochastic generation of queries. Alternatively, learning-based techniques can be used to detect anomalies during query execution, as implemented in SolarWinds [8].

Apart from performance testing, several techniques in the literature deal with functional testing of the system. For example, SQLsmith [10] has been effective in finding the queries that cause the DBMS process to crash. Similarly, general-purpose fuzzers such as AFL [1] are routinely applied to DBMSs, and have found many bugs. RAGS uses a differential testing based approach, where a bug is flagged if inconsistent outputs are obtained on evaluating automatically generated queries over multiple DBMSs. SQLancer [9] strives to find logic bugs in the DBMS that causes incorrect query results. It uses the following testing approaches: (a) Pivoted Query Synthesis (PQS) [63], where the core idea to automatically generate queries for

which it is expected that a specific, randomly selected row (called pivot row), is fetched – if it is not fetched, a bug is flagged; (b) Non-optimizing Reference Engine Construction (NoREC) [64], which aims to find optimization bugs by translating a query that is potentially optimized by the DBMS to one for which hardly any optimizations are applicable, and comparing the results – if there is a mismatch, a bug is flagged; (c) Ternary Logic Partitioning (TLP) [65], which partitions the query into three, whose results are composed and compared to the original query – a mismatch indicates a bug.

Chapter 3

Problem Framework

In this chapter, we summarize the problem statement, the underlying assumptions, the output delivered, and a tabulation of key notations used in the thesis.

3.1 Problem Statement

Given an SPJ query-workload $\mathcal{W}$, with its corresponding set of AQPs $\mathcal{Q}$, derived from an original database with schema $\mathcal{S}$ and statistical metadata $\mathcal{M}$, the objective is to generate a synthetic database $\mathcal{D}$ such that it conforms to $\mathcal{S}$ and $\mathcal{Q}$. That is, the AQPs obtained from the original database match, wrt the cardinality annotations, the AQPs obtained on $\mathcal{D}$.

3.2 Assumptions

We assume that $\mathcal{W}$ comprises of only PK-FK joins[1]. Further, we assume that the filters and projections are applied only on non-key columns. These assumptions are common in prior work as well as OLAP benchmarks.

For simplicity, we assume that $\mathcal{Q}$ is collectively feasible, that is, there exists at least one legal database instance conforming to $\mathcal{Q}$. In the regeneration usecase, this assumption holds trivially since the constraints are produced from an original client database. The infeasibility scenario is deferred to Section 3.7.

[1]By PK-FK joins, we mean that for each join node in the query plan tree, the participating intermediate tables should possess PK-FK nature, where one table has join column with all distinct values, and the other table has the join column with the subset property. In a nutshell, we want that queries can be mapped to denormalized tables.

3.3 Output

Given $\mathcal{S}$, $\mathcal{M}$, $\mathcal{W}$ and $\mathcal{Q}$, Hydra outputs a collection of database summaries $\mathbb{S}$. Each summary $s^{\mathcal{D}} \in \mathbb{S}$ can be used to deterministically produce the associated database $\mathcal{D}$. The databases produced are such that: (a) all of them conform to $\mathcal{S}$, and (b) for each query in $\mathcal{W}$, its corresponding AQP in $\mathcal{Q}$ matches with the AQP obtained on at least one output database instance.

3.4 Notations

The main acronyms and key notations used in the thesis are summarized in Tables 3.1 and Table 3.2, respectively. We provide a more detailed notation set in the respective chapters.

Table 3.1: Acronyms

Acronym	Meaning
AQP	Annotated Query Plan
CC	Cardinality Constraint
PIC	Projection-inclusive Constraint
SPJ	Select Project Join
DNF	Disjunctive Normal Form
PAS	Projection Attribute Set
PRB	Projected Refined Block
ARB	Aligned Refined Block
CPB	Constituent Projection Block
PSD	Projection Subspace Division
TAS	Target Attribute-Set

The mechanism for synthesizing data is explained progressively in the upcoming chapters. Specifically, we begin by discussing the single table synthesis method, where initially we assume only filter operators in the workload in Chapter 4, and subsequently include projection operator in Chapter 5. The more general problem, including multiple table synthesis with join operator in the workload, is discussed in Chapter 6.

We now discuss the simplified problem frameworks for Chapters 4 and 5.

3.5 Filter Constraints Problem

Here, we assume the AQPs are given for producing a single table T with the schema $\mathcal{S}$. Further, these AQPs are comprised of only filter predicates over T, as shown in Figure 3.1(a). Therefore, beside the base table cardinality, each AQP can consist of a series of filters and their associated cardinalities. Each filtered output can be represented as a CC of the form $\langle f, l \rangle$. This means

Table 3.2: Notations

(a) Database Related

Symbol	Meaning
$\mathcal{S}$	Database Schema
$\mathcal{D}$	Output Database
$\mathcal{M}$	Statistical Metadata
T	Output Table
A	Attribute
t	Tuple
$s^{\mathcal{D}}$	Summary of $\mathcal{D}$
s^{T}	Summary of T
F	Fact Table
D	Dimension Table
V_T	View wrt T
$\mathbb{B}$	Borrowed Attribute-Set
S	Sub-Table
$\mathcal{A}$	Attributes in a S

(b) Workload Related

Symbol	Meaning
q	Query
$\mathcal{W}$	Query Workload
$\mathcal{Q}$	Set of AQPs
c	A CC
f	Filter Predicate
$\mathbb{A}$	PAS
l	Output row card. after filter
k	Output row card. after projection

(c) Block Related

Symbol	Meaning
b	filter-block
r	refined-block
a	ARB
p	CPB

(d) Relation/Function Related

Symbol	Meaning
$U(T)$	Set of attributes wrt the input table
$dom(.)$	Domain of the input parameter
R	A relation btw two tuples
M	A relation btw CCs and blocks
L	A relation btw CPBs and blocks
H	A relation btw ARBs wrt V_F and V_D
J	A relation btw CPBs wrt V_F and V_D

that l tuples qualify after applying f filter predicate on the table T. For example, consider the following filter constraint on Std ($RollNo$, Age, GPA, $Scholarship$):

$$\langle f, 4000 \rangle \mid f = (Age \geq 30 \wedge 4 \leq GPA < 9)$$

The above constraint denotes that applying the f predicate on Std should produce 4000 rows in the output. We show a visualization of this constraint wrt a sample Std Table in Figure 3.2. In the figure, the purple rectangle depicts the filter predicate. We see that there are some points instead the rectangle denoting the tuples in the table. Further, each tuple is annotated with a number representing its cardinality. If we sum the cardinalities wrt all the tuples in the purple rectangle, we indeed get 4000, which is the filter output.

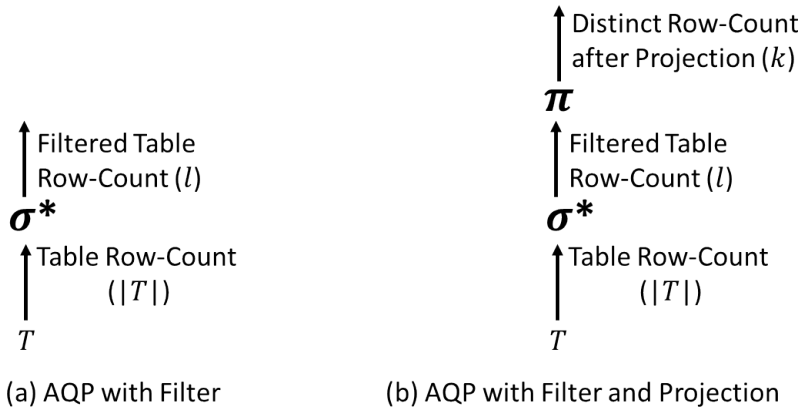


Figure 3.1: AQP with (a) Filter (b) Filter and Projection

A filter predicate f can be followed by another filter f' in the AQP. Then, the corresponding filter CC would have $f \wedge f'$ as the filter condition.

In Chapter 4, we simply assume a given set $\mathbb{C}$ of filter CCs in the input. Using $\mathbb{C}$, Hydra produces a table summary s^T that can be used to deterministically produce the associated table T. Further, T conforms to $\mathcal{S}$ and satisfies $\mathbb{C}$.

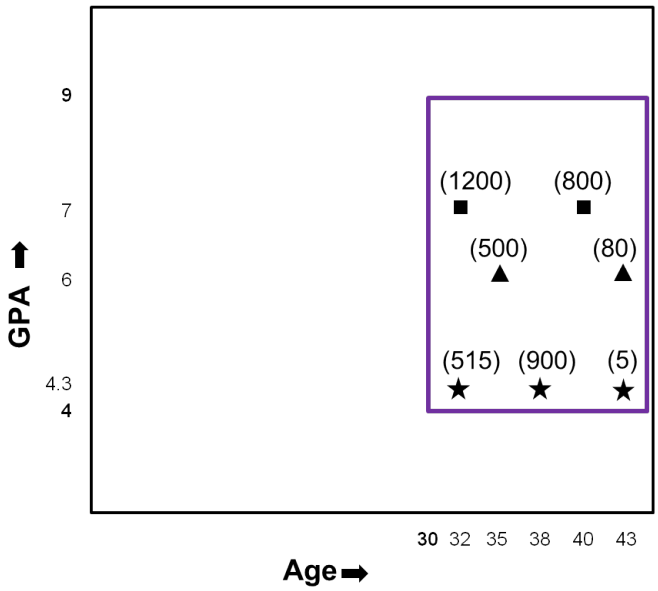


Figure 3.2: Filter and Projection Visualization

3.6 Projection Constraints Problem

Here as well, we assume that the AQPs are given for a single table T with the schema $\mathcal{S}$. The AQPs now include projection on top of the filter predicate result on T. Therefore, besides base table cardinality, each AQP gives two CCs, one filter CC and one projection CC. This pair of CCs is collectively called a Projection-inclusive Cardinality Constraint (PIC). A PIC is represented using the quadruple $\langle f, \mathbb{A}, l, k \rangle$, as a shorthand notation. Here, f represents the filter predicate applied on 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. For example, consider the following PIC on $Std(RollNo, Age, GPA, Scholarship)$:

$$\langle f, GPA, 4000, \mathbf{3} \rangle \mid f = (Age \geq 30 \wedge 4 \leq GPA < 9)$$

The above PIC denotes that applying the f predicate on Std should produce 4000 rows in the output, which is further reduced to 3 rows after projecting on the GPA column. Let us revisit Figure 3.2. We can see that the points in the rectangle had three kinds of shape – rectangle, triangle and star. These three shapes denote the three distinct GPA values (4.3, 6 and 7) present within the filter predicate boundary.

In Chapter 5, we simply assume a given workload $\mathcal{Q}$ of PICs in the input. Using $\mathcal{Q}$, Hydra outputs a collection of table summaries $\mathbb{S}$. Each summary $s^T \in \mathbb{S}$ can be used to deterministically produce the associated table T. The tables produced are such that: (a) all of them conform to $\mathcal{S}$, and (b) each PIC in $\mathcal{Q}$ is satisfied by at least one of them.

3.7 Workload Feasibility

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, [46] 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 [26, 70, 44]. A class of constraints, called **BT** inequalities, were proposed in [26], 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 [70]. 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 behavior, making it inexpressible as a set of linear constraints [44].

Although we assume a feasible workload as input to Hydra, if the vendor, while constructing "what-if" scenarios, ends up giving an infeasible workload as input, the following two possibilities may arise:

1. It may so happen that Workload Decomposition, while resolving conflicting projections, may as a *collateral benefit*, also produce feasible sub-workloads. In this scenario, Hydra produces one database per sub-workload.

2. Alternatively, in case this beneficial effect of decomposition does not happen, then the LP constraints 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.

Chapter 4

Regeneration using Filter Constraints

4.1 Introduction

This chapter discusses the mechanism for synthesizing a table that is compliant with a given set of filter cardinality constraints.

4.1.1 Filter Cardinality Constraints

A filter cardinality constraint applicable on a synthetic table T is of the following form:

$$|\sigma_f(T)| = l$$

Here, f is a selection predicate and l is a non-negative integer equal to the number of rows satisfying predicate f. We assume that each predicate is in disjunctive normal form (DNF).

As a shorthand notation, we represent the above filter CC using the pair $\langle f, l \rangle$. Also, for brevity, we refer to filter CCs as CCs simply in the rest of this chapter.

As a sample instance, consider the following pair of CCs on a generated table *Std* (*RollNo*, *Age*, *GPA*, *Scholarship*):

$$c_1 : \langle f_1, 40000 \rangle \mid f_1 = (15 \leq Age < 35 \wedge 6 \leq GPA < 8)$$
$$c_2 : \langle f_2, 45000 \rangle \mid f_2 = (20 \leq Age < 40 \wedge 5 \leq GPA < 9)$$

Here, c_1 denotes that applying the f_1 predicate on *Std* should produce 40000 rows in the output; likewise, c_2 denotes that 45000 rows qualify the f_2 predicate.

4.1.2 Technical Challenges

The key challenges in handling a set of filter constraints are as follows:

Overlapping Filters. For a given set of CCs, the filter predicates in these constraints may *intersect*. A pair of filters f_1 and f_2 intersect if there is a point t in the domain space of the table that satisfies both f_1 and f_2. Intersection among filters induces an interdependence among the CCs that has to be accounted for while generating the data. For example, the filters wrt c_1 and c_2 in the aforementioned example were intersecting. Therefore, the data for them cannot be generated independently.

Curse of Dimensionality. The CCs that do not have a direct intersection, may indirectly intersect due to the presence of additional dimensions in the table. For example, consider the following two CCs: $\langle Age < 25, 25000\rangle, \langle GPA < 8, 43000\rangle$. If the data for the table along the two dimensions Age and GPA is generated independently and simply concatenated thereafter, then both the CCs are satisfied. However, if we consider the both the dimensions together, then the 2-dimensional data points cannot be generated independently for the two CCs. Therefore, as the number of dimensions being considered increase, typically, this kind of dependence among the constraints become more profound.

4.1.3 Our Contributions

Hydra adopts a *partitioning* strategy that divides the data space into *optimal* (minimum) number of regions. This provides an efficient way to handle overlapping filters, as the input CCs can be modelled using an LP with lower complexity. Further, a combination of *dimensionality reduction* strategy and an *align-merge* strategy to recover the original space, help to overcome the intersections in higher dimensions. Hence, these techniques provide Hydra the strength to overcome the limitations of prior work in terms of (a) handling larger workloads, (b) providing efficient end-to-end solution, (c) giving improved volumetric similarity.

Therefore, the key contributions of the ideas presented in this chapter are as follows:

Extended Workload Coverage: Hydra incorporates a novel LP formulation technique, *region-partitioning*, that can encode volumetric constraints with an LP of low complexity. When compared with the *grid-partitioning* approach used in DataSynth, region-partitioning reduces the LP complexity by many orders of magnitude. For instance, an LP with more than a *billion* variables in DataSynth is reduced to an LP with a few *thousand* variables in Hydra– in fact, in this case, the LP solver crashes on the DataSynth formulation, but runs to completion in less than a minute on the Hydra formulation. The

beneficial outcome of the low LP complexity is that it facilitates the efficient handling of much richer query workloads.

Efficiency: To synthesize the output table from its lower dimensional parts, Hydra replaces the *sampling-based* approach proposed in DataSynth by an *align-merge* strategy. The primary benefit of the latter is that it operates directly on the table summary, and is therefore extremely efficient.

Accuracy: The aforementioned align-merge strategy, is deterministic in nature, and does not suffer the probabilistic errors that affect the sampling approach, and therefore delivers ideal fidelity with regard to volumetric similarity.

Enhanced Evaluation: We evaluate Hydra on workloads of 300-plus filter CCs constructed from the complex TPC-DS benchmark and the real world JOB benchmark, and the results show that it can efficiently regenerate data for such workloads at various data scales. This makes our evaluation more comprehensive than prior techniques, which have largely been evaluated on simpler and small-sized query workloads operating on modest databases. For instance, DataSynth has been evaluated on simple TPC-H database environments that resulted, with their formulation, in LPs with only a few thousand variables.

The entire end-to-end pipeline operates in the summary space, hence providing data-scale-independence.

4.1.4 Organization

The remainder of this chapter is organized as follows: The problem framework is discussed in Section 4.2. The key design principles in handling filter CCs are introduced in Section 4.3, and then described in detail in Sections 4.4 through 4.6. The mechanism to handle LIKE filter predicates is discussed in Section 4.7. The experimental evaluation is reported in Section 4.8. Finally we conclude in Section 4.9.

4.2 Problem Framework

We summarize the basic problem statement, assumptions, the output delivered, and a tabulation of the notations used in this chapter.

4.2.1 Problem Statement

Given a workload $\mathbb{C}$ of filter CCs and a table schema $\mathcal{S}$, the objective of data generation is to synthesize a table instance T such that it conforms to $\mathcal{S}$ and satisfies $\mathbb{C}$.

4.2.2 Assumptions

It is assumed that CCs consist of filters on only *non-key* attributes. Further, we assume the predicates are in the DNF form. Finally, for simplicity, we assume that the CCs are collectively feasible, that is, there exists at least one legal table instance satisfying all the constraints.

4.2.3 Output

Using $\mathbb{C}$ and $\mathcal{S}$, Hydra produces a table summary s^T that can be used to deterministically produce the associated table T. Further, T conforms to $\mathcal{S}$ and satisfies $\mathbb{C}$.

4.2.4 Notations

The main acronyms and key notations used in this chapter are summarized in Tables 4.1 and Table 4.2, respectively.

Table 4.1: Acronyms

Acronym	**Meaning**
CC	Cardinality Constraint
DNF	Disjunctive Normal Form

Table 4.2: Notations

(a) Database Related

Symbol	**Meaning**
$\mathcal{S}$	Table Schema
T	Output Table
A	Attribute
t	Tuple
s^T	Summary of T
S	Sub-Table
$\mathcal{A}$	Attributes in a S
K	Cardinality of $\mathcal{A}$
Ω	Domain of S

(b) Workload Related

Symbol	**Meaning**
c	A CC
f	Filter Predicate
l	Output row card. after filter
$\mathbb{C}$	Set of CCs

(c) Block Related

Symbol	**Meaning**
b	filter-block
$\mathbb{F}$	Partition (Set of filter-blocks)
x	variable for row cardinality

(d) Relation/Function Related

Symbol	**Meaning**
$dom(.)$	Domain of the input parameter
R	A relation btw two tuples

4.3 Design Principles

In this section, we give an overview of the core design principles of handling filter CCs, with the *Std* table of the Introduction used as the running example to explain their impact. Subsequently, in Sections 4.4 through 4.6, each principle is described in detail.

4.3.1 Region Partitioning

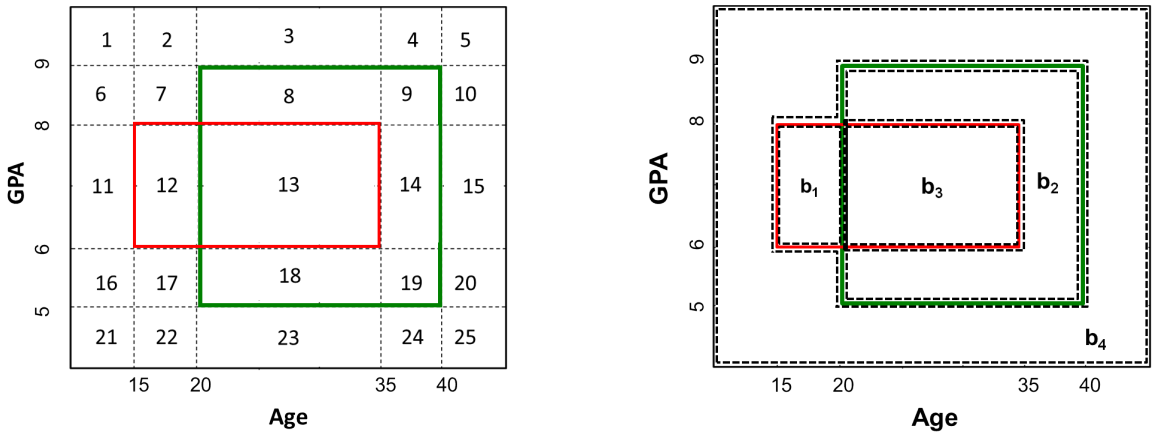
At its core, Hydra models the input CCs into an LP. Using the CCs, the data space of the table is partitioned into regions using a novel *region-partitioning* algorithm. There is one variable for each region, corresponding to the number of tuples chosen from the region. Each CC is encoded as an LP constraint on these variables, and the solution of the LP is used in deciding which tuples to include in the table.

Our region-partitioning strategy is in marked contrast to the *grid-partitioning* strategy used in DataSynth. Grid-Partitioning first intervalizes the domain of each attribute based on the constants appearing in the CCs, and divides the domain into a grid aligned with the interval boundaries for each attribute. If the table has K attributes, and in the worst case an attribute gets divided into ℓ intervals, then the data space is partitioned into a grid of $\mathcal{O}(\ell^K)$ cells. For each cell in the grid, a variable is created that represents the number of data rows present in that cell. In contrast, our region-partitioning strategy divides the domain into only the number of regions required to precisely write out each CC, and assigns one variable to each region. This results in $\mathcal{O}(2^Q)$ variables, where Q is the number of CCs – this typically leads to far fewer variables than grid-partitioning.

To make the above concrete, consider the CCs c_1 and c_2 from our running example. Grid-partitioning divides the domain of the table as shown in Figure 4.1a. With a variable assigned to each grid cell, there is a total of 25 variables. In contrast, the region-partitioning strategy partitions the space into 4 regions as shown in Figure 4.1b, resulting in a tally of only 4 variables.

The CCs, expressed in terms of LP constraints, are shown below in Figure 4.2a and 4.2b for grid-partitioning and region-partitioning, respectively. In the former, variable x_i denotes the cardinality of cell i, while in the latter, variable x_j denotes the cardinality of region b_j.

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



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

Figure 4.1: Grid-Partitioning vs Region-Partitioning

$$x_{12} + x_{13} = 40000$$
$$x_8 + x_9 + x_{13} + x_{14} + x_{18} + x_{19} = 45000$$
$$x_1 + x_2 + ... + x_{25} = 50000$$

(a) **Grid-Partitioning**

$$x_1 + x_3 = 40000$$
$$x_2 + x_3 = 45000$$
$$x_1 + x_2 + x_3 + x_4 = 50000$$

(b) **Region-Partitioning**

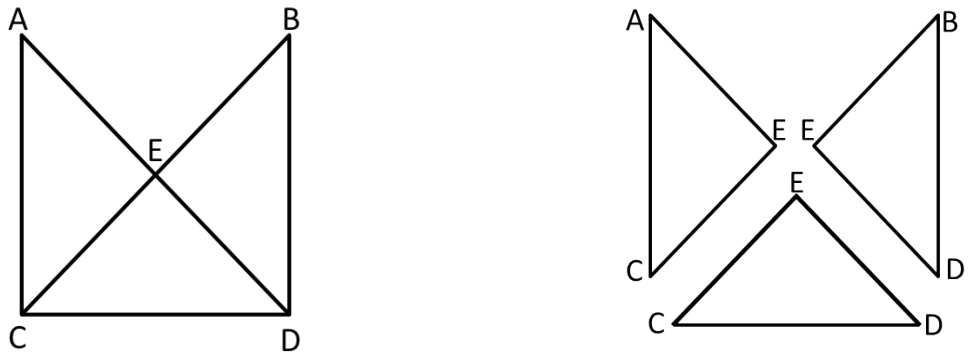
Figure 4.2: LP Constraints

4.3.2 Dimensionality Reduction

Since the intersections among CCs tend to increase in higher dimensional space, the LP complexity is also adversely affected due to exponential increase in the number of regions. To address this, the table is first decomposed into a set of *sub-tables* to reduce the intersections among CCs in lower dimensions. This effectively reduces the LP complexity. This optimization was proposed in [17]. We summarize the algorithm here: Construct a "table-graph" by first creating a node for each attribute, and then inserting an edge between a pair of nodes if the corresponding attributes appear together in one or more CCs. Further, additional edges are added (if required) to make the table-graph to be *chordal*[1], a property required to ensure acyclicity in the subsequent processing. Now, the sub-tables are identified as the *maximal cliques* in the table-graph.

For instance, the sub-tables of the table-graph in Figure 4.3a are AEC, BED and CED, as shown in Figure 4.3b.

[1]A chordal graph is one in which all cycles of four or more vertices have a chord.



(a) **Table Graph** (b) **Sub-Tables**

Figure 4.3: Table Decomposition

Once the sub-tables are identified, the region partitioning is executed on each sub-table. Since the sub-tables can have common attributes, additional consistency constraints are added in the LP to ensure the marginal distribution of tuples along these common columns are identical.

Since partitioning is carried out at a sub-table level, the LP solution, which is expressed in terms of sub-table variables, needs to be mapped to equivalents in the original table space. A sampling-based approach was proposed in [17] for this purpose – for example, say a table (A, B, C) is split into a pair of sub-tables (A, B) and (B, C), the algorithm computes the distributions $Prob(A, B)$ and $Prob(C|B)$. Then, each tuple is generated by first sampling a point from the former distribution, and then sampling a point from the latter conditioned on this outcome.

However, we have chosen not to take this approach since the computational overheads incurred are enormous, and the sampling process introduces errors in volumetric fidelity. Instead, we have designed and implemented an alternative data-scale free, deterministic align-merge algorithm, which produces an intermediate table summary in the output. This not only provides data scalability but also eliminates the accuracy errors incurred by the sampling strategy.

4.3.3 Summary Based Computation

Hydra produces a table summary in its output. An example summary for *Std* table, is shown in Figure 4.4.

The Tuple Generator module of Hydra resides in the database engine. It ensures that whenever a query is fired, data is not fetched from the disk but instead gets generated *on-demand*, using the table summary.

Age, GPA, Scholarship	NumTuples
15,8,4K	3000
15,8,1K	2000
20,6,8K	40000
20,5,8K	3000
35,5,12K	1000
35,5,35K	1000

Figure 4.4: Example Table Summary

4.4 LP Formulation

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

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

4.4.1 Mathematical Basis for LP Formulation

Let K denote the number of attributes in the given sub-table S, $dom(A_i)$ the domain of the ith attribute (A_i). So, $dom(A_1) \times dom(A_2) \times \cdots \times dom(A_K)$ represents the domain of the S. We represent this domain using the shorthand notation Ω.

We are given a set of Q CCs that are applicable on S. For $1 \leq j \leq Q$.

Simple LP Formulation Let us first consider a simple way of formulating an LP that encodes all CCs. For each tuple $t \in \Omega$, assign a variable x_t that denotes the number of copies of t in the sub-table S. Then, the LP formulation shown in Figure 4.5 ensures that a feasible solution satisfies all CCs, including a constraint on the total size of S.

The problem with this formulation is that the number of variables in the resulting LP is as large as the size of the universe Ω. Hence, it is infeasible to work directly with this formulation.

(1) For each $t \in \Omega, x_t \geq 0$

(2) $\left[\sum_{t \in \Omega} x_t\right] = |T|$

(3) For each $j, 1 \leq j \leq Q$, $\left[\sum_{t:\sigma_j(t)=true} x_t\right] = l_j$

Figure 4.5: Simple LP formulation

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

We hasten to add that the above LP construction process based on merging variables is only for illustrating the concept – the actual algorithm employed in our system *directly* derives the regions, as described in Section 4.4.2.

For constraint c and $t \in \Omega$, let $c(t)$ be an indicator variable:

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

Definition 4.1 *For a pair of points $t_1, t_2 \in \Omega$ and a set of constraints $\mathbb{C}$, we say $t_1 R t_2$ if for each $c \in \mathbb{C}$, $c(t_1) = c(t_2)$.*

Observation 4.1 *R is an equivalence relation on Ω.*

Proof: It can be easily seen that R is reflexive and symmetric. For transitivity, suppose that for $t_1, t_2, t_3 \in \Omega$, $t_1 R t_2$ and $t_2 R t_3$. Note that for each $c \in \mathbb{C}$, it must be true that $c(t_1) = c(t_2)$ and $c(t_2) = c(t_3)$. Therefore, it must be true that $c(t_1) = c(t_3)$ for each $c \in \mathbb{C}$, showing that the relation is transitive. □

A partition of Ω is a set of subsets of Ω such that every element $t \in \Omega$ is in exactly one of these subsets. The individual sets in a partition are called *filter-blocks*.

Definition 4.2 *A set of points b is said to be valid with respect to a set of constraints $\mathbb{C}$ if for any two points $t_1, t_2 \in b$, $t_1 R t_2$. Given a set of constraints $\mathbb{C}$, a partition $\mathbb{F}$ of Ω is said to be a* valid partition *if for each filter-block $b \in \mathbb{F}$, b is valid with respect to $\mathbb{C}$.*

In a valid partition of Ω with respect to $\mathbb{C}$, any pair of points within the same filter-block satisfy the same set of CCs. Once we obtain a valid partition $\mathbb{F}$, the LP can be re-formulated as shown in Figure 4.6. Instead of a variable for each point $t \in \Omega$, there is now a single variable x_b for each filter-block $b \in \mathbb{F}$ representing the number of tuples of the sub-table that are contained in b. Note that the tuples in a sub-table need not be unique, therefore x_b may include duplicates in its count.

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

(2) $\left[\sum_{b \in \mathbb{F}} x_b\right] = |T|$

(3) For each $j, 1 \leq j \leq Q$, $\left[\sum_{b:\sigma_j(b)=true} x_b\right] = l_j$

Figure 4.6: Reduced LP formulation

The total number of variables in the reduced LP shown in Figure 4.6 is equal to the number of filter-blocks in the partition $\mathbb{F}$ and is potentially much smaller than the number of variables in the original LP, shown in Figure 4.5. Since we desire an LP with the smallest number of variables, we look for a valid partition of Ω with the minimum number of filter-blocks. A valid partition with respect to $\mathbb{C}$ is an *optimal partition* if it has the smallest number of filter-blocks from among all valid partitions of Ω with respect to $\mathbb{C}$.

Lemma 4.1 *The quotient set of Ω by R is the (unique) optimal partition of Ω with respect to $\mathbb{C}$.*

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

[1]The quotient set is the set of equivalence classes resulting from R on Ω.

Suppose that $\mathbb{F}_1$ is not the unique optimal partition. Then, there must exist another valid partition $\mathbb{F}_2$ such that $\mathbb{F}_2 \neq \mathbb{F}_1$ and $|\mathbb{F}_2| \leq |\mathbb{F}_1|$. This implies that there exist two points $t_1, t_2 \in \Omega$ such that t_1 and t_2 are in different filter-blocks in $\mathbb{F}_1$, but in the same filter-block in $\mathbb{F}_2$. Since t_1 and t_2 belong to different filter-blocks in $\mathbb{F}_1$, it must be true that t_1 and t_2 are not related by R. But, in $\mathbb{F}_2$ points t_1 and t_2 belong to the same filter-block, which implies that $\mathbb{F}_2$ cannot be a valid partition, a contradiction. □

4.4.2 Deriving the Optimal Partition

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

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

is divided into the basic sub-constraints:

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

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

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

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

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

Lemma 4.2 *Given a set of DNF constraints $\mathbb{C}$, Algorithm 1 returns an optimal partition of Ω with respect to $\mathbb{C}$.*

Proof: As in the algorithm, let $\mathbb{C}'$ denote the set of sub-constraints resulting from constraints in $\mathbb{C}$. From Lemma 4.3, we know that $\mathbb{F}'$ is a valid partition with respect to $\mathbb{C}'$. Consider any filter-block $b \in \mathbb{F}'$. Since b is valid with respect to $\mathbb{C}'$, and each constraint in $\mathbb{C}'$ is stricter than a corresponding constraint in $\mathbb{C}$, b is valid with respect to $\mathbb{C}$. Hence, $\mathbb{F}'$ is a valid partition with respect to $\mathbb{C}$.

Next, consider that each filter-block b^* in $\mathbb{F}^*$ was obtained by merging filter-blocks in $\mathbb{F}'$ that have the same label. For any pair of points t_1, t_2 in b^*, it is true they satisfy the same set of constraints in $\mathbb{C}$, showing that $\mathbb{F}^*$ is a valid partition wrt $\mathbb{C}$. Also, any two filter-blocks in $\mathbb{F}^*$ have distinct labels (if they had the same label, they would have been merged). Therefore, we conclude using arguments similar to Lemma 4.1 that $\mathbb{F}^*$ is an optimal partition of Ω with respect to $\mathbb{C}$. □

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

Definition 4.3 *For a sub-constraint c and dimension i, let c^i denote the restriction (projection) of c to dimension i. Further, let $c_1^i = \bigwedge_{j=1\ldots i} c^j$ denote the restriction of c to dimensions $1, 2, \ldots, i$. For instance, if $c = (A_1 \geq 1) \wedge (A_2 \geq 4) \wedge (A_2 \leq 5) \wedge (A_3 > 6)$, then $c^2 = (A_2 \geq 4) \wedge (A_2 \leq 5)$, and $c_1^2 = (A_1 \geq 1) \wedge (A_2 \geq 4) \wedge (A_2 \leq 5)$. For convenience, if c does not have a constraint along dimension i, then c^i is defined to be "true".*

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

Definition 4.4 *A constraint c is said to split an filter-block $b \subseteq \Omega$ if there exist a pair of points $t_1, t_2 \in b$ such that $c(t_1) = true$ and $c(t_2) = false$. If c splits b, then refining b by c partitions b into two subsets $b^+(c) = \{t \in b | c(t) = true\}$ and $b^-(c) = \{t \in b | c(t) = false\}$.*

Lemma 4.3 *Given a set of sub-constraints $\mathbb{C}$, Algorithm 2 returns a valid partition of Ω with respect to $\mathbb{C}$.*

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

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

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

Proof: For $1 \leq i \leq K$, let $\mathbb{C}_1^i = \{c_1^i | c \in \mathbb{C}\}$. We show by induction on i that after the ith iteration of the outermost for loop in the algorithm, $\mathbb{F}^i$ contains a valid partition of Ω with respect to $\mathbb{C}_1^i$. Since $\mathbb{C}_1^K = \mathbb{C}$, it follows that after K iterations, $\mathbb{F}^K$ contains a valid partition of Ω with respect to $\mathbb{C}$. We consider $i = 0$ as the base case, and the set $\mathbb{C}_1^0$ as a set of "always true" constraints. Hence, $\mathbb{F}^0$, which consists of only one element, Ω, is a valid partition with respect to $\mathbb{C}_1^0$.

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

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

4.4.3 Consistency Constraints

Since different sub-tables can have common attribute(s), additional constraints need to be added to ensure that their distributions for the common attribute(s) are the same. In order to do so, we may need to further refine the partition generated from the above procedure. Specifically, consider a pair of sub-tables S_1 and S_2 with attribute sets $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, such that $\mathcal{A}_1 \cap \mathcal{A}_2 \neq \emptyset$. Let $\Omega^1 = \prod_{i \in \mathcal{A}_1} dom(A_i)$, and $\Omega^2 = \prod_{j \in \mathcal{A}_2} dom(A_j)$ be the corresponding domains for S_1 and S_2 respectively, and $\Omega^{1,2} = \prod_{i \in \mathcal{A}_1 \cap \mathcal{A}_2} dom(A_i)$. Let the partitions obtained on Ω^1 and Ω^2 be $\mathbb{F}_1$ and $\mathbb{F}_2$, respectively. In order to keep $\mathbb{F}_1$ and $\mathbb{F}_2$ consistent with each other, we need to ensure that their region boundaries are aligned with each other, and this is achieved by refining $\mathbb{F}_1$ and $\mathbb{F}_2$ so that they have common boundaries along dimensions $\mathcal{A}_1 \cap \mathcal{A}_2$. We consider the union of the "split points" of $\mathbb{F}_1$ and $\mathbb{F}_2$ along dimensions $\mathcal{A}_1 \cap \mathcal{A}_2$ and further for each filter-block in $\mathbb{F}_1$ (and $\mathbb{F}_2$), we refine this filter-block until it no longer crosses such a split point. Finally, we add LP constraints that equate distributions of the common attributes in $\mathbb{F}_1$ and $\mathbb{F}_2$.

4.5 Table Summary Construction

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

Recall that a variable in the LP represents an underlying filter-block in a sub-table's partition, and its assigned value is the number of rows present in that filter-block– this value is hereafter referred to generically as NumTuples. The collection of NumTuples values represent the sub-table solutions, and these solutions are integrated to obtain the solution for the complete table. This process can be enumerated in two steps: (a) Constructing a solution for complete tables, (b) Instantiating table summaries. Each of these are detailed next.

4.5.1 Constructing Solution for the Table

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

Algorithm 3: Table Solution Construction

```
1 𝕊 ← OrderSubTables(𝕊);
2 tableSol ← ∅;
3 foreach S ∈ 𝕊 do
4 |   tableSol, S ← Align(tableSol, S) ;
5 |   tableSol ← Merge(tableSol, S);
```

4.5.1.1 Sub-Table Ordering

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

For instance, in the example of Figure 4.3, there are three sub-tables: AEC, ECD and BED. Here, if we start with sub-table AEC as the first choice in the ordering sequence, then BED cannot be the next selection – this is because even after removing the common attribute E between BED and AEC, the edge DC continues to connect the two components. The sub-table ECD, on the other hand, satisfies the required condition, and is chosen to be the second in the order. After ECD is chosen, then BED follows as the last sub-table in the sequence.

4.5.1.2 Aligning

After obtaining the sub-table merge order as per above, in every iteration we merge the next sub-table solution (S) in the sequence to the current table-solution ($tableSol$), after a process of alignment. The alignment algorithm is a two step exercise, as shown in the example of Figure 4.7:

Solution Sorting: First, the $tableSol$ and S solutions are each sorted on their common set of attributes to facilitate direct comparison of their matching ranges. For instance, the solutions (Age, GPA) and $(Age, Scholarship)$ in Figure 4.7a are each sorted on the intervals enumerated in the common attribute Age.

Row Splitting: Our addition of consistency constraints during the LP formulation ensured that the distribution of tuples along the common set of attributes is the same in the various sub-tables. Therefore it easy to see that the sum of NumTuples values in any interval of the common attributes is the same for the sub-table solutions under alignment. For

Age	GPA	NumTuples
[15,20)	[8,10)	5000
[20,35)	[6,8)	40000
[20,35)	[5,6) U [8,9)	3000
[35,40)	[5,9)	2000

Age	Scholarship	NumTuples
[15,20)	[4K,12K)	3000
[15,20)	[1K,4K)	2000
[20,35)	[8K,35K)	43000
[35,40)	[12K,35K)	1000
[35,40)	[35K,70K)	1000

(a) **Sub-table Solution**

Age	GPA	NumTuples
[15,20)	[8,10)	3000
[15,20)	[8,10)	2000
[20,35)	[6,8)	40000
[20,35)	[5,6) U [8,9)	3000
[35,40)	[5,9)	1000
[35,40)	[5,9)	1000

Age	Scholarship	NumTuples
[15,20)	[4K,12K)	3000
[15,20)	[1K,4K)	2000
[20,35)	[8K,35K)	40000
[20,35)	[8K,35K)	3000
[35,40)	[12K,35K)	1000
[35,40)	[35K,70K)	1000

(b) **Table Alignment**

Age	GPA	Scholarship	NumTuples
[15,20)	[8,10)	[4K,12K)	3000
[15,20)	[8,10)	[1K,4K)	2000
[20,35)	[6,8)	[8K,35K)	40000
[20,35)	[5,6) U [8,9)	[8K,35K)	3000
[35,40)	[5,9)	[12K,35K)	1000
[35,40)	[5,9)	[35K,70K)	1000

(c) **Merged Table Solution**

Figure 4.7: Align and Merge Example

example, in Figure 4.7a, the total number of tuples with $Age = [15, 20)$ is 5000 in both the (Age, GPA) and $(Age, Scholarship)$ solutions. Likewise, the other entries in column Age also have matching total number of tuples across the solutions. The align step *splits* the rows in these solutions such that the corresponding rows in both solutions have the same number of tuples. The sub-table solutions of Figure 4.7a are shown in Figure 4.7b after undergoing the alignment process, with both solutions now having identical NumTuples

in the corresponding rows.

4.5.1.3 Merging

This is the last step in the construction of the table solution. Here we simply merge the two solutions obtained after alignment through a "position" based join, where the physically corresponding rows in each solution are combined, with the common attributes being represented once. For example, the aligned solutions of Figure 4.7b are merge-joined using the positions (or row identifiers) to deliver the final table solution of Figure 4.7c.

As discussed earlier, DataSynth adopted a sampling algorithm for constructing the table solutions post LP solving. In marked contrast, Hydra *deterministically* generates the table solutions, facilitating us to operate purely in the summary space. There are two tangible benefits of this deterministic strategy: (a) elimination of the time and space overheads due to sampling, and (b) elimination of sampling-based errors in satisfying CCs.

4.5.2 Instantiating Table Summaries

As shown in Figure 4.7c, each row in the table solution is comprised of a series of intervals (across various attributes) and the number of tuples in the region represented by these intervals. We now need to decide as to how these tuples are distributed *within* the attribute intervals. Our current solution is very simple: Assign the *entire* cardinality to the *left boundaries* of the intervals. For example, the first row in Figure 4.7c would result in generation of 3000 tuples all having $Age = 15, GPA = 8, Scholarship = 4K$ values.

The schema of the summary s^T of a table T has all the non-key columns of T and additionally has the NumTuples entry denoting the number of rows with that value combination. A sample table summary was previously shown in Figure 4.4.

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

4.6 Tuple Generation

The Tuple Generator component resides inside the database engine, and needs to be explicitly incorporated in the engine codebase by the vendor. Whenever this feature is enabled for a table, the scan operator for that table is replaced with the dynamic generation operator. As a result, during query execution, the executor does not fetch the data from the disk but is instead supplied by the Tuple Generator in an *on-demand* manner, using the available table summary.

Each row in the table summary has a value combination and an associated NumTuples entry. We consider the PK values to be the row numbers of the table. Therefore, to get the ith tuple of a table T, the PK is chosen as i and the rest of the attributes come from the table summary. We iterate over the rows of s^T and take the cumulative sum of the NumTuples entries until the sum exceeds i. Say the summation crosses the value i in jth row of $s(T)$. Then the rest of the values of the ith tuple are assigned to be precisely the same as those present in the jth row of s^T. For example, the 5100th row of table *Std* in Figure 4.4, would be $\langle 5100, 20, 6, 8K \rangle$.

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

4.7 Like Predicates

LIKE predicate constants are typically expressed as *regular expressions*. There are two wildcards often used in conjunction with the LIKE operator:

- The percent sign (%) represents an arbitrary number, including zero, characters.
- The underscore sign (_) represents a single character.

To handle LIKE predicates, we begin by collecting all the predicate constants that occur with the LIKE operator. This collection is segregated on the basis of the attribute on which they are applied. Depending on the number of expressions associated with an attribute, the handling mechanisms are classified as follows.

Single LIKE Predicate. Consider an attribute that has only one LIKE predicate applied on it in the entire workload. We take the regular expression and generate any valid representative string that satisfies the regular expression, and with that, we convert the LIKE predicate into an equality filter predicate.

For example, consider the following CC on an attribute A from table T:

$$|\sigma_{A\sim\text{`\_a\_\%'}}(T)| = 4$$

Here the $\sim$ symbol represents the LIKE predicate.

The LIKE predicate is applied on A, and the regular expression is '_a_%'. 'aaa' is a valid representative string that satisfies the regular expression. Therefore, the transformed CC is:

$$|\sigma_{A=\text{`aaa'}}(T)| = 4$$

Now, this modified CC can be handled in the same way in which other equality filter predicate CCs are handled with the existing setup.

Multiple LIKE Predicates. Now, consider an attribute having multiple LIKE predicates applied upon it in the workload. The data generation mechanism in such cases needs to be aware of any possible intersections among the different predicates. To make the above concrete, consider the following three CCs:

$$|\sigma_{A\sim\text{'\%b\%'}}(T)|= 8, \quad |\sigma_{A\sim\text{'\%bc\%'}}(T)|= 4, \quad |\sigma_{A\sim\text{'\%c\%'}}(T)|= 7$$

Note that here we cannot transform the CCs in the similar way with an equality representative string. This is because regular expressions corresponding to different CCs may have an intersection. For instance, a tuple 'bc' would satisfy both '%b%' and '%bc%' regular expressions. Hence, direct transformation can cause infeasible constraint-set. Therefore, we need to do partitioning of data space into blocks such that that regular expressions attached to any two blocks do not intersect. Once, the partitioning is done, the input regular expressions in the CCs are transformed into disjunctive equality predicates. We discuss each of these steps in detail next.

4.7.1 Partioning using Regular Expressions

To do the partitioning, we first construct a Venn diagram that captures intersections of the regular expressions. For example, the Venn diagram for three regular expressions from our running example is shown in Figure 4.8.

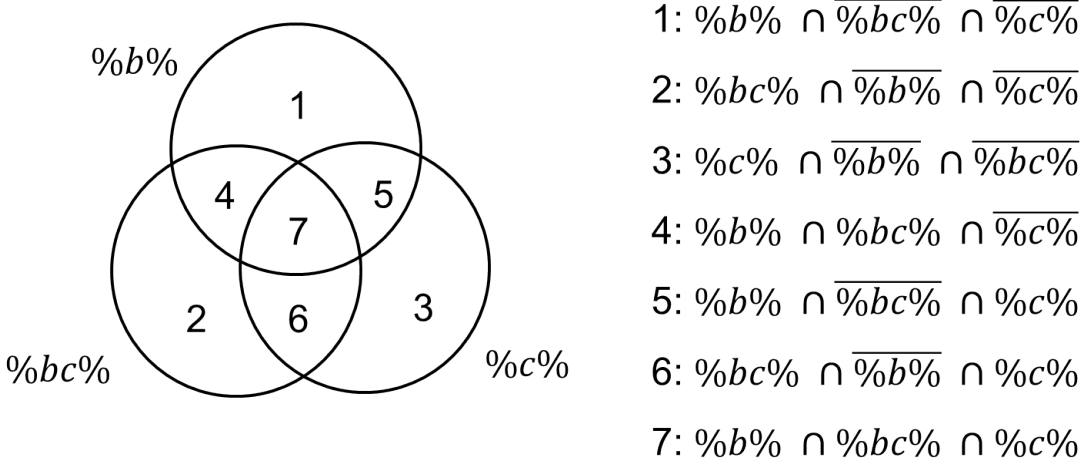


Figure 4.8: Venn diagram showing all disjoint spaces

The diagram features *intersections* and *complements* of regular expressions. To perform intersection and complement on regular expression strings, we use concepts from *automata theory*. First, we convert each regular expression to a DFA. This is done by following the sequence: **Regular Expression** $\rightarrow$ **Epsilon-NFA** $\rightarrow$ **NFA** $\rightarrow$ **DFA**. Note that DFAs are closed under intersection and complement. Therefore, once we have equivalent DFAs for the regular expressions, we can perform these operations on the DFAs easily.

From the Venn diagram in Figure 4.8 we can see that there are seven disjoint regions. Some of these regions can be empty, that is their corresponding final DFA results in NULL. For example, regions 2, 4, 6 in the Venn diagram shown above correspond to empty regions. For each of the non-empty regions, we generate a representative string. For example, the representative strings for the populated regions in our running example are as follows:

1: 'b', 3: 'c', 5: 'cb', 7: 'bc'

Optimization. The intersection of two DFAs can result in a large number of states in the resultant DFA. This can make the subsequent intersection computation complex. Hence as an optimization measure, we minimize the resultant DFA after each individual intersection and then proceed with subsequent intersections. Specifically, once we get a NULL for an intersection, its subsequent intersections with other regular expressions are avoided.

4.7.2 Predicate Transformation

After obtaining representative strings for each populated region of the Venn diagram, we rewrite each input regular expressions in the CCs as a disjunction of equality predicates. Specifically, for a regular expression, the disjunctive equality predicates are with respect to the representative strings associated with its constituent regions of the Venn diagrams. We show the transformed CCs for our running example in Table 4.3.

Original CC	Transformed CC
$\lvert\sigma_{A\sim'\%b\%'}(T)\rvert = 8$	$\lvert\sigma_{A\in\{'b','bc','cb'\}}(T)\rvert = 8$
$\lvert\sigma_{A\sim'\%bc\%'}(T)\rvert = 4$	$\lvert\sigma_{A\in\{'bc'\}}(T)\rvert = 4$
$\lvert\sigma_{A\sim'\%c\%'}(T)\rvert = 7$	$\lvert\sigma_{A\in\{'c','bc','cb'\}}(T)\rvert = 7$

Table 4.3: Transformation of Regular Expression CCs

Once this transformation is done, the CCs can be handled like any CC having disjunction with equality predicates.

4.8 Experimental Evaluation

We have implemented the Hydra design, described in the previous sections, in a Java tool. Z3 [14] solver is leveraged to compute solutions for the LP formulations. In this section, we evaluate Hydra's empirical performance, using our implementation of DataSynth as the comparative yardstick in the analysis. For a fair comparison of Hydra's performance against DataSynth, we extended the implementation of DataSynth to also handle constraints in DNF.

Database Environment. The TPC-DS [12] decision-support benchmark database, with a default size of 100GB, is used as the baseline in our experiments. The database is hosted on a PostgreSQL v9.6 engine [7] with the hardware platform being a vanilla HP Z440 workstation.

Workload Construction. A complex query workload, WL_c, featuring 351 filter CCs was created by customizing the 99 queries of the benchmark. The constructed CCs were rewritten as a filter predicate on denormalized tables. Hence, in the output, denormalized tables were constructed. This was done to stress test the system by (a) including more CCs per table; (b) dealing with high dimensional tables. The distribution of the cardinalities for these CCs are shown in Figure 4.9, with the cardinalities measured on a log-scale. The figure clearly indicates that a wide range of cardinalities are present in the constraints, going from a few tuples to almost a billion.

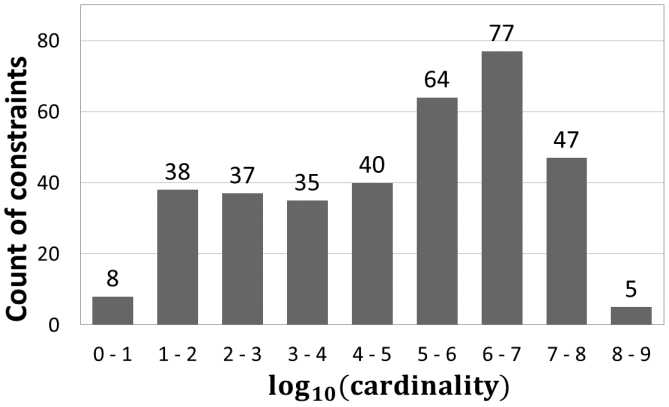


Figure 4.9: Distribution of Cardinality in CCs (WL_c)

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

wherein the variables created by DataSynth were less than a million, and therefore well within the solver's processing power.

4.8.1 Constraint Accuracy

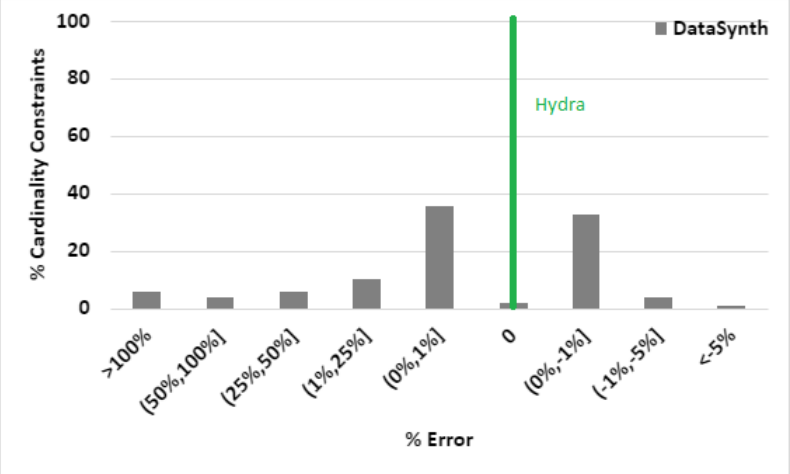


Figure 4.10: Quality of Volumetric Similarity (WL_s)

We begin by investigating how closely the volumetric similarity, with regard to operator output cardinalities, is achieved between the client and vendor sites for the WL_s workload by the Hydra and DataSynth regenerators. This behavior is captured in Figure 4.10, which plots the percentage of CCs that are within a given error percentage of volumetric similarity. From the plot it is evident that Hydra satisfies all the CCs with no error. This is in contrast to DataSynth, which satisfies around 80 percent of the CCs with minor error, but then incurs high error to achieve complete coverage of the remaining CCs.

As a final observation, it is interesting to note that DataSynth has to contend with both *negative* (volumes less than desired) and *positive* (volume greater than desired) errors, due to its sampling strategy – in fact, about one-third of the CCs suffered negative errors. From a practical standpoint, it is especially undesirable to have negative error because these do not induce required stress on the data processing elements in the engine.

4.8.2 Scalability with Workload Complexity

We now turn our attention to evaluating the complexity of the underlying LP that is formulated by Hydra and DataSynth. Since LP complexity is essentially proportional to the number of variables in the problem, we compare this number for the two techniques.

The number of LP variables for a representative set of TPC-DS tables, including the major fact and dimension tables (catalog_sales, store_sales, item) is captured, on a log-scale, in Fig-

ure 4.11 for the WL_c complex workload. We observe here that the LPs formulated using the region-partitioning strategy in Hydra generate *several orders of magnitude* fewer variables than the corresponding LPs derived from the grid-partitioning in DataSynth. As a case in point, consider the catalog_sales table – the number of variables created by DataSynth was almost 5.5 million, which is reduced to as low as 1620 by Hydra. Even more dramatic is the change for item table, where the number of variables is reduced from an enormous 10^{11} to around 3700.

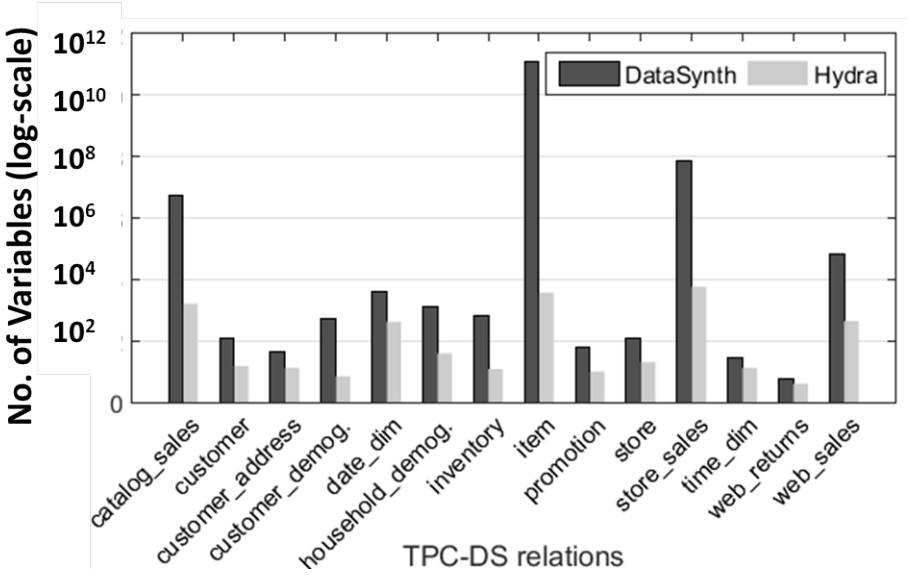


Figure 4.11: Number of variables in the LP (WL_c)

From an absolute perspective also, the large number of variables created by DataSynth is a critical problem since, as mentioned previously, the LP solver crashed in handling these cases. In marked contrast, the few thousands of LP variables generated by Hydra were easily solvable in less than a minute. Moreover, even when we switched to the simple workload, WL_s, the LP solution time for DataSynth was almost an hour, whereas Hydra completed in a few seconds as shown in Figure 4.12.

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

Figure 4.12: LP Processing Time

4.8.3 Scalability with Materialized Data Size

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

The experimental results are shown in Figure 4.13, where we also present, for comparative purposes, the performance with 10 GB and 1000 GB databases, apart from the default 100 GB database. We see here that there is a huge reduction in the materialization time of Hydra at all scales. Further, even in absolute terms, Hydra is able to output a 100 GB database in around 11 minutes, whereas DataSynth takes 42 hours to complete the same task.

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

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

Figure 4.13: Data Materialization Time

4.8.4 Scalability to Big Data Volumes

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

4.8.5 Dynamism in Data Generation

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

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

Figure 4.14: Data Supply Times

The results of this experiment are shown in Figure 4.14 for the five biggest tables in the 100 GB database instance. We see here that the tuple generator is not only competitive with a materialized solution, but is in fact typically *faster*. Therefore, using dynamic generation can prove to be a good option since it can help to eliminate the large time and space overheads incurred in: (1) dumping generated data on the disk, and (2) loading the data on the engine under test.

4.8.6 Performance on JOB Benchmark

A legitimate concern with regard to the above encouraging results for Hydra is that they may be an artifact of the TPC-DS database, and perhaps might under-perform on other datasets. To address this concern, we consider in our final experiment, a schematically highly different database, namely the JOB benchmark [45, 5], which is based on the IMDB real-world dataset. Here, we created a workload of 523 CCs, whose cardinality distribution is again highly varied as seen in Figure 4.15.

We found that Hydra efficiently solved this workload as well, with the number of variables in each table being typically in the few thousands, and never exceeding a hundred thousand, as shown quantitatively in Figure 4.16. The overall table summaries were quickly generated in around 20 seconds while ensuring high fidelity.

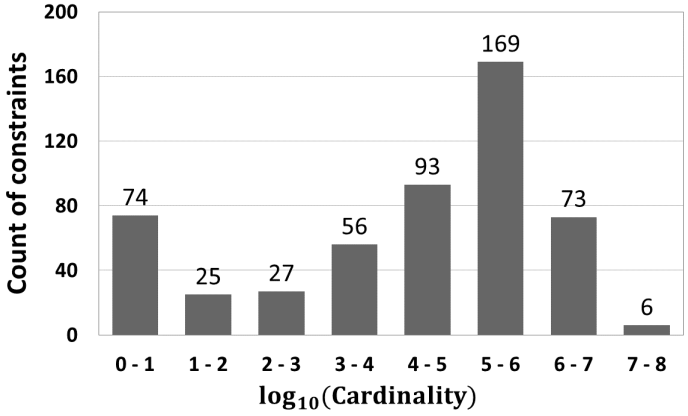


Figure 4.15: Cardinality distribution of CCs in JOB

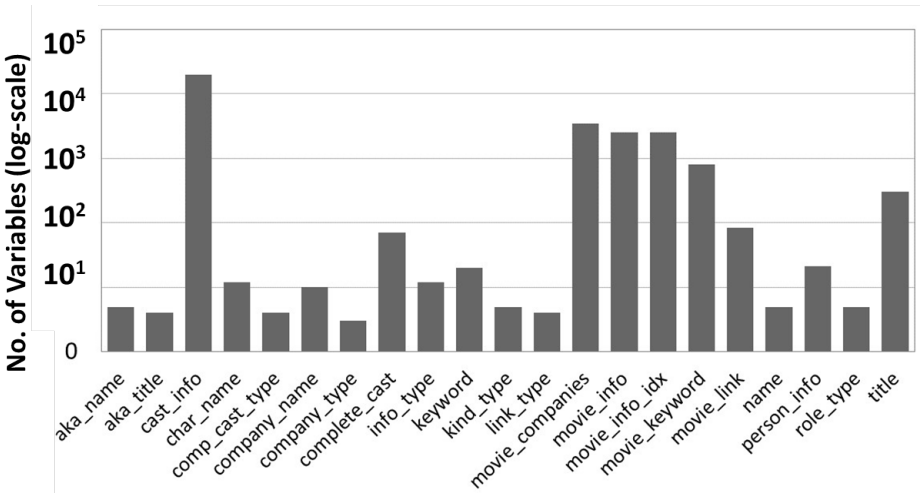


Figure 4.16: Number of Variables for JOB

4.9 Conclusion

In this chapter we discussed the mechanism for regenerating synthetic table using filter constraints. Specifically, by reworking the basic LP problem formulation into a region-based variable assignment, Hydra improves on the state-of-the-art DataSynth's performance by orders of magnitude with regard to problem complexity, data materialization time, and scalability to large volumes. Secondly, by using a deterministic alignment technique for database consistency, it provides far better accuracy in meeting volumetric constraints as compared to the probabilistic approach employed in DataSynth. Finally, its summary-based framework organically supports the dynamic regeneration of streaming data sources, an essential prerequisite for efficiently testing contemporary deployments.

Chapter 5

Regeneration using Projection Constraints

5.1 Introduction

This chapter discusses the mechanism for synthesizing a table using AQPs including projection operator. 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 [11], as many as 16 feature the projection operation.

5.1.1 Projection-inclusive Constraints

We represent an AQP including a filter and projection on a table, using a projection-inclusive cardinality constraint (PIC). To express a PIC c on a table 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 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.

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). In the context of volumetric similarity, Group By and Union clause behaves same as Distinct SQL operation. Distinct is expressed as duplicate eliminating projection relational algebra operation and therefore we have used projection as an umbrella term to include Group By and Union as well.

As a sample instance, consider the following set of PICs on *Std* (*RollNo*, *Age*, *GPA*,

$Scholarship$):

$$c_1 : \langle f_1, GPA, 30000, \mathbf{5} \rangle \mid f_1 = (Age < 30) \wedge (5 \leq GPA < 8)$$
$$c_2 : \langle f_2, GPA, 4000, \mathbf{3} \rangle \mid f_2 = (Age \geq 30) \wedge (4 \leq GPA < 9)$$
$$c_3 : \langle f_3, Age, 48000, \mathbf{9} \rangle \mid f_3 = (Age \geq 15)$$

Here, PIC c_1 denotes that applying the f_1 predicate on Std should produce 30000 rows in the output, which is further reduced to 5 rows after projecting on the GPA column; the other PICs can be interpreted analogously.

5.1.2 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 [67], 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 Std table:

$t_1 : (RollNo = 10001,\ GPA = 7,\ Age = 25,\ Scholarship = 20K)$, and

$t_2 : (RollNo = 10002,\ GPA = 7,\ Age = 20,\ Scholarship = 30K)$.

Both rows satisfy the filter f_1, but the union of their projections along GPA yields a single outcome – namely, $GPA = 7$.

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 Std, satisfying the following disjunctive filter condition:

$$b = \{t \in Std \mid (t.Age \geq 30 \wedge t.GPA \geq 9) \vee (15 \leq t.Age < 30 \wedge t.GPA \geq 8)\}$$

Here, a tuple with $GPA = 8.5$ and $Age = 35$ can belong to both $\pi_{GPA}(b)$ and $\pi_{Age}(b)$, 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 [44]. Hence, alternative methods are needed to address this issue.

5.1.3 Our Contributions

In this chapter, we present the strategy 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, the concept of dynamic data regeneration is leveraged to 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 Hydra accurately and efficiently models Projection outcomes. As a case in point, for a workload of PICs, comprising over a hundred PICs in total, Hydra generated data that satisfied all the PICs, with *perfect accuracy*. Moreover, the entire summary production pipeline

completed within viable time and space overheads.

Note: If we want to add cardinality constraint on a group (output of Group By), that can be easily done by applying a filter on the group key. For instance, consider the following constraint: *for the group with column-set* $\mathbb{A}$ *having value* a*, the total number of rows is equal to* γ. This constraint on the group can be modeled as: $|\sigma_{\mathbb{A}=a}(T)|= \gamma$. This is the standard filter constraint considered in the previous chapter. Therefore, using the techniques described for filter constraints, these cardinality constraints per group can also be handled.

5.1.4 Organization

The remainder of this chapter is organized as follows: The problem framework is discussed in Section 5.2. Further, the key design principles of the proposed solution are introduced in Section 5.3, and then described in detail in Sections 5.4 through 5.7. The end-to-end implementation pipeline is presented in Section 5.8 with the critique in Section 5.9. The experimental evaluation is reported in Section 5.10. Finally, we conclude this chapter in Section 5.11.

5.2 Problem Framework

In this section, we summarize the basic problem statement, the underlying assumptions, the output delivered, and a tabulation of the notations used in this chapter.

5.2.1 Problem Statement

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

5.2.2 Assumptions

We assume that each PIC in $\mathcal{Q}$ is of the form described in the Introduction. Further, for simplicity, we assume that $\mathcal{Q}$ is collectively feasible, that is, there exists at least one legal database instance satisfying all the constraints. Finally, for brevity, we present the ideas using tables with columns having float data type; the extension to other data types is straightforward.

5.2.3 Output

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

5.2.4 Notations

The main acronyms and key notations used in the rest of this chapter are summarized in Tables 5.1 and 5.2, respectively.

Table 5.1: Acronyms

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

5.3 Design Principles

In this section we overview the core Hydra design principles, with the *Std* table of the Introduction used as the running example to explain their impact. Subsequently, in Sections 5.4 through 5.7, each principle is described in detail. To set the stage, here are some basic definitions underlying our work.

Definition 5.1 *A block is a bag of points (i.e. tuples) in the data space $\mathbb{D}(T)$ of the synthetic table T.*

Definition 5.2 *A projection block is a subset of points from $\mathbb{D}^{\mathbb{A}}(T)$, where $\mathbb{D}^{\mathbb{A}}(T)$ represents the data subspace of the synthetic table T spanned by a given PAS $\mathbb{A}$.*

Since this chapter focuses deals with a single table, we denote $\mathbb{D}(T)$ and $\mathbb{D}^{\mathbb{A}}(T)$ as simply $\mathbb{D}$ and $\mathbb{D}^{\mathbb{A}}$, respectively.

5.3.1 Region Partitioning

To model the filter predicates associated with $\mathcal{Q}$, the data space $\mathbb{D}$ is logically partitioned into a set of blocks. For this, we leverage the *region partitioning* technique from Chapter 4, which partitions the data space into the minimum number of blocks.

Each resultant block is referred to as a *filter-block*. The algorithm outputs the domain of each filter-block, which forms its logical condition. The domain of an filter-block b is denoted as $dom(b)$.

Consider the three filter predicates, f_1, f_2, f_3 on *Std* from Section 5.1.1.

Table 5.2: Notations

(a) Database Related

Symbol	Meaning
$\mathcal{S}$	Table Schema
T	Output Table
t	Tuple
s^T	Summary of T
$\mathbb{D}$	Data space of T
$\mathbb{D}^{\mathbb{A}}$	Data subspace spanned by $\mathbb{A}$

(b) Workload Related

Symbol	Meaning
$\mathcal{Q}$	Set of PICs
c	A PIC
f	Filter Predicate
$\mathbb{A}$	PAS
l	Output row card. after filter
k	Output row card. after projection
$\mathcal{Q}^c$	A compatible PICs Workload
β	Number of PASs across $\mathcal{Q}^c$

(c) Block Related

Symbol	Meaning
b	filter-block
r	refined-block
$\mathbb{R}$	Set of refined-blocks
$\overline{r}$	PRB wrt r and a PAS
$\overline{\mathbb{R}}^{\mathbb{A}}$	Set of PRBs for $\mathbb{A}$
m	Cardinality of $\overline{\mathbb{R}}^{\mathbb{A}}$
p	CPB
$\mathbb{P}^{\mathbb{A}}$	Set of CPBs for $\mathbb{A}$
x_r	variable for $\lvert r \rvert$
y_p	variable for $\lvert p \rvert$

(d) Relation/Function Related

Symbol	Meaning
$U(T)$	Set of attributes in T
$dom(.)$	Domain of the input parameter
M	A relation btw PICs and refined-blocks
$L^{\mathbb{A}}$	A relation btw CPBs and PRBs

For simplicity, Figure 5.1 shows only the 2D data space comprising the *Age* and *GPA* 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 filter-blocks: b_1, b_2, b_3, b_4 whose domains are depicted with dashed-line boundaries.

5.3.2 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 filter-block into a set of disjoint *refined-blocks* such that each resultant refined-block exhibits translation symmetry along each applicable projection subspace. That is, for each domain point of a refined-block r along a particular PAS, the projection of r along the remaining attributes is identical.

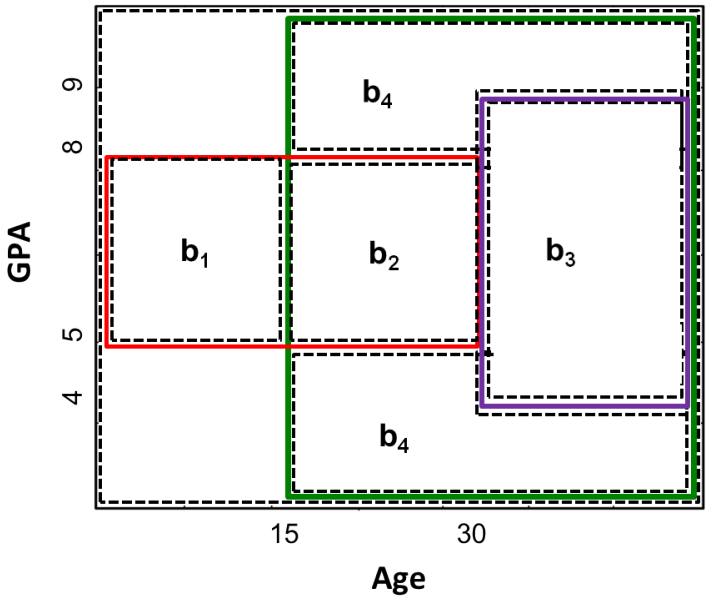
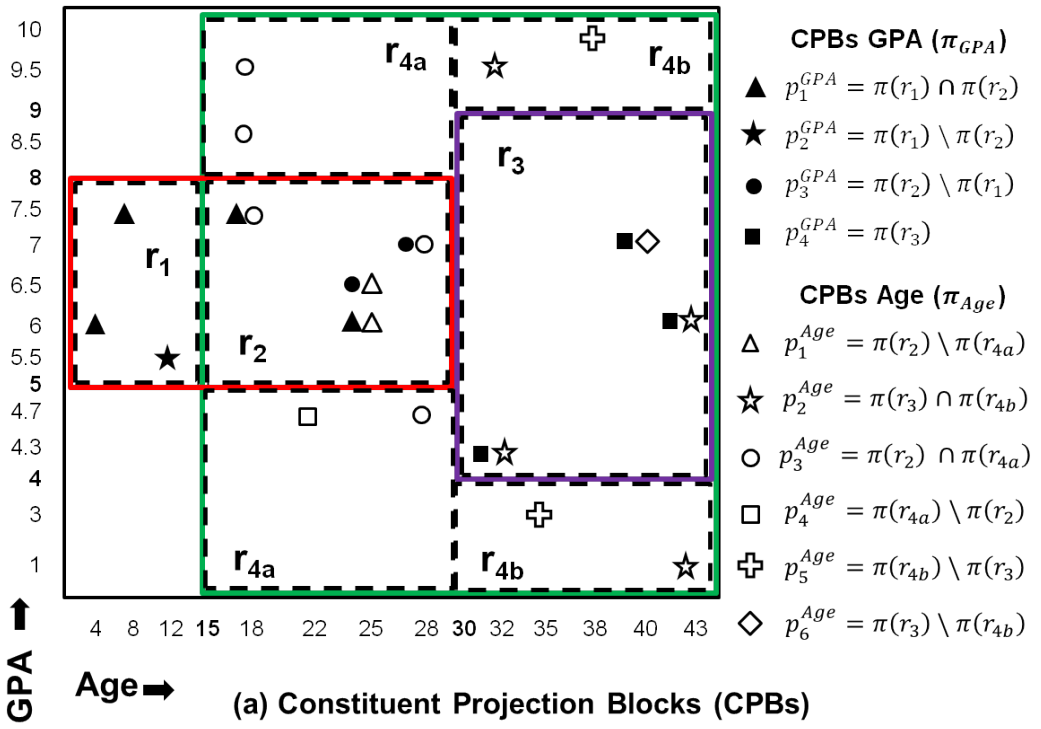


Figure 5.1: Region Partitioning

For instance, consider filter-block b_4 in Figure 5.1. Clearly, it is asymmetric along the PAS Age – specifically, compare the spatial layout in the range $15 \leq Age < 30$ with that in $Age \geq 30$. After refinement, this block breaks into r_{4a} and r_{4b} as shown in Figure 5.2(a) – it is easy to see that r_{4a} and r_{4b} are symmetric. (The other filter-blocks (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 5.2(a)). This refinement allows for the values along different projection subspaces to be generated independently. That is, $dom(r) = dom(\pi_{GPA}(r)) \times dom(\pi_{Age}(r))$, for each refined-block r.

The above refinement, however, does not scale when the projections applied on an filter-block 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.



(a) Constituent Projection Blocks (CPBs)

GPA	6	7.5	5.5	7.5	7	6.5	6	9.5	8.5
Age	4	8	12	18	28	25	25	18	18

GPA	4.7	4.7	9.5	10	3	1	4.3	6	7
Age	22	28	32	38	35	43	32	43	40

(b) Sample Student Table (Distinct Rows)

Figure 5.2: Symmetric Refinement and PSD

5.3.3 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, Hydra divides the data subspace associated with the GPA dimension into 4 CPBs: $p_1^{GPA}, p_2^{GPA}, p_3^{GPA}, p_4^{GPA}$, and the Age dimension subspace into 6 CPBs: $p_1^{Age}, p_2^{Age}, ..., p_6^{Age}$, as shown in Figure 5.2(a). Each CPB has a semantic meaning associated with it. For example, p_1^{GPA} semantically represents the GPA values present in both r_1 and r_2.

Further, the CPBs need not be mutually disjoint, as in the case of p_3^{GPA} and p_4^{GPA}. Finally, Figure 5.2(a) also shows the unique tuples enumerated by the sample output table shown in Figure 5.2(b), and the CPB (s) to which each of these tuples belongs.

5.3.4 Constraints Formulation

The LP solving procedure is constructed using variables representing the row cardinalities of refined-blocks and CPBs. For instance, if x_i represents the cardinality of refined-block r_i, and y_i^{GPA} and y_i^{Age} represent the cardinalities of CPBs p_i^{GPA} and p_i^{Age}, respectively, then PICs are expressed by linear equations as follows:

$$
\begin{aligned}
c_1: &\quad x_1 + x_2 = 30000, \quad y_1^{GPA} + y_2^{GPA} + y_3^{GPA} = 5 \\
c_2: &\quad x_3 = 4000, \quad y_4^{GPA} = 3 \\
c_3: &\quad x_2 + x_3 + x_{4a} + x_{4b} = 48000, \\
&\quad y_1^{Age} + y_2^{Age} + y_3^{Age} + y_4^{Age} + y_5^{Age} + y_6^{Age} = 9
\end{aligned}
$$

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

A sample solution to the above LP is shown below:

$$
\begin{aligned}
&x_1 = 30000, \quad x_2 = 0, \quad x_3 = 4000, \quad x_{4a} = 0, \quad x_{4b} = 14000 \\
&y_1^{GPA} = 0, \quad y_2^{GPA} = 5, \quad y_3^{GPA} = 0, \quad y_4^{GPA} = 3, \quad y_1^{Age} = 0 \\
&y_2^{Age} = 5, \quad y_3^{Age} = 0, \quad y_4^{Age} = 0, \quad y_5^{Age} = 0, \quad y_6^{Age} = 4
\end{aligned}
$$

5.3.5 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 *Std* table with respect to the aforementioned LP solution is shown in Figure 5.3, after incorporating an additional attribute *Scholarship* to illustrate a multi-dimensional projection.

Each segment of the summary corresponds to a populated refined-block. Specifically, the figure shows the tabulation for the r_1, r_3 and r_{4b} refined-blocks. Each tabulation comprises of a column for each PAS acting on the refined-block, and an additional last column indicating the total number of tuples present in the refined-block. 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 30000 tuples, such that there are 5 distinct values of GPA in the interval `[5,8)`, and 20 distinct value pairs of $\{Age, Scholarship\}$ of which 12 are from the 2D interval `[1,15), [8K,20K)`, and the remaining 8 from the 2D interval `[1,15), [25K,35K)`."

r_1

GPA	Age, Scholarship	#Tuples
[5, 8): 5	(A) [1, 15), (S) [8K,20K): 12 (A) [1, 15), (S) [25K, 35K): 08	30000

r_3

GPA	Age	Scholarship	#Tuples
[4, 9): 3	[30,37): 5 [37, 50): 4	[35K,70K)	4000

r_{4b}

Age	GPA, Scholarship	#Tuples
[30, 37): 5	(G) [1,4) U [9,10), (S) [31K, 35K): 6	14000

Figure 5.3: Table Summary Featuring Projection

For attributes that do not feature in any projection subspace, no associated distinct cardinality is maintained – an example is $Scholarship$ in r_3. Lastly, the primary-key column ($RollNo$ 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 $\{GPA, Scholarship\}$ points in r_{4b} are sourced from two separate intervals: `[1,4)` and `[9,10)` for GPA 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 the pure filter case. The key difference lies in that earlier we neither maintained intervals nor distinct value counts. Also, since only filter-blocks were maintained, which being inherently disjoint have no dependency among them. Here, we need to handle the dependencies enforced due to common CPBs being shared between refined-blocks.

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

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

5.4 Isolating Projections

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

Definition 5.3 *A block r in the data space of a table T with set of dimensions $U(T)$ is symmetric along a PAS $\mathbb{A}$ iff*

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

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

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

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

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 filter-block 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 5.4.1.

Impact of Overlapping Projection Subspaces. When partially overlapping PASs, say $\mathbb{A}_1$ and $\mathbb{A}_2$, are applied on an filter-block 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 $dom(\mathbb{A}_1 \cap \mathbb{A}_2)$. This is easily done by enumeration for small cardinality domains, but does not scale in general. Hence, in Hydra we bypass such overlapping projection operations by ensuring, as described in Section 5.4.2, that the input workload is initially itself decomposed such that there are no projection subspace overlaps in the resulting sub-workloads.

5.4.1 Symmetric Refinement

The refinement for each filter-block is done independently. Given an filter-block b and its associated PASs, this module refines b into a group of refined-blocks, such that each refined-block is symmetric along the input PASs.

Let us first understand the refinement procedure for an filter-block 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 $dom(\mathbb{A})$ 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 $I \in \mathbb{I}$, let b_I denote the part of b whose projection along $\mathbb{A}$ is I.

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

3. A hashmap is created with keys as $\pi(b_I)$ and value as I. Hence, the parts of b where the projection of b along $U(T) \setminus \mathbb{A}$ 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 $U(T) \setminus \mathbb{A}$ subspaces.

4. Each entry e in the hashmap corresponds to an refined-block, constructed by taking the region stored as key in e for the $U(T) \setminus \mathbb{A}$ 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. The domain of b along $\mathbb{A}$ be denoted as $dom(\pi_{\mathbb{A}}(b))$. Further, let E be a relation associated with the points in $dom(\pi_{\mathbb{A}}(b))$. For a pair of points $t_1, t_2 \in dom(\pi_{\mathbb{A}}(b))$, we say $t_1 E t_2$ iff the projection t_1 and t_2 along the rest of the attributes i.e. $U(T) \setminus \mathbb{A}$ is identical. It is easy to verify that E forms an *equivalence relation*. For an equivalence relation, the *quotient set* of the relation gives the minimum partition.

Lemma 5.1 *The Symmetric Refinement algorithm returns the quotient set of* $dom(\pi_{\mathbb{A}}(b))$ *by* E.

The proof follows from the fact that Symmetric Refinement algorithm uses a hashmap, which enables grouping of points in $dom(\pi_{\mathbb{A}}(b))$ together such that their projection on $U(T) \setminus \mathbb{A}$ are identical. Hence, for a PAS, the symmetric refinement algorithm produces the quotient set of E, 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 $\pi_{U(T)\setminus(\mathbb{A}_1 \cup \mathbb{A}_2 \cup ... \cup \mathbb{A}_\alpha)}(b)$. 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 refined-blocks, and usually more overlaps of refined-blocks 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 Hydra: The subspace having the maximum filter-block overlaps with b is chosen as the next subspace to be refined in the iterative sequence.

Mapping refined-blocks to PICs

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

Definition 5.4 *An refined-block $r \in \mathbb{R}$ is related by relation M to a PIC c containing filter predicate f, iff $dom(r)$ satisfies f. That is,*

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

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

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

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

5.4.2 Workload Decomposition

As discussed previously, symmetric refinement is performed when distinct PASs applicable on an filter-block 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 *conflicting pair* of PICs.

Definition 5.5 *A pair of PICs ($c_1 : \langle f_1, \mathbb{A}_1, l_1, k_1 \rangle$, $c_2 : \langle f_2, \mathbb{A}_2, l_2, k_2 \rangle$) 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 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 *Std* table:

$$c_4 : \langle GPA \leq 8 \wedge Scholarship \geq 40K, (Age, Scholarship), 500, 20 \rangle$$
$$c_5 : \langle Age \geq 10 \wedge Scholarship \leq 4K, (GPA, Scholarship), 2000, 6 \rangle$$

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 conflicting pairs of PICs in the resulting sub-workloads. We refer to a workload with no conflicting pairs as a *compatible* workload, and denote it using $\mathcal{Q}^c$.

Given $\mathcal{Q}$, the set of conflicting pairs is computed first. Subsequently, we construct the set of compatible sub-workloads that cover the entire workload. Additionally, we aim towards minimizing 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 conflicts in the set is picked and assigned to a compatible sub-workload $\mathcal{Q}^c$. 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. Note that a single query always produces at most one PIC and hence is always free from conflicts.

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.

5.5 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 a refined-block 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 refined-blocks, the PRBs need not be disjoint. Therefore, 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.

5.5.1 Valid Division

A valid division is defined as follows:

Definition 5.6 *Given $\mathcal{Q}^c, \overline{\mathbb{R}}^{\mathbb{A}}$ 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 $c \in \mathcal{Q}^c$ 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 $c \in \mathcal{Q}^c$.*

Condition 1 is needed to associate an PRB with its constituent CPBs. This is required during data generation in order to populate appropriate refined-blocks 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 5.2 *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 $p_2 \in \mathbb{P}$ are such that $p_1 L \overline{r} \Leftrightarrow p_2 L \overline{r}$ for $\overline{r} \in \overline{\mathbb{R}}$. We need to prove that replacing p_1 and p_2 with $p_{1,2} = p_1 \cup p_2$ in $\mathbb{P}$ 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 $\mathbb{P}$ 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 $\rho = \{p \in \mathbb{P} \setminus \{p_1, p_2\} : pL\overline{r}\}$. 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 PIC in $\mathbb{Q}^c$ 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 $\mathbb{P}$ that are related to c through $M \circ L^{\mathbb{A}}$. That is,

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

  Likewise, p_2 will also be disjoint with all the other elements of $\mathbb{P}$ 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 $\mathbb{P}$ that are related to c.

□

From Lemma 5.2, 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:

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

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 GPA in our running example. $\overline{\mathbb{R}}^{GPA} = \{\overline{r}_1, \overline{r}_2, \overline{r}_3\}$. Since there are three PRBs, seven possible mappings exist. Figure 5.4 illustrates these seven mappings. Powerset Division (`Pow-PSD`) creates seven CPBs, one CPB corresponding to each mapping. Hence, the seven resulting CPBs in $\mathbb{P}^{GPA}$ are as follows:

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

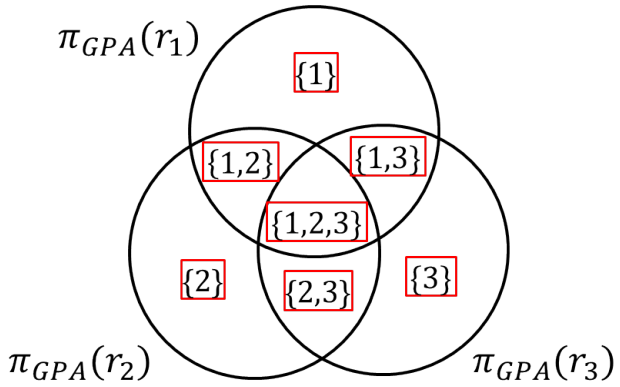


Figure 5.4: Partitioning in Projected Space

5.5.2 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 5.7 *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,

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

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 $c \in \mathcal{Q}^c$. 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 $c \in \mathcal{Q}^c$, 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 p associated with each vector $v_p \in \mathbb{V}$. 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_p 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 5.5 (for simplicity, the elements of $\mathbb{V}$ are shown instead of $\mathbb{Q}$).

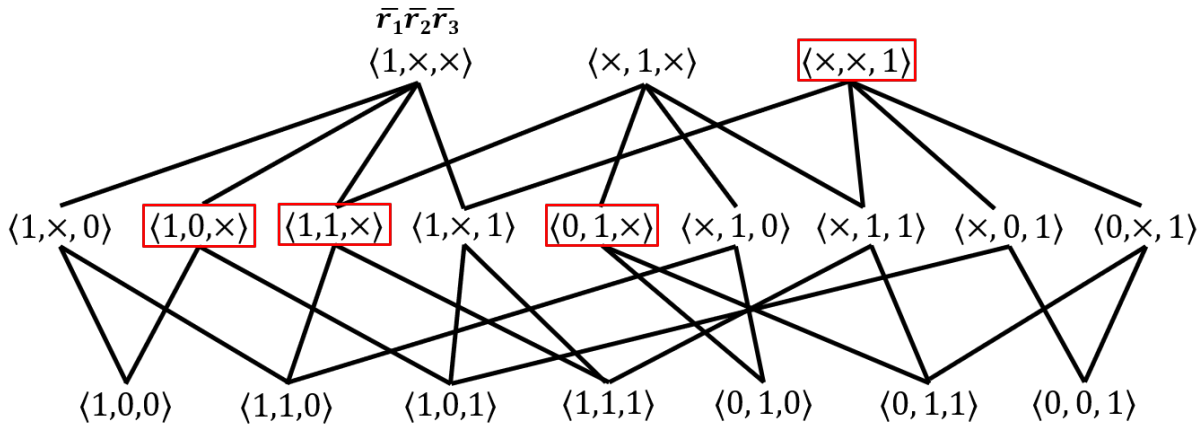


Figure 5.5: Hasse Diagram

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 4, are described next.

5.5.3 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 4, 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 4). 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_p 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_p 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) $v_p(pivot) = 1$, 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_p 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.

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.

Example Division

Consider the projection subspace of GPA in Example 1. $\overline{\mathbb{R}}^{GPA} = \{\overline{r}_1, \overline{r}_2, \overline{r}_3\}$. Let us see how the CPBs for projection subspace of GPA are created by `Opt-PSD`. The input DG for the example is shown in Figure 5.6.

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

Algorithm 4: Optimal Projection Subspace Division

Input: Division Graph G
Output: Optimal Vectors-set $\mathbb{V}^*$

```
1  toBeSplit ← ∅;
2  visited ← ∅;
3  for r̄ in ℝ̄ do
4      visited ← visited ∪ r̄ v_p_init ← {×}^m, v_p_init(r̄) ← 1;
5      toBeSplit ← {v_p_init};
6      while toBeSplit ≠ ∅ do
7          v_p ← toBeSplit.pop();
8          pivot, targets ← getPivot(G, v_p);
9          if pivot exists then
10             toBeSplit ← toBeSplit ∪ Split(v_p, pivot, targets, visited);
11         else
12             V* ← V* ∪ {v_p};
13 return V*;
```

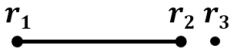


Figure 5.6: Example Division Graph

Iteration 1: $\overline{r}_1$ is picked, $v_p\_init = \langle 1 \times \times \rangle$ is added to $toBeSplit$, $toBeSplit = \{\langle 1 \times \times \rangle\}$. After popping, $v_p = \langle 1 \times \times \rangle$, $getPivot$ returns $pivot = 1, targets = \{2\}$ as vertex 1 is connected to vertex 2. The split function splits v_p 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 $v_p = \langle \times 1 \times \rangle$ is added to $toBeSplit$, $toBeSplit = \{\langle \times 1\times \rangle\}$. After popping, $v_p = \langle \times 1\times \rangle$ 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, $v_p = \langle \times \times 1 \rangle$, no pivot is found by $getPivot$ as vertex 3 is not connected to any other vertex. v_p is added to $\mathbb{V}^*$.

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

Finally, $\mathbb{V}^* = \{\langle 11\times\rangle, \langle 10\times\rangle, \langle 01\times\rangle, \langle \times\ \times\ 1\rangle\}$ (highlighted in Figure 5.5). Using Equation 5.5, it yielded in 4 CPBs for GPA, $\mathbb{P}^* = \{p_1, p_2, p_3, p_4\}$ (as discussed in Section 5.3) 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 behavior, the number of CPBs for `Opt-PSD` for a few general DGs are shown in Table 5.3.

Table 5.3: 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)	$2^m - 1$

Proof of Optimality

We now prove that `Opt-PSD` produces the optimal division. For a CPB $p \in \mathbb{P}$, consider the set of points given as: $\left[\bigcap_{\overline{r}:v_p(\overline{r})=1} \overline{r} \setminus \bigcup_{\overline{r}':v_p(\overline{r}')=0,\times} \overline{r}'\right]$. This is a subset of p and cannot overlap with any $p' \in \mathbb{P} \setminus \{p\}$. This restriction leads to the following lemma:

Lemma 5.3 *Given $(\mathbb{P}, L)$ returned by* `Opt-PSD`*, $\forall p \in \mathbb{P}$, there exists a point $t \in p$ such that $t \notin p', \forall p' \in \mathbb{P} \setminus \{p\}$.*

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

Lemma 5.4 `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 $|\mathbb{P}'| \leq \mathbb{P}$.

$$\implies \exists t_1 \in p_1, t_2 \in p_2 (\neq p_1) \text{ for some } p_1, p_2 \in \mathbb{P}, \text{ where } p_1 L \overline{r}_1, p_2 L \overline{r}_2,$$
$$\overline{r}_1, \overline{r}_2 \in \overline{\mathbb{R}}, \text{ such that } t_1, t_2 \in p', p' L' \overline{r}_1, p' L' \overline{r}_2 \text{ for some } p' \in \mathbb{P}'.$$

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

$$\implies \exists c \text{ such that } \overline{r} M c, \overline{r}' M c, \text{ for some } \overline{r}' \in \overline{\mathbb{R}} \text{ and}$$
$$(p_1, \overline{r}') \in L, (p_2, \overline{r}') \notin L \text{ (wlog) (using Lemma 5.2)}$$
$$\implies t_2 \notin \overline{r}', \text{ otherwise there would exist } p_3 \in \mathbb{P} \text{ such that } t_2 \in p_3;$$
$$p_2 \cap p_3 \neq \emptyset \text{ and } p_3 L \overline{r}' \text{ would imply Condition 2 violation.}$$
$$\implies \exists p'' \in \mathbb{P}' \text{ such that } p'' L' \overline{r}', t_1 \in p'' \text{ and } t_2 \notin p''.$$
$$Since, p' \cap p'' \neq \emptyset \text{ and } (p', c), (p'', c) \in M \circ L'$$
$$\text{Hence, contradiction (Condition 2 violation).}$$

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

(2a): $t_1 \in p_1 \setminus p_2$ (or $t_2 \in p_2 \setminus p_1$, wlog)

Since, $t_1 \in p_1, p_1 L \overline{r}_2$, therefore $t_1 \in \overline{r}_2$

$\implies \exists p_3 \in \mathbb{P}$ such that $t_1 \in p_3$ and $p_3 L \overline{r}_2$

p_2, p_3, p' are such that $t_1 \in p_3, t_2 \in p_2, t_1, t_2 \in p', p_2 L \overline{r}_2, p_3 L \overline{r}_2, p' L' \overline{r}_2$.

This is not possible using result of Case (1). Contradiction.

(2b): $t_1, t_2 \in p_1 \cap p_2$

p_1, p_2 has at least one point each that is absent in all the other CPBs (using Lemma 5.3). Therefore, if t_1, t_2, 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. □

5.6 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 $c \in \mathbb{Q}^c$ 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 $M \circ L^{\mathbb{A}}$.

$$|\pi_{\mathbb{A}}(\sigma_f(T))| = \sum_{p:(p,c)\in M\circ L^{\mathbb{A}}} |p| \tag{5.6}$$

Further, since each $\overline{r} \in \overline{\mathbb{R}}^{\mathbb{A}}$ is related to at least one $c \in \mathbb{Q}^c$ 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.

$$|\overline{r}| = \sum_{p:pL^{\mathbb{A}}\overline{r}} |p| \tag{5.7}$$

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.

5.6.1 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

$$\sum_{r:rMc} x_r = l \tag{5.8}$$

(b) Distinct Row Cardinality Constraint (using Equation 5.6)

$$\sum_{p:(p,c)\in M\circ L^{\mathbb{A}}} y_p = k \tag{5.9}$$

5.6.2 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 refined-block and CPB are non-negative in the LP solution. That is,

$$x_r \geq 0, \forall r \in \mathbb{R}, \quad \text{and} \quad y_p \geq 0, \forall p \in \mathbb{P}^{\mathbb{A}}, \text{ for all PAS } \mathbb{A} \tag{5.10}$$

Type 2: These constraints ensure that the total number of tuples for each refined-block is greater than or equal to the number of distinct tuples along each applicable PAS for that block. Using Equation 5.7, these constraints, for each refined-block r and each of its associated PAS $\mathbb{A}$, are expressed as follows:

$$\sum_{p:pL^{\mathbb{A}}\overline{r}} y_p \leq x_r \tag{5.11}$$

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 a refined-block 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 refined-block r and each of its associated PAS $\mathbb{A}$:

$$x_r \leq |T| \sum_{p:pL^{\mathbb{A}}\overline{r}} y_p \tag{5.12}$$

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

We had already seen, in Section 5.3, the explicit constraints for our running example. The associated sanity constraints are shown in the box below:

$$
\begin{aligned}
\textbf{Type 1} \quad & x_1, x_2, x_3, x_{4a}, x_{4b} \geq 0 \\
& y_1^{GPA}, y_2^{GPA}, y_3^{GPA}, y_4^{GPA} \geq 0 \\
& y_1^{Age}, y_2^{Age}, y_3^{Age}, y_4^{Age}, y_5^{Age}, y_6^{Age} \geq 0 \\
\textbf{Type 2, 3} \quad & y_1^{GPA} + y_2^{GPA} \leq x_1 \leq |T|(y_1^{GPA} + y_2^{GPA}) \\
& y_1^{GPA} + y_3^{GPA} \leq x_2 \leq |T|(y_1^{GPA} + y_3^{GPA}) \\
& y_4^{GPA} \leq x_3 \leq |T| y_4^{GPA} \\
& y_1^{Age} + y_3^{Age} \leq x_2 \leq |T|(y_1^{Age} + y_3^{Age}) \\
& y_2^{Age} + y_6^{Age} \leq x_3 \leq |T|(y_2^{Age} + y_6^{Age}) \\
& y_3^{Age} + y_4^{Age} \leq x_{4a} \leq |T|(y_3^{Age} + y_4^{Age}) \\
& y_2^{Age} + y_5^{Age} \leq x_{4b} \leq |T|(y_2^{Age} + y_5^{Age})
\end{aligned}
$$

5.6.3 Sufficiency for Data Generation

For a refined-block 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 refined-block is symmetric in nature, data across different PASs can be generated independently and concatenated together. Therefore, the constructed LP is sufficient for data generation.

5.7 Data Generation

The LP solution gives the following information:

1. A list of refined-blocks with their corresponding row cardinalities, and
2. For each refined-block 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 refined-blocks. Now, we drill down to assign intervals for each CPB, thereby producing the summary tabulation for all refined-blocks. The CPBs along each PAS are assigned intervals independently since each refined-block 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.

5.7.1 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^{Age} and p_6^{Age} intersect in Std. 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^{Age} and p_6^{Age} are allocated disjoint intervals for PAS Age as $(p_2^{Age}, c_3), (p_6^{Age}, c_3) \in M \circ L^{Age}$. On the other hand, in the case of PAS GPA, p_2^{GPA} and p_4^{GPA} are not related to any c in $\mathbb{Q}^c$, and therefore their data generation intervals can overlap.

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

$p_2^{GPA} \leftarrow [5, 8)$	$p_2^{Age} \leftarrow [30, 37)$
$p_4^{GPA} \leftarrow [4, 9)$	$p_6^{Age} \leftarrow [37, 50)$

The summary is maintained refined-block-wise, with the template structure shown in Figure 5.7. 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, a refined-block 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 refined-block, shown as $\mathbb{A}_{left}$, the domain of the projection block is kept as is and no distinct tuple count is maintained. Lastly, each refined-block has an associated total cardinality. A populated instance of the template, and its interpretation, was discussed earlier in Section 5.3.5.

$\mathbb{A}_1$	$\mathbb{A}_2$	...	$\mathbb{A}_\alpha$	$\mathbb{A}_{left}$	
CPB-1: card.,	CPB-1: card.,	...	CPB-1: card.,		refined-block
CPB-2: card.,	CPB-2: card.,	...	CPB-2: card.,	PB	Card.
...	...	...	...		

Figure 5.7: Sample refined-block in Summary

5.7.2 Tuple Generation

Using the information in the summary, the tuples of the table are instantiated. Specifically, the algorithm iterates over each refined-block and generates the number of rows specified in

the associated total cardinality value. For a refined-block 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 refined-block 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 refined-blocks. 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 refined-blocks. As a case in point, for the sample summary in Figure 5.3, the partial tuples generated for the CPB with interval $[30, 37)$ and distinct row cardinality 5 are identical in both r_3 and r_{4b}.

5.8 Pipeline

The extension of the end-to-end Hydra pipeline, to incorporate Projection, is shown in Figure 5.8. The new/updated modules that differ from previous chapter are shown in green color.

Hydra takes a set $\mathbb{Q}$ of PICs over a single table T as input. Let β be the total number of PASs across all the constraints, as indicated in Figure 5.8. From the constraints, Hydra produces data for T. This is carried out by a sequence of core components, namely *Workload Decomposition*, *LP Formulation*, and *Data Generation* modules. **Workload Decomposition** splits $\mathbb{Q}$ 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 $\mathbb{Q}^c$ begins with **Region Partitioning** followed by **Symmetric Refinement** algorithm. This gives the set of refined-blocks. For each PAS across all PICs, the PRBs are computed using the refined-blocks. These PRBs and $\mathbb{Q}^c$ 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 refined-blocks 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

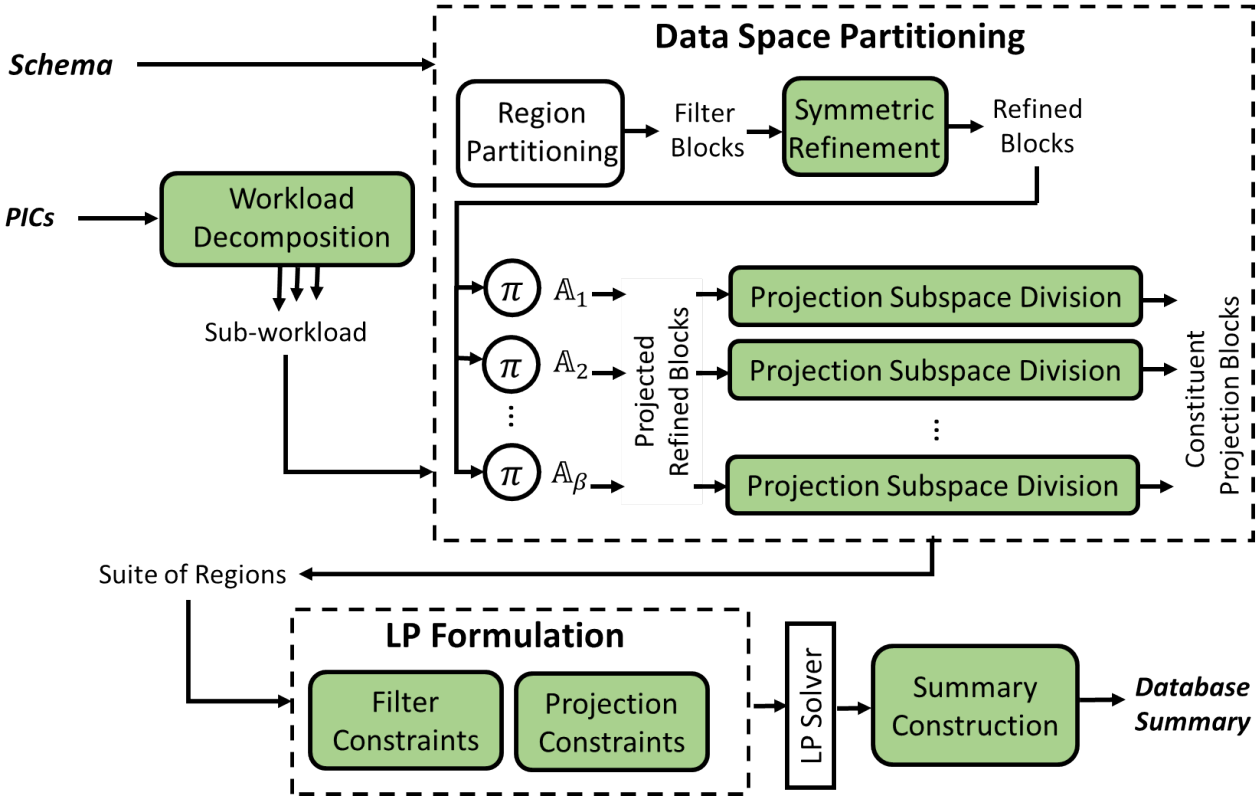


Figure 5.8: Projection Solution Pipeline

desires a materialized database instance, it can be generated from the summary and stored persistently.

5.9 Discussion

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

5.9.1 Solution Guarantees

The input workload feasibility is true by definition since the PICs have been derived from an existing setup. In such scenarios, Hydra 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 5.5 *For a feasible and compatible set of PICs, Hydra 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 Hydra can cover all the input constraints. We formally prove Lemma 5.5 next.

Proof of Lemma 5.5

We briefly discuss the proof for Lemma 5.5, which is split into two parts: (a) The LP constructed for a feasible compatible workload $\mathcal{Q}^c$ is always satisfiable; (b) Given any LP solution, data can be always be constructed from it, and this data will satisfy $\mathcal{Q}^c$.

Part (a): Given workload feasibility, there exists at least one instance T of the table that satisfies $\mathcal{Q}^c$. Further, due to compatibility, $\mathcal{Q}^c$ 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 refined-block, the total tuple count in the refined-block 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 refined-block 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 modeled in the LP using the Explicit constraints, the generated data is compliant with $\mathcal{Q}^c$.

5.9.2 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 introduced in Section 5.5. 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^m. Here, m is the number of refined-blocks that are subjected to projection along a PAS $\mathbb{A}$.

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 has a corresponding CPBs in the solution and vice versa.

Further, m itself is $\mathcal{O}(2^{|\mathcal{Q}^c|})$. 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 5.10).

Lastly, the decision version of the general data generation problem is `NEXP-complete`, as shown in [17].

5.10 Experimental Evaluation

In this section, we evaluate the empirical performance of a Java-based implementation of Hydra. Z3 solver [14] 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 Hydra, and are conducted using the PostgreSQL v9.6 engine [7] on a vanilla HP Z440 workstation.

Workload Construction. In presenting the experimental results, we initially focus on fully compatible workloads. Subsequently, in Section 5.10.5, we discuss the corresponding performance for workloads featuring conflicts. 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 [12] 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 [37] 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 [45, 5] 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 5.4. 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).

Table 5.4: 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

We compare Hydra against the **DataSynth** framework that supports value cardinality constraints. For DataSynth, projection constraints need to be restricted to single attribute tables.

5.10.1 Constraint Accuracy

When Hydra 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:

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

$\mathbf{f} : d\_year = 2002 \ \wedge$
$(i\_category \in (\text{'Jewelry'},\text{'Women'}) \wedge i\_class \in (\text{'mens watch'},\text{'dresses'})) \vee$
$(i\_category \in (\text{'Men'},\text{'Sports'}) \wedge i\_class \in (\text{'sports-apparel'},\text{'sailing'}))$ 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.

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. 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 5.5: 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

Having established the shortcomings of the prior work, we restrict our attention to Hydra in the rest of this section.

5.10.2 Generated Data

We now show a concrete example of how the data generated by Hydra 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 5.6. 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.

Table 5.6: 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
...	...	...	...	...

5.10.3 Time and Space Overheads

Having established the accuracy credentials of Hydra, 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 5.7. 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 minuscule, 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 5.8 – specifically, the table shows the number of filter-blocks, refined-blocks, and CPBs created by Hydra. We see here that there is huge jump in the number of regions from the initial filter-block to the final CPBs, testifying that $\mathcal{Q}^c$ 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 5.4), 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`

Table 5.7: 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 5.8: Block Profiles

Table	Cardinality of		
	filter-blocks	refined-blocks	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

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.

Table 5.9: No. of Blocks and Comparison against `Pow-PSD`

Table	# Filter-Blocks	#RBs	#PRBs Opt-PSD	#PRBs Pow-PSD	Multiplicative Speed-up
SS	74	88	132662	524404	70
CS	139	141	165936	524336	16

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

Table 5.10: 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

5.10.4 Scalability Profile

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 Hydra 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 5.9(a) and 5.9(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 5.9(a) is on a log scale).

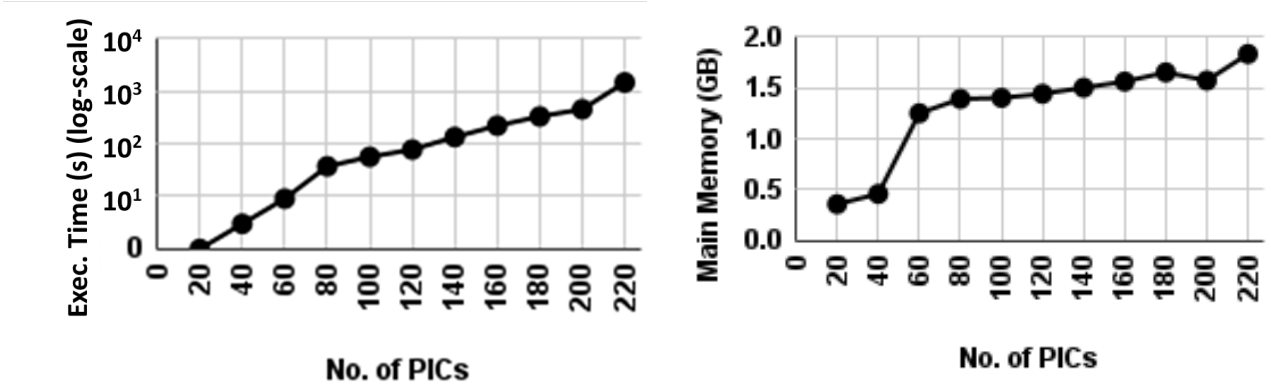


Figure 5.9: (a) Execution Time (b) Memory Usage

5.10.5 Workload Decomposition

We now turn our attention to conflicting workloads, which require the pre-processing step of workload decomposition. To model this scenario, we added conflicting 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 Hydra results on $\mathcal{Q}$ for two decomposition strategies: (a) Instance-based Decomposition, and (b) Template-based Decomposition, which are discussed below.

5.10.5.1 Instance-based Decomposition (ID)

Here the decomposition algorithm uses Definition 5.5 of a conflicting pair, and for this framework, the number of workloads obtained for the four tables are shown in Table 5.11. We observe that despite using an approximate vertex coloring algorithm (Section 5.4.2), a partitioning of $\mathcal{Q}$ 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 $\mathcal{Q}^c$ in Table 5.7. At first glance, this might

appear surprising given that $\mathcal{Q}$ is more complex in nature – the reason is that due to workload decomposition, an array of databases is produced for $\mathcal{Q}$ with low individual production complexity, whereas a single unified database is produced for $\mathcal{Q}^c$. 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 5.11: 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

5.10.5.2 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 5.12 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.

Table 5.12: 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

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

5.11 Conclusion

In this chapter, the scope of the supported constraints in Hydra were expanded to include 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 Hydra successfully produces generation summaries within viable time and space overheads.

Chapter 6

Regeneration using Join Constraints

6.1 Introduction

So far the regeneration algorithms discussed were restricted to constraints on a single table. In presence of multiple tables, *join* constitutes an important primitive operation. Ensuring volumetric similarity for queries involving multiple tables is dependent critically on modeling joins. To understand the underlying concepts let us revisit the university database schema from the Introduction chapter.

```
Register (Rid, RollNo, Course, Year, Score)
Student(RollNo, Age, Scholarship, GPA)
```

Here, *Register.Rid* and *Student.RollNo* are the primary key columns of *Register* and *Student* tables respectively. Further, *Register.RollNo* is a foreign key column referencing *Student.RollNo*.

Now, consider the following queries featuring joins: (For ease of exposition, we use the same constants in the queries.)

1. ```
   Select Distinct GPA From Register Reg, Student Std
   Where Reg.RollNo = Std.RollNo and GPA < 6 and Age < 20;
   ```

2. ```
   Select Distinct GPA From Register Reg, Student Std
   Where Reg.RollNo = Std.RollNo and (GPA < 6 or Age >= 20);
   ```

The client AQPs obtained for the above two queries are shown in Figure 6.1. As we know, the aim is to generate a synthetic database that preserves volumetric similarity. That is, the AQPs obtained on the synthetic database match the client AQPs.

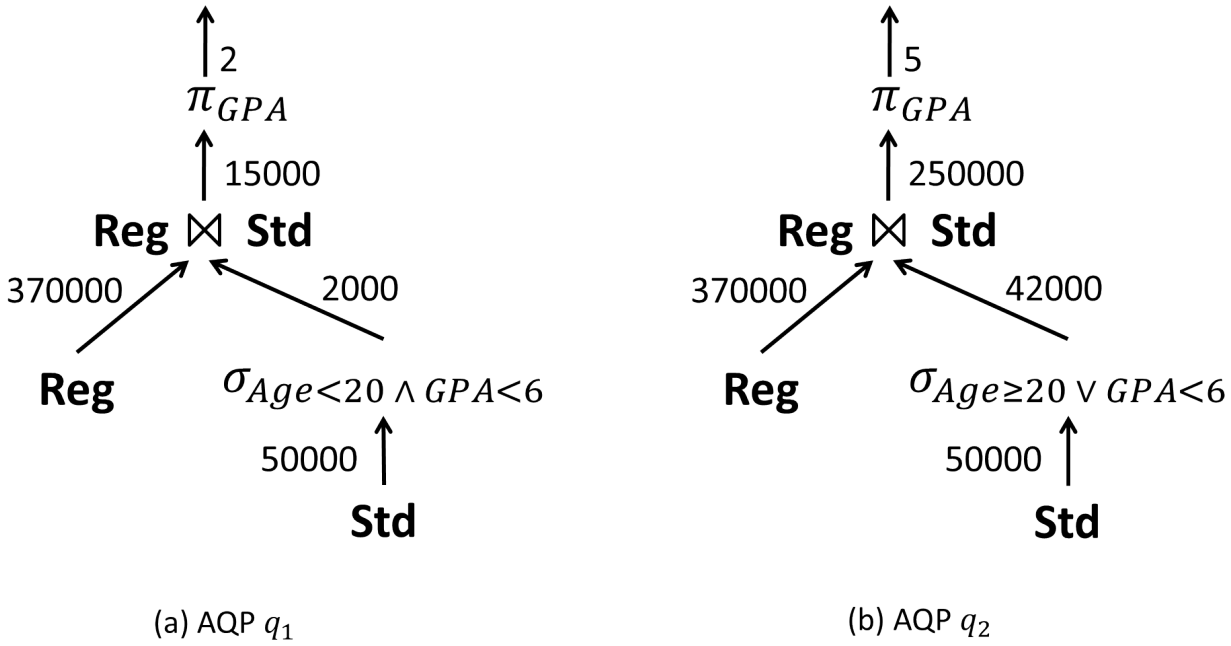


Figure 6.1: Example AQPs

In the previous two chapters, we encoded each AQP in a single CC/PIC. In presence of joins, each AQP gives multiple CCs. Specifically, we construct a CC for each edge in the AQPs. A CC comprising of SPJ operators is represented as:

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

where k is the number of rows that are output after applying the complete relational expression, i.e., the output cardinality, $\mathbb{A}$ represents the set of attributes on which projection is applied (PAS), and f represents the filter conditions on the inner join of relations $T_1, T_2, ..., T_N$. Further, the joins are restricted to PK-FK joins.

For the AQPs shown in Figure 6.1, following set of CCs are obtained:

$c_1 : |\pi_{GPA}(\sigma_{Age<20 \wedge GPA<6}(Reg \bowtie Std))|= 2$ $\quad$ $c_2 : |\pi_{GPA}(\sigma_{Age\geq20 \vee GPA<6}(Reg \bowtie Std))|= 5$

$c_3 : |\sigma_{Age<20 \wedge GPA<6}(Reg \bowtie Std)|= 15000$ $\quad$ $c_4 : |\sigma_{Age\geq20 \vee GPA<6}(Reg \bowtie Std)|= 250000$

$c_5 : |\sigma_{Age<20 \wedge GPA<6}(Std))|= 2000$ $\quad$ $c_6 : |\sigma_{Age\geq20 \vee GPA<6}(Std))|= 42000$

$c_7 : |Reg|= 370000$ $\quad$ $c_8 : |Std|= 50000$

In the above we can see that the c_1, c_2, c_3, c_4 CCs feature the join operator. Among these, c_1

and c_2 additionally feature the projection operator. To ensure matching AQPs at the vendor, the synthetic data needs to satisfy all the CCs.

6.1.1 Challenge

Filter predicates considered so far specify the constants/value-ranges that are permissible for the constrained columns. In other words, these were of the type $\langle col \quad op \quad val \rangle$. In contrast, modeling join predicates require constructing dependence with respect to the join columns such that the generated tables obey the required join output cardinality across all constraints. For instance consider the queries wrt Figure 6.1. The filter predicates directly operate on the domain of GPA and Age columns, whereas the join predicates equate $RollNo$ column across the two tables. Hence, these columns need to possess a relation that can lead to the desired output cardinality.

6.1.2 Background

A way to deal with join operators was proposed in DataSynth[17], where they first constructed a view for each table in the database, and at the end, extracted the original table. Specifically, the view V_T for a table T comprises of all the non-key attributes of T and the tables on which it depends through referential constraints. These views allow rewriting the join expression on a single view. Using these views, filters on each view can be handled independently using the single table algorithm. However, the challenge then lies in extracting the original tables from the views. This is because these views need to obey referential integrity. For DataSynth, due to the absence of projection operation in its input, adding some spurious tuples in the referenced table was sufficient to ensure referential integrity. However, this volume discrepancy magnifies in presence of projection because each value-combination with respect to the borrowed columns needs to be represented in the referenced table. Therefore, the LP formulation and the subsequent data generation from the solution need to explicitly model referential constraints.

6.1.3 Our Contributions

In this work, we provide a comprehensive solution to handle the join constraints. Specifically, we update the aforementioned views to be equivalent to the denormalized versions of the original tables (excluding key columns). Specifically, for each borrowed column in the view, we maintain the path from the source to the destination table in the join graph through which the column is borrowed. This allows for multiple copies of a borrowed column in the view if there are multiple join paths possible between the tables involved. Hence, this helps to handle DAG join graphs instead of the tree-join graphs that DataSynth handled.

Further, a marked contrast is in the way we model join conditions in the solution pipeline. We construct a *unified* LP for the *linked* (through referential constraints) tables. This LP models the referential constraints to ensure that the number of distinct value-combinations generated, with respect to the borrowed columns, in various intervals of the Foreign-Key table (referencing) view is upper bounded by the corresponding interval in the (referenced) Primary-Key table. Further, our *Key Curation* module ensures that the key values picked are such that the corresponding tuples in the dimension table have the prescribed number of distinct value-combinations for borrowed columns.

Additionally, the feature of dynamic database regeneration is preserved. Therefore, no materialized table is required in the entire testing pipeline. Further, the time and space overheads incurred in constructing the summary are independent of the size of the table to be constructed.

A detailed evaluation on workloads derived from TPC-DS and JOB benchmarks has been conducted. The results demonstrate that the proposed solution accurately and efficiently models the SPJ CCs. As a case in point, for TPC-DS based workload of over 100+ queries, leading to ∼500 CCs, the generated data satisfied all the CCs with perfect accuracy. Moreover, the entire summary production pipeline completed within viable time and space overheads.

6.1.4 Organization

The rest of the chapter is organized as follows: we discuss the problem framework in Section 6.2. The overview of the design principles of the proposed technique is presented in Section 6.3 and further described in detail in Sections 6.4-6.8. Further, an optimized solution to handle Select-Join workloads is given in Section 6.9. The experimental evaluation is presented in Section 6.10 and finally, we conclude in Section 6.11.

6.2 Problem Framework

In this section, we summarize the problem statement, the underlying assumptions, the output delivered, and a tabulation of the notations used in this chapter.

6.2.1 Problem Statement

Given an SPJ query-workload $\mathcal{W}$, with its corresponding set of AQPs $\mathcal{Q}$, derived from an original database with schema $\mathcal{S}$ and statistical metadata $\mathcal{M}$, the objective is to generate a synthetic database $\mathcal{D}$ such that it conforms to $\mathcal{S}$ and $\mathcal{Q}$. That is, the AQPs obtained from the original database match, wrt the cardinality annotations, the AQPs obtained on $\mathcal{D}$.

6.2.2 Assumptions

We assume that $\mathcal{W}$ comprises of only PK-FK joins such that each query can be mapped to denormalized tables as discussed in Chapter 3. Further, we assume that the filters and projections are applied only on non-key columns. These assumptions are common in prior work as well as OLAP benchmarks.

Again, we assume that $\mathcal{Q}$ is collectively feasible. Finally, for brevity, we present the ideas using tables with columns having float data type; the extension to other data types is straightforward.

6.2.3 Output

Given $\mathcal{S}$, $\mathcal{M}$, $\mathcal{W}$ and $\mathcal{Q}$, Hydra outputs a collection of database summaries $\mathbb{S}$. Each summary $s^{\mathcal{D}} \in \mathbb{S}$ can be used to deterministically produce the associated database $\mathcal{D}$. The databases produced are such that: (a) all of them conform to $\mathcal{S}$, and (b) for each query in $\mathcal{W}$, its corresponding AQP in $\mathcal{Q}$ matches with the AQP obtained on at least one output database instance.

6.2.4 Notations

The main acronyms and key notations used in this chapter are summarized in Tables 6.1 and Table 6.2, respectively.

Table 6.1: Acronyms

Acronym	**Meaning**
AQP	Annotated Query Plan
CC	Cardinality Constraint
SPJ	Select Project Join
PAS	Projection Attribute Set
ARB	Aligned Refined Block
CPB	Constituent Projection Block
PSD	Projection Subspace Division
NoPB	No Projection on a Subset of Borrowed Columns
PB	Projection on a Subset of Borrowed Columns

6.3 Design Principles

The solution pipeline is illustrated in Figure 6.2. The green boxes illustrate the modules added/updated to handle joins. We briefly discuss each of the core modules in this section. For ease of exposition, we use the *fact and dimension table* terminology from data warehousing to

Table 6.2: Notations

(a) Database Related

Symbol	Meaning
$\mathcal{S}$	Database Schema
$G_{\mathcal{S}}$	Schema Graph
$\mathcal{D}$	Output Database
T	Output Table
$s^{\mathcal{D}}$	Summary of $\mathcal{D}$
F	Fact Table
D	Dimension Table
V_T	View wrt T
$\mathbb{B}$	Borrowed Attribute-Set

(b) Workload Related

Symbol	Meaning
q	Query
$\mathcal{W}$	Query Workload
$\mathcal{Q}$	Set of AQPs
c	A c
f	Filter Predicate
$\mathbb{A}$	PAS
l	Output row card. after filter
k	Output row card. after projection

(c) Block Related

Symbol	Meaning
r^T	refined-block wrt V_T
a^T	ARB wrt V_T
p^T	CPB wrt V_T
$x(a^T)$	variable for $\|a^T\|$
$y(p^T)$	variable for $\|p^T\|$

(d) Relation/Function Related

Symbol	Meaning
$U(T)$	Set of attributes in T
$dom(.)$	Domain of the input parameter
M	A relation btw CCs and ARBs
L	A relation btw CPBs and ARBs
H	A relation btw ARBs wrt V_F and V_D
J	A relation btw CPBs wrt V_F and V_D

refer to the tables having FK and the corresponding PK, respectively. Further, the notations F and D are used to denote a fact table and dimension table, respectively.

6.3.1 Denormalization

Inspired from DataSynth, we also construct views, where the view V_T for a table T is its denormalized equivalent (excluding key columns). Each column in a view is stored as a structure comprising of two fields: {Column Name, Column Path}. The Column Path is stored as an array of the foreign key columns involved in the query. Each view is associated with a view name and an array of the column structures. To populate this array, first a schema graph $G_{\mathcal{S}}$ is constructed. Here, a vertex is made corresponding to each table in the database. For each PK-FK dependency, a directed edge is added from the FK table to PK table. The edge is annotated with the participating columns of both the relations. Note that, since two relations can have multiple PK-FK dependencies, there can be multiple directed edges between tables. We assume $G_{\mathcal{S}}$ to be a DAG, which is common in real-world databases and is reflected in the benchmarks too.

Once $G_{\mathcal{S}}$ is constructed, we traverse it in reverse topological order. For each edge

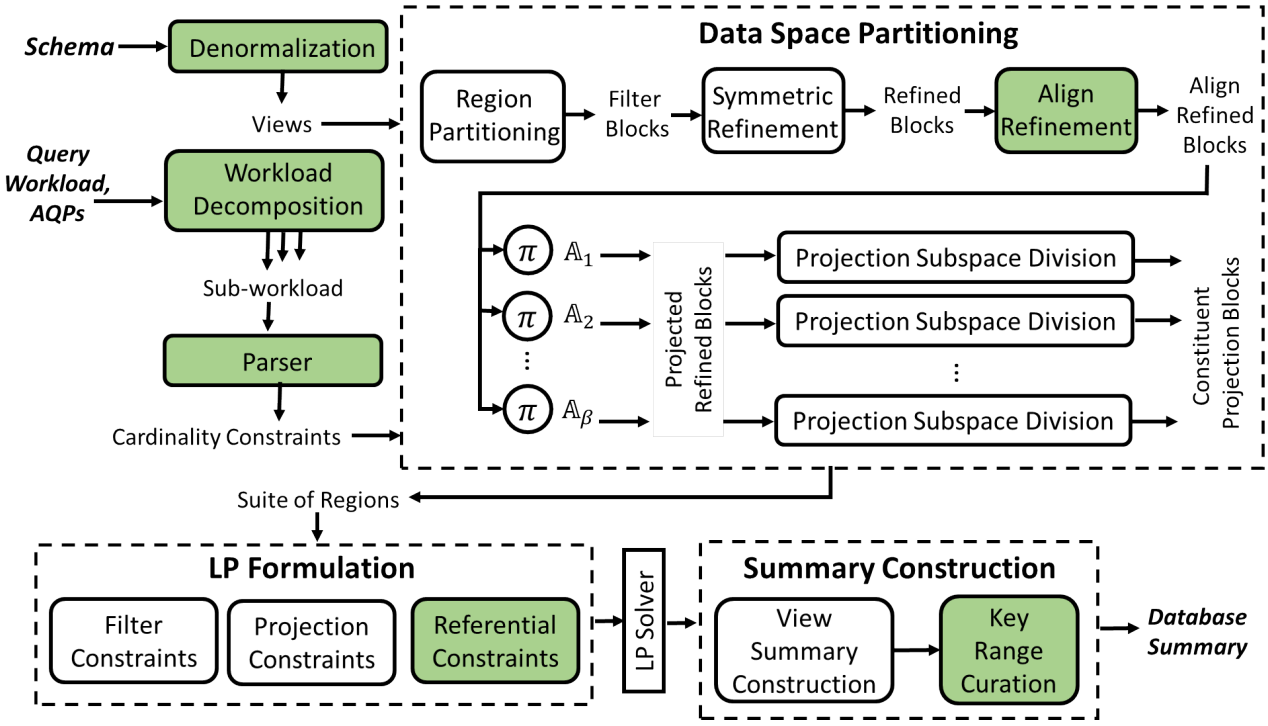


Figure 6.2: Join Solution Pipeline

$(F.fk, D.pk)$ in the order, the columns in view V_D are added to V_F. For each such column A, the Column Path of $V_D.A$, prepended with $F.fk$, is inserted as the Column Path for $V_F.A$.

The join expression in a CC c, consists of series of PK-FK columns that participate in the joins in the subtree rooted at the AQP node corresponding to c. It is easy to see that this PK-FK column series can be uniquely mapped to a path in G_s. The view V_T corresponding to the source vertex in this path will have columns from all the tables included in the path. Also, it is easy to see that the filter and projection operation on a join expression can be equivalently written on V_T. Hence, in this way, each CC can be written as a filter and projection operation on a single view.

For our running example, the Views that are constructed are as follows:

$$V_{Reg}(Age, Scholarship, GPA, Course, Year, Score), \quad V_{Std}(Age, Scholarship, GPA)$$

Further, the aforementioned CCs from our running example, can be rewritten on the views V_{Reg} and V_{Std} as follows:

$$c_1 : |\pi_{GPA}(\sigma_{Age<20 \wedge GPA<6}(V_{Reg}))| = 2 \qquad c_2 : |\pi_{GPA}(\sigma_{Age\geq 20 \vee GPA<6}(V_{Reg}))| = 4$$
$$c_3 : |\sigma_{Age<20 \wedge GPA<6}(V_{Reg})| = 15000 \qquad c_4 : |\sigma_{Age\geq 20 \vee GPA<6}(V_{Reg})| = 250000$$

$c_5 : |\sigma_{Age<20 \wedge GPA<6}(V_{Std}))| = 2000$ $\quad$ $c_6 : |\sigma_{Age \geq 20 \vee GPA<6}(V_{Std}))| = 42000$

$c_7 : |V_{Reg}| = 370000$ $\quad$ $c_8 : |V_{Std}| = 50000$

As described earlier, referential integrity has to be ensured in the data that is generated. Let us first describe referential integrity in the view semantics.

Theorem 6.1 *Two tables F and D, with F having an FK column referencing table D, satisfy a referential integrity dependency, iff the corresponding views V_F and V_D obey the following condition:*

$$\pi_{\mathbb{B}}(V_F) \subseteq \pi_{\mathbb{B}}(V_D)$$

where $\mathbb{B}$ is the set of columns in V_D borrowed by V_F.

6.3.2 Workload Decomposition

In the previous chapter, a pair of constraints were defined to be overlapping if their PASs partially intersect and their filters overlap. To handle this, we had an additional workload decomposition module that splits the input workload into sub-workloads such that each of them is free from these overlapping projection conflicts.

We have extended this case of overlapping projections to include the projection conflicts that surface in the presence of joins. For example, a pair of queries q_1, q_2 on a dimension table D, with PASs $\mathbb{A}_1$ and $\mathbb{A}_2$ respectively, induce a conflict if (a) the PASs $\mathbb{A}_1, \mathbb{A}_2 \subseteq D$, and partially overlap with each other, and (b) the filters in q_1 and q_2 intersect. We discuss the details of all the conflicts in Section 6.4. These conflicts are additionally used by the workload decomposition module to do the workload split.

6.3.3 Data Space Partitioning

Region Partitioning and Symmetric Refinement. To model the filter predicates associated with the input workload, the data space of each view 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. To do this partitioning, we leverage the *region partitioning* technique discussed in Chapter 4, which partitions the data space into the minimum number of filter-blocks.

To handle various projection subspaces (corresponding to the different PASs in the input queries) independently, a *Symmetric Refinement* strategy is adopted (discussed in Chapter 5). Specifically, it refines an filter-block into a set of disjoint *refined blocks* such that each resultant refined-block exhibits translation symmetry along each applicable projection subspace.

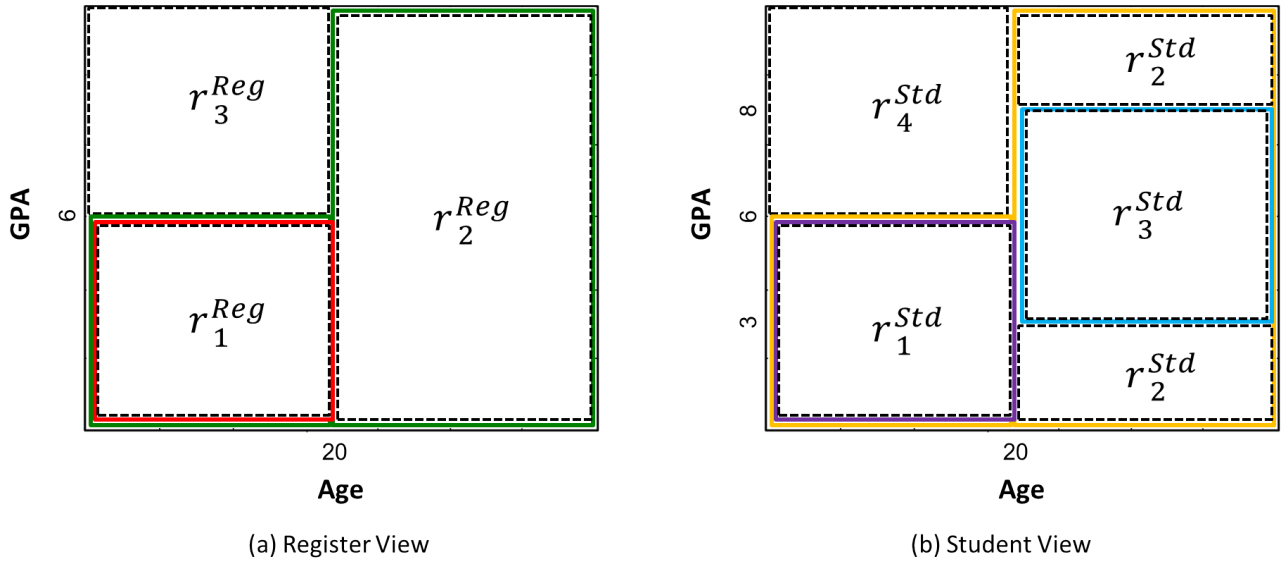


Figure 6.3: Partitioning of *Reg* and *Std* Views

To make the above concrete, the partitioning of data space of V_{Reg} is shown in Figure 6.3(a). For ease of presentation, we show only the two dimensions that participate in the example queries. 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 three disjoint filter-blocks: $r_1^{Reg}, r_2^{Reg}, r_3^{Reg}$, whose domains are depicted with dashed-line boundaries. For our running example, the resultant blocks are already symmetric. The partitioning with respect to V_{Std} is also shown in Figure 6.3(b). Here, we have additionally added a constraint shown by blue solid-line to add complexity in the example.

Align Refinement. To obtain the original tables from their denormalized equivalents, the views need to obey referential integrity. As discussed in Theorem 6.1, the referential integrity constraint between fact table view V_F and dimension table view V_D is expressed as follows:

$$\pi_{\mathbb{B}}(V_F) \subseteq \pi_{\mathbb{B}}(V_D)$$

To add referential constraints, for a CC c with PAS $\mathbb{A}$ (where $\mathbb{A} \subseteq \mathbb{B}$) applicable on V_{Reg}, its constituent refined-blocks need to be aligned with each other along $\mathbb{A}$. By align we mean that the domain of refined-blocks along $\mathbb{A}$ are either identical or disjoint. The blocks obtained after refinement are called *Aligned Refined Blocks* (ARBs). For example, r_2^{Reg} in Figure 6.3(a) is split into a_{2a}^{Reg} and a_{2b}^{Reg} to ensure alignment with a_1^{Reg}, as shown in Figure 6.4(a). (The other refined-blocks happen to be already aligned, so their equivalent ARBs are unaltered).

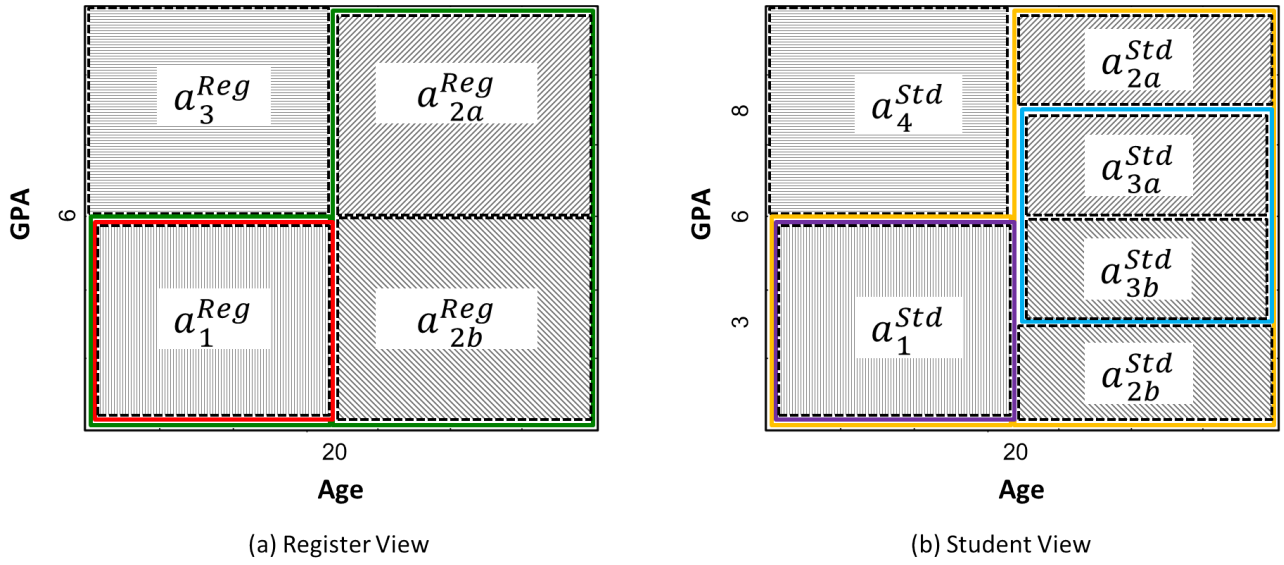


Figure 6.4: Align Refinement

Further, we also need to ensure that each refined-block in V_D is either completely contained or disjoint from the domain of each ARB in V_F. Therefore, we also do a refinement of refined-blocks in V_D as part of this module and produce ARBs that ensure alignment with V_F. For example, r_2^{Std} and r_3^{Std} in Figure 6.3(b) are split into $a_{2a}^{Std}, a_{2b}^{Std}$ and $a_{3a}^{Std}, a_{3b}^{Std}$ respectively, as shown in Figure 6.4(b). We discuss the details of this module in Section 6.5.

Projection Subspace Division. This technique divides each projection subspace into a set of *constituent projection blocks* (CPBs), as discussed in Chapter 5. For our running example, the CPBs obtained for V_{Reg} are as follows:

$$p_1^{Reg} = \pi_{GPA}(a_1^{Reg}) \cap \pi_{GPA}(a_{2b}^{Reg}), \qquad p_2^{Reg} = \pi_{GPA}(a_1^{Reg}) \setminus \pi_{GPA}(a_{2b}^{Reg})$$
$$p_3^{Reg} = \pi_{GPA}(a_{2b}^{Reg}) \setminus \pi_{GPA}(a_1^{Reg}), \qquad p_4^{Reg} = \pi_{GPA}(a_{2a}^{Reg})$$

Further, the CPBs obtained for V_{Std} are as follows:

$$p_1^{Std} = \pi_{GPA}(a_1^{Std}) \cap \pi_{GPA}(a_{3b}^{Std}), \qquad p_2^{Std} = \pi_{GPA}(a_1^{Std}) \setminus \pi_{GPA}(a_{3b}^{Std})$$
$$p_3^{Std} = \pi_{GPA}(a_1^{Std}) \cap \pi_{GPA}(a_{2b}^{Std}), \qquad p_4^{Std} = \pi_{GPA}(a_1^{Std}) \setminus \pi_{GPA}(a_{2b}^{Std})$$
$$p_5^{Std} = \pi_{GPA}(a_{3b}^{Std}) \setminus \pi_{GPA}(a_1^{Std}), \qquad p_6^{Std} = \pi_{GPA}(a_{2b}^{Std}) \setminus \pi_{GPA}(a_1^{Std})$$
$$p_7^{Std} = \pi_{GPA}(a_{2a}^{Std}), \qquad p_8^{Std} = \pi_{GPA}(a_{3a}^{Std})$$

6.3.4 LP Formulation

After the above processing is completed for each view, we formulate a *unified* LP for the *linked* (through referential constraints) tables. The LP is constructed using variables representing the cardinalities of ARBs and CPBs. Specifically, Filter Constraints and Projection Constraints are modeled for each view in the same way as proposed in the previous chapter. For modeling Referential Constraints we also do a Block Mapping where the ARBs and CPBs of V_F are mapped to those of V_D. The referential constraints ensure that for each block in V_F the number of distinct values along borrowed columns is upper bounded by the number of distinct values in the corresponding blocks in V_D. Once this is ensured, the exact subset property is ensured in the final Key Curation stage.

6.3.5 Summary Construction

Using the LP solution, we first build a summary data structure for each view that contains all the relevant information for extracting the corresponding base relation. Specifically, from the view summary the base relation summary is obtained by replacing the borrowed columns with the appropriate FK column. The process to do this is described next.

Key Range Curation. This final stage is responsible for the curation of FK values in F. Specifically, for each ARB a^F in V_F, to construct its equivalent in F, a range of FK values is assigned to it. This assignment is done using a range of PK values associated to a set of blocks in V_D, such that:

1. The chosen V_D blocks are contained within the boundaries of a^F after projecting along $\mathbb{B}$.
2. The tuples associated with the selected PK values have the desired number of distinct values along the PAS prescribed by the projection applied on a^F.

In this way, we get the summary for each table, which is used for dynamic data regeneration. A sample summary is shown in Figure 6.5. To appreciate the summary in entirety, we also show the other columns in the *Std* table schema. Therefore, we can see that the distinct row counts along different projection subspaces are represented similar to the discussion in the previous chapter. Additionally, we show the FK column in *Reg* table summary having the range of $RollNo$ values to be included.

6.4 Workload Decomposition

As discussed in the previous section, the case of conflicting projection constraints is handled by splitting the workload into sub-workloads such that each sub-workload is free from such

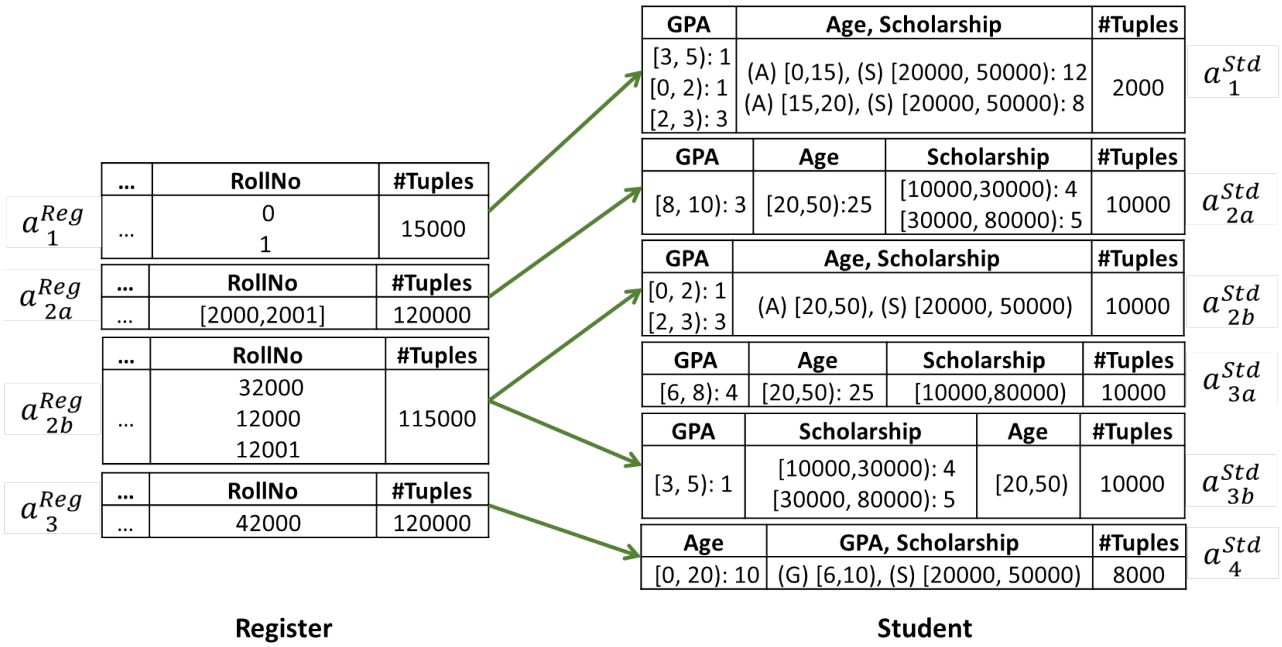


Figure 6.5: Sample Summary

conflicts. In addition to the characterization of such overlapping projections in case of single table queries (mentioned in Chapter 5), we have extended the class of overlapping projections to include the cases that appear in presence of joins. These additional cases of projection conflicts can be categorized based on the nature of referential dependencies as follows:

1F : 1D. Assume a pair of queries q_1 and q_2 having common fact (F) and dimension (D) tables, and the PASs applicable are $\mathbb{A}_1$ and $\mathbb{A}_2$ respectively, where $\mathbb{A}_1 \neq \mathbb{A}_2$. Further, the filter conditions in the queries intersect. In this case, q_1 and q_2 are conflicting if $\mathbb{A}_1, \mathbb{A}_2 \subseteq D$. The conflict arises because there is an implied projection dependency between F and D with respect to $\mathbb{A}_1 \cup \mathbb{A}_2$ as well. Therefore, F is subjected to projection constraints along $\mathbb{A}_1 \cup \mathbb{A}_2, \mathbb{A}_1$ and $\mathbb{A}_2$, which are overlapping.

multi-F : 1D. Assume a pair of queries q_1, q_2 with PASs $\mathbb{A}_1$ and $\mathbb{A}_2$. Further, both the queries involve a dimension table D such that the filters along D in the queries are overlapping. Now, if $\mathbb{A}_1, \mathbb{A}_2 \subseteq D$ and partially overlap with each other, then it is a straightforward case of overlapping constraints on D. Therefore, q_1 and q_2 form a conflicting pair of queries.

1F : multi-D. Assume a query q involving fact table F and dimension tables D_1 and D_2. Further, the PAS $\mathbb{A}$ applied on q is such that $\mathbb{A} \subseteq D_1 \cup D_2$ and $\mathbb{A} \nsubseteq D_1, \mathbb{A} \nsubseteq D_1$. To ensure referential integrity, projection constraints on F along $\mathbb{A} \cap D_1$ and $\mathbb{A} \cap D_2$ are required. Both these constraints conflict with the preexisting projection constraint along $\mathbb{A}$.

The conflicts in the $1F : 1D$ and multi-$F : 1D$ category can be handled by splitting the workload into sub-workloads. Specifically, we construct a graph with each query being a vertex and adding an edge between two queries if there is conflict between them. Now, if we run vertex coloring algorithm on the graph, the subset of queries having the same color assigned form a sub-workload.

Unlike the previous two conflicts which were inter-query, the third case of $1F : 1D$ type conflict is intra-query. A workaround to handle these queries is to generate all distinct tuples along $\mathbb{A} \cap D_1$ for filter compliant region of the dataspace in D_1, and along $\mathbb{A} \cap D_2$ in D_2. Subsequently, for F, the requisite number of distinct rows along $\mathbb{A}$ are generated by curating FKs from D_1, D_2. This is always possible since the distinct row cardinality along $\mathbb{A}$ in F can at most be the product of the distinct cardinality along $\mathbb{A} \cap D_1$ in D_1 and the distinct cardinality along $\mathbb{A} \cap D_2$ in D_2. Due to the distinct rows generation in D_1 and D_2, any other query with overlapping filters on D_1 (or D_2) will lead to a conflict. Again, we use workload decomposition to take care of these conflicts.

6.5 Align Refinement

To apply referential constraints on the various blocks of the fact table and dimension table views, we need to ensure that the blocks are well aligned. We next discuss the alignment process separately for the fact and dimension tables.

6.5.1 Fact Table Refinement

We know that the CPBs related to the same CC c need to be assigned disjointed set of values, even if their domains overlap. Therefore, for solution tractability, as a first step, we ensure that for the CPBs that are related to the same constraint c featuring PAS $\mathbb{A}$, have either identical or disjointed domains. This helps to divide the domain of $\mathbb{A}$, for a constraint c, into a set of intervals such that each constituent CPB is associated with an interval.

This CPB to interval mapping is done by ensuring that any two refined-blocks related to the same CC with PAS $\mathbb{A}$ are aligned with each other along $\mathbb{A}$. That is, they are either identical or disjoint with each other along the subspace spanned by $\mathbb{A}$. Specifically, two blocks r_1^F, r_2^F are considered aligned with each other if either $dom(\pi_{\mathbb{A}}(r_1^F)) = dom(\pi_{\mathbb{A}}(r_2^F))$ or $dom(\pi_{\mathbb{A}}(r_1^F)) \cap dom(\pi_{\mathbb{A}}(r_2^F)) = \emptyset$.

The Align Refinement module for a fact table view V_F takes the refined-blocks for the view as input and refines them such that the resultant blocks are mutually aligned. A block in V_F obtained after refinement is called an Aligned Refined Block (ARB) and is denoted as a^F. In Figure 6.3(a), we saw r_1^{Reg} and r_2^{Reg} participate in a common constraint comprising of projection

along GPA, and are not aligned along GPA. As a result, r_2^{Reg} is split in a_{2a}^{Reg} and a_{2b}^{Reg}, as shown in Figure 6.4(a) after which alignment is preserved. Further we can see that the red constraint is associated with a single interval $I_1 : GPA \leq 6$ and the green constraint is associated with two intervals $I_1 : GPA \leq 6$ and $I_2 : GPA \geq 6$.

The algorithm for performing this refinement iterates over each pair of refined-blocks (r_1^F, r_2^F) related to the same CC with PAS $\mathbb{A}$, and outputs four (if non-empty domains) blocks, the domains of which are represented as follows:

$$dom(\pi_{U(F)\setminus\mathbb{A}}(r_1^F)) \times (dom(\pi_{\mathbb{A}}(r_1^F)) \setminus dom(\pi_{\mathbb{A}}(r_2^F))),$$
$$dom(\pi_{U(F)\setminus\mathbb{A}}(r_2^F)) \times (dom(\pi_{\mathbb{A}}(r_2^F)) \setminus dom(\pi_{\mathbb{A}}(r_1^F))),$$
$$dom(\pi_{U(F)\setminus\mathbb{A}}(r_1^F)) \times (dom(\pi_{\mathbb{A}}(r_1^F)) \cap dom(\pi_{\mathbb{A}}(r_2^F))),$$
$$dom(\pi_{U(F)\setminus\mathbb{A}}(r_2^F)) \times (dom(\pi_{\mathbb{A}}(r_1^F)) \cap dom(\pi_{\mathbb{A}}(r_2^F)))$$

6.5.2 Dimension Table Refinement

Each ARB a^F along fact table view has to be given values for the borrowed columns that appear in the dimension table and are within its domain boundary. For this, we do a refinement of the refined-blocks in dimension table as well so that each resultant block is either completely contained in the domain of a block a^F (along $\mathbb{B}$) or is disjoint from it.

The Align Refinement module for a dimension table view V_D takes the refined-blocks for the view as input and refines them such that the resultant blocks are mutually aligned. A block in V_D obtained after refinement is also called an aligned refined block (ARB) and is denoted as a^D. In Figure 6.3(b), the refined-blocks r_2^{Std} and r_3^{Std} partially overlap with the blocks a_{2a}^{Reg} and a_{2b}^{Reg} (from Figure 6.4(a)) along the 2-D space (GPA, Age). Therefore, we split these blocks into a_{2a}^{Std}, a_{2b}^{Std}, a_{3a}^{Std}, and a_{3b}^{Std} ARBs, as shown in Figure 6.4(b).

The algorithm for performing this refinement iterates over each refined-block r^D one by one. For an r^D, each ARB a^F in V_F is compared along $\mathbb{B}$, and if $dom(\pi_{\mathbb{B}}(a^F)), dom(r^D)$ partially overlap, then r^D is broken into two blocks, the domains of which are represented as follows:

$$dom(r^D) \setminus dom(\pi_{\mathbb{B}}(a^F)), \quad dom(r^D) \cap dom(\pi_{\mathbb{B}}(a^F))$$

6.6 Block Mappings

To ensure referential integrity, we need to establish a mapping between the ARBs and CPBs in V_F and V_D. We define these mappings first.

6.6.1 Aligned Refined Blocks Mapping

An ARB a^F is related to an ARB a^D by a relation H iff the domain of a^D is contained into domain of a^F along the borrowed columns $\mathbb{B}$. That is:

$$(a^F, a^D) \in H \iff dom(a^D) \subseteq dom(\pi_{\mathbb{B}}(a^F))$$

For our running example, the ARB mapping is given as follows:

$$H = \{(a_1^{Reg}, a_1^{Std}), (a_{2b}^{Reg}, a_{2b}^{Std}), (a_{2b}^{Reg}, a_{3b}^{Std}), (a_{2a}^{Reg}, a_{2a}^{Std}), (a_{2a}^{Reg}, a_{3a}^{Std}), (a_3^{Reg}, a_4^{Std})\}$$

6.6.2 Constituent Projection Blocks Mapping

A CPB p^F is related to a CPB p^D by a relation J iff for each ARB a^F associated with p^F, there is an ARB a^D associated with p^D, such that, the domain of a^D is contained in the domain of a^F. That is:

$$(p^F, p^D) \in J \iff \forall a^F s.t.\ p^F L a^F, \exists a^D s.t.\ (p^D L a^D) \wedge (a^F H a^D)$$

Essentially all the CPBs in D that are related to a CPB p^F through J together form its domain. For our running example, the CPB mapping is given as follows:

$$\begin{aligned} J = \{&(p_1^{Reg}, p_1^{Std}), (p_1^{Reg}, p_3^{Std}), (p_2^{Reg}, p_1^{Std}), (p_2^{Reg}, p_2^{Std}), (p_2^{Reg}, p_3^{Std}), (p_2^{Reg}, p_4^{Std}), \\ &(p_3^{Reg}, p_1^{Std}), (p_3^{Reg}, p_3^{Std}), (p_3^{Reg}, p_5^{Std}), (p_3^{Reg}, p_6^{Std}), (p_4^{Reg}, p_7^{Std}), (p_4^{Reg}, p_8^{Std})\} \end{aligned}$$

6.7 Referential Constraints

The referential constraints are imposed on the ARBs and CPBs depending on the nature of projections applied. Therefore, we first classify ARBs into two main categories:

No Projection on a Subset of Borrowed Columns (NoPB) If an ARB a^F in V_F is either not subjected to a projection constraint, or the projection is along PAS $\mathbb{A}$ such that $\mathbb{A} \nsubseteq \mathbb{B}$, then a^F is included in the *NoPB* category. For example, a_3^{Reg} is not subjected to any projection along the borrowed columns. Therefore, it belongs to the NoPB category.

Projection on a Subset of Borrowed Columns (PB) If an ARB a^F in V_F is subjected to a projection constraint with PAS $\mathbb{A}$ such that $\mathbb{A} \subseteq \mathbb{B}$, then we call a^F to be in the *PB* category. For example, $a_1^{Reg}, a_{2a}^{Reg}, a_{2b}^{Reg}$ are all subjected to projection constraint along

GPA borrowed column. Therefore, these ARBs belong to the PB category.

6.7.1 NoPB Blocks

If there is a projection constraint applied on a block a^F along a PAS $\mathbb{A}$, which is not a proper subset of $\mathbb{B}$, then this means that there is at least one attribute in $\mathbb{A}$ that was present in the original schema of the fact table F itself. In this case, we replace the PAS from $\mathbb{A}$ to $\mathbb{A} \cap U(F)$, where $U(F)$ are the set of attributes in F. By ensuring distinctness for a subset of $\mathbb{A}$, we automatically get distinctness for $\mathbb{A}$ as well.

After the above pre-processing, each block a^F in the NoPB category has no projection constraint along any attribute of $\mathbb{B}$. Therefore, we can generate any single value along $\mathbb{B}$ that lies in the domain boundary of the block. In other words, we can pick any value from the ARBs in V_D that are related to a^F by relation H. To do this, we need to ensure that if a^F is populated then at least one related ARB a^D is also populated. This is ensured by the following constraint:

$$|a^F| \leq |F| \sum_{a^D:(a^F,a^D)\in H} |a^D| \tag{6.1}$$

Here $|F|$ is a trivial upper bound on a^F.

6.7.2 PB Blocks

Consider a block a^F on which a constraint with PAS $\mathbb{A}$, such that $\mathbb{A} \subseteq \mathbb{B}$, is applied. The distinct cardinality relationship has to be ensured at two levels for a^F – (a) Constituent CPBs level, and (b) Constraint level, stemming from the interdependence of the ARBs. Specifically, the following constraints are applied:

CPB Bound. For a CPB p^F, its cardinality is upper bounded the summation of the CPBs in D that form its domain. That is:

$$|p^F| \leq \sum_{p^D:(p^F,p^D)\in J} |p^D| \tag{6.2}$$

Constraint Interval Bound. For each interval I obtained wrt a CC c (after Align Refinement), the sum of the cardinalities of the CPBs wrt V_F that are associated with c, and have domain as I, is upper bounded by the sum of the cardinalities of the CPBs wrt V_D, which are related to any of the aforementioned CPB wrt V_F, using J. That is,

$$\sum_{p^F:dom(p^F)=I\wedge(p^F,c)\in MoL} |p^F| \leq \sum_{p^D:(p^F,p^D)\in J\wedge dom(p^F)=I\wedge(p^F,c)\in MoL} |p^D| \tag{6.3}$$

6.7.3 LP Constraints

The LP has variables with respect to each ARB and CPB in both the fact table and dimension table views. Specifically, for a CPB p, we have a corresponding variable $y(p)$ denoting $|p|$, and for each ARB a, we have a corresponding variable $x(a)$ denoting $|a|$. Therefore, replacing these cardinality expressions (in Equations 6.1, 6.2, and 6.3) with the corresponding variables, we get the referential constraints. The filter and projection constraints are modeled as discussed in the previous chapters.

For our running example, following are the LP constraints for the two example AQPs:

Filter Constraints

$$\begin{aligned}
x(a_1^{Reg}) &= 15000 \\
x(a_1^{Reg}) + x(a_{2a}^{Reg}) + x(a_{2b}^{Reg}) &= 250000 \\
x(a_1^{Reg}) + x(a_{2a}^{Reg}) + x(a_{2b}^{Reg}) + x(a_3^{Reg}) &= 370000 \\
x(a_1^{Std}) &= 2000 \\
x(a_1^{Std}) + x(a_{2a}^{Std}) + x(a_{2b}^{Std}) + x(a_{3a}^{Std}) + x(a_{3b}^{Std}) &= 42000 \\
x(a_1^{Std}) + x(a_{2a}^{Std}) + x(a_{2b}^{Std}) + x(a_{3a}^{Std}) + x(a_{3b}^{Std}) + x(a_4^{Std}) &= 50000
\end{aligned}$$

Projection Constraints

$$\begin{aligned}
y(p_1^{Reg}) + y(p_2^{Reg}) &= 2 \\
y(p_1^{Reg}) + y(p_2^{Reg}) + y(p_3^{Reg}) + y(p_4^{Reg}) &= 5
\end{aligned}$$

Referential Constraints

[NoPB]

$$x(a_3^{Reg}) \leq 370000 x(a_4^{Std})$$

[PB - CPB Bound]

$$\begin{aligned}
y(p_1^{Reg}) &\leq y(p_1^{Std}) + y(p_3^{Std}) \\
y(p_2^{Reg}) &\leq y(p_1^{Std}) + y(p_2^{Std}) + y(p_3^{Std}) + y(p_4^{Std}) \\
y(p_3^{Reg}) &\leq y(p_1^{Std}) + y(p_3^{Std}) + y(p_5^{Std}) + y(p_6^{Std}) \\
y(p_4^{Reg}) &\leq y(p_7^{Std}) + y(p_8^{Std})
\end{aligned}$$

[PB - Constraint Interval Bound]

$$y(p_1^{Reg}) + y(p_2^{Reg}) + y(p_3^{Reg}) \leq y(p_1^{Std}) + y(p_2^{Std}) + y(p_3^{Std}) + y(p_4^{Std}) + y(p_5^{Std}) + y(p_6^{Std})$$
$$y(p_4^{Reg}) \leq y(p_7^{Std}) + y(p_8^{Std})$$

In addition to these constraints, we also add the sanity constraints as discussed in the previous chapters. One possible solution to the LP is as follows:
[*Reg* ARBs]:

$$x(a_1^{Reg}) = 15000, x(a_{2a}^{Reg}) = 120000, x(a_{2b}^{Reg}) = 115000, x(a_3^{Reg}) = 120000$$

[*Std* ARBs]:

$$x(a_1^{Std}) = 2000, x(a_{2a}^{Std}) = 10000, x(a_{2b}^{Std}) = 10000,$$
$$x(a_{3a}^{Std}) = 10000, x(a_{3b}^{Std}) = 10000, x(a_4^{Std}) = 8000$$

[*Reg* CPBs]:

$$y(p_1^{Reg}) = 2, y(p_2^{Reg}) = 0, y(p_3^{Reg}) = 1, y(p_4^{Reg}) = 2$$

[*Std* CPBs]:

$$y(p_1^{Std}) = 2, y(p_2^{Std}) = 0, y(p_3^{Std}) = 4, y(p_4^{Std}) = 0,$$
$$y(p_5^{Std}) = 0, y(p_6^{Std}) = 0, y(p_7^{Std}) = 3, y(p_8^{Std}) = 4$$

6.8 Data Generation

From the LP solution, we get the following information for each view:

1. The cardinality for each ARB in the view.
2. The cardinality for each CPB in the view.

In this section we discuss how the database summary is constructed using this information and subsequently tuples are generated from it.

6.8.1 View Summary Construction

To begin with we first construct the view summaries independently using the same technique as discussed in Chapter 5. To summarize, we first assign domain intervals to each CPB. Recall

that two CPBs that are associated with a same constraint need to be disjoint even if there domains are overlapping. Therefore, during the domain interval assignment, we break the domain into disjoint intervals and assigned them to these CPBs. After this assignment, we have view summaries ready. In these summaries, we omit the domain assignment to the CPBs of the ARBs in V_F that belong to PB category. As a next step, we need to extract relation summaries back from the views. Specifically, we need to replace the borrowed columns in the views with the appropriate FK column.

6.8.2 Key Curation

A number of CPBs in V_F map to a CPB in V_D using the relation J. Since these CPBs may require disjoint domain intervals assigned to them, we further split the domain interval assigned to a CPB in the V_D based on the CPBs in V_F related to it. Specifically, we do the following:

1. We construct a graph G_p^D for each CPB p^D in V_D. Construct a vertex for each CPB p^F such that $(p^F, p^D) \in J$. An edge is added between two vertices if the corresponding CPBs are not related to a common constraint through MoL. In other words, these CPBs need not be disjoint and can therefore can be assigned the same values.

2. Extract maximal cliques from the graph. Number of maximal cliques correspond to the number of sub-domains in which the domain of p^D is to be broken. Let the cardinality share of p^D from the sub-domain s be expressed using the variable $y_{p^D}(s)$.

3. Each vertex in the graph can be a part of multiple maximal cliques. This means, the corresponding CPB p^F can be assigned values from multiple sub-domains. Let the share of p^F from a sub-domain s in the graph G_p^D be expressed using variable $y_{p^F}(p^D, s)$.

4. Now, we compute the values for all the variables $y_{p^D}(s)$, $y_{p^F}(p^D, s)$ across all the graphs and maximal cliques such that:

   - Summation of the share of a dimension table CPB p^D across all its sub-domains (maximal cliques) is equal to its total cardinality. That is:

     $$\sum_s y_{p^D}(s) = y_{p^D}$$

     .

   - Summation of the share of a fact table CPB p^F across all the sub-domains (maximal

cliques) in various graphs is equal to its total cardinality. That is:

$$\sum_{(p^D,s)} y_{p^F}(p^D, s) = y_{p^F}$$

- The cardinality share of a vertex is upper bounded by the cardinality share of the graph itself. That is:

$$y_{p^F}(p^D, s) \leq y_{p^D}(s)$$

We use the above information to break the domain of p^D into sub-domains and its associated cardinality divided wrt each sub-domain.

For our running example, considering only populated blocks, the domains of the CPBs of V_{Std} are as follows:

$$p_1^{Std} : [3, 6), \quad p_3^{Std} : [0, 3), \quad p_7^{Std} : [8, 10), \quad p_8^{Std} : [6, 8)$$

Among these, the graphs corresponding to both p_1^{Std} and p_3^{Std} have two cliques corresponding to the singleton vertices p_1^{Reg} and p_3^{Reg}. Therefore the domains of p_1^{Std} and p_3^{Std} are split into two sub-domains and the cardinality is also split. We show one possible split as follows:

$p_1^{Std}(\text{Split 1}) : [3, 5), \text{Card.: } 2$ $\quad$ $p_1^{Std}(\text{Split 2}) : [5, 6), \text{Card.: } 0$

$p_3^{Std}(\text{Split 1}) : [0, 2), \text{Card.: } 1$ $\quad$ $p_3^{Std}(\text{Split 2}) : [2, 3), \text{Card.: } 3$

Using these domain splits, we now show the domain assignments done to each of the populated CPBs of V_{Reg}:

$p_1^{Reg}(\text{Split 1}) : [3, 5), \text{Card.: } 1$ $\quad$ $p_1^{Reg}(\text{Split 2}) : [0, 2), \text{Card.: } 1$

$p_3^{Reg}(\text{Split 1}) : [5, 6), \text{Card.: } 0$ $\quad$ $p_3^{Reg}(\text{Split 2}) : [2, 3), \text{Card.: } 1$

$p_4^{Reg}(\text{Split 1}) : [8, 10), \text{Card.: } 2$ $\quad$ $p_4^{Reg}(\text{Split 2}) : [6, 8), \text{Card.: } 0$

Next, we need to populate the FK column for each ARB in V_F. We do this depending on the nature of ARB.

NoPB

If an ARB a^F is in the NoPB category, then we first pick any ARB a^D such that $(a^F, a^D) \in H$. In the summary of D, we compute the cumulative ARB cardinality from the beginning till a^D. This gives the number of tuples generated before a^D is generated. Let this value be γ. Since

PK values are generated from 0, the tuple with PK value γ will be the first tuple in a^D. Since any PK value in a^D is a valid value for the FK column wrt a^F, we assign the FK value γ to the entire a^F block.

PB

Now we consider the case where an ARB a^F is in the PB category. Each of its constituent CPBs are to be assigned FK value ranges. For a CPB p^F, we take its share with respect to each (p^D, s) combination. We then iterate on the ARBs in V_F that are related to a^F using H and get a target ARB that constitutes the CPB p^D. There we identify the fraction of p^D that features s. Now, we identify the number of tuples generated before this point. This is by adding the total ARB cardinality before the target ARB and then adding the total distinct cardinality for the CPBs before the specific p^D CPB. Further, the subdomain cardinalities before the specific s are also added. This gives the first FK value that we need. Say this is fk-val. Then the range assigned is [fk-val, fk-val+ $y_{p^F}(p^D, s)$.

In this way, for a particular fact table, a sample template for an ARB summary is shown in Figure 6.6. There is total cardinality of the ARB, and for every PAS acting on it, all the CPBs associated with different projection subspaces and their distinct cardinalities are maintained. Further, for the attributes not involved in any projections ($\mathbb{A}_{left}$), only the domain is stored without any distinct cardinality. What is different here from the no-join case, is that we now also store the value ranges for each Foreign Key column. Each range is wrt to a CPB and its corresponding CPB in dimension table.

$\mathbb{A}_1$	$\mathbb{A}_2$	...	$\mathbb{A}_\alpha$	$\mathbb{A}_{left}$	fk	
CPB-1: card.,	CPB-1: card.,	...	CPB-1: card.,	PB	Key Ranges	ARB Card.
CPB-2: card.,	CPB-2: card.,	...	CPB-2: card.,			
...	...	...	...			

Figure 6.6: Sample ARB Summary

A sample database summary for our running example was already shown in Figure 6.5.

6.8.3 Tuple Generation

The tuples are generated similar to how we discussed in Chapter 5. Additionally, for FK columns, we use the value ranges present in the summary and do a round robin on them until the entire ARB is instantiated.

6.9 Handling Select-Join Workload

If the input workload features only select and join operators, i.e. no projection operation, then certain optimizations become feasible. Specifically, following modifications are done:

One LP per View. In the existing solution, we had added referential constraints. These constraints required a unified LP for all the linked tables. In a pure SJ workload, we construct one LP per view and therefore, no referential constraints are added.

Ensuring Referential Integrity. Due to this independent solving of views, the views need not obey referential integrity. To address this concern, post LP processing, we add some spurious tuples to the view summaries – this leads to minor additive errors in satisfying volumetric similarity.

Additionally, note that since there are no projections, there are no conflicts in a workload. Therefore, we create a single database summary in the output for the entire workload.

6.9.1 View Summary Construction

Recall the way table summaries were constructed in the pure filter case discussed in Chapter 4. In the same way, we first get sub-view solutions from the LP solver and perform the Align-Merge algorithm. At this point, each view solution is comprised of a series of intervals (across various attributes) and the number of tuples in the region represented by these intervals. Next, we assign the entire cardinality to the *left boundaries* of the intervals.

6.9.2 Making View Summaries Consistent

Since the solution for each view is obtained independently, there could be inconsistencies across them. To address this problem, we first carry out a topological sort on the schema graph G_S and then iteratively make the current view consistent with its predecessors. To make a pair of views V_F and V_D consistent with each other, where V_F is dependent on V_D, we iterate over the rows in the view solution of V_F and look for the value combination that each row has for the attributes borrowed from V_D. If that value combination is not present in the solution of V_D, we add a new row in its solution with the corresponding NumTuples attribute set to 1. This results in an additive error in the total number of tuples in the view as compared to the original AQP at the client. But we hasten to add that the error is a fixed number of rows, determined by the nature of the constraints and the LP solution, and not by the data scale. Therefore, at Big Data volumes, the discrepancy can be expected to be minuscule, and our experiments empirically confirm this expectation.

This technique is similar to DataSynth. However, since the latter operates on complete database instantiations, and not just summaries, the time and space overheads incurred for making the views consistent can be large. Moreover, the additive error in this case is amplified due to the inherent sampling errors. Our experiments also capture this distinction between the errors incurred due to referential constraints in Hydra and DataSynth.

6.9.3 Key Curation

The non-key columns and PK are generated in the same way as discussed in Chapter 4. To instantiate foreign key values for a table F, its corresponding borrowed columns in V_F are considered. We iterate over the value combinations for this column-set in each row of the view summary. For an iteration, let the value combination be v. Now, we iterate over the view solution of the corresponding dimension table view V_D and find where v is present. We compute the cumulative sum of the cardinality entries till v is reached. This sum provides the fk-value to be inserted against v in F.

Again, DataSynth iterates over the complete instantiated (consistent) views to construct the corresponding materialized relations. This leads to considerable time and space overheads in contrast to our data-scale independent summary based approach.

6.10 Experimental Evaluation

In this section, we evaluate the empirical performance of a Java based implementation of our proposed solution. The Z3 solver [14] is invoked to compute the solutions for the LP formulations. Our experiments cover the accuracy, time and space overheads aspects of our work. The experiments were conducted using the PostgreSQL v9.6 engine [7] on a vanilla HP Z440 workstation.

Workload Construction. We designed a workload of 145 SQL queries derived from the TPC-DS decision support benchmark such that they satisfy the underlying assumptions. These queries covered four fact tables and their corresponding dimension tables. These fact tables were – STORE_SALES (SS), CATALOG_SALES (CS), WEB_SALES (WS), INVENTORY (INV). The distribution of queries among these four tables were: SS (73 Queries), CS (33 Queries), WS (32 Queries), INV (7 Queries). We have shown the snapshot of a fraction of the schema graph in Figure 6.7. We can see the four fact tables with their dimension tables in the figure.

These 145 queries led to a tally of 540 CCs. The constraints had PAS of upto length ten. The join distribution in these CCs is shown in Figure 6.8. As we can see from the figure, the number of joins ranges from 1 to 4.

We show a sample SQL query from our input workload below. The corresponding AQP is

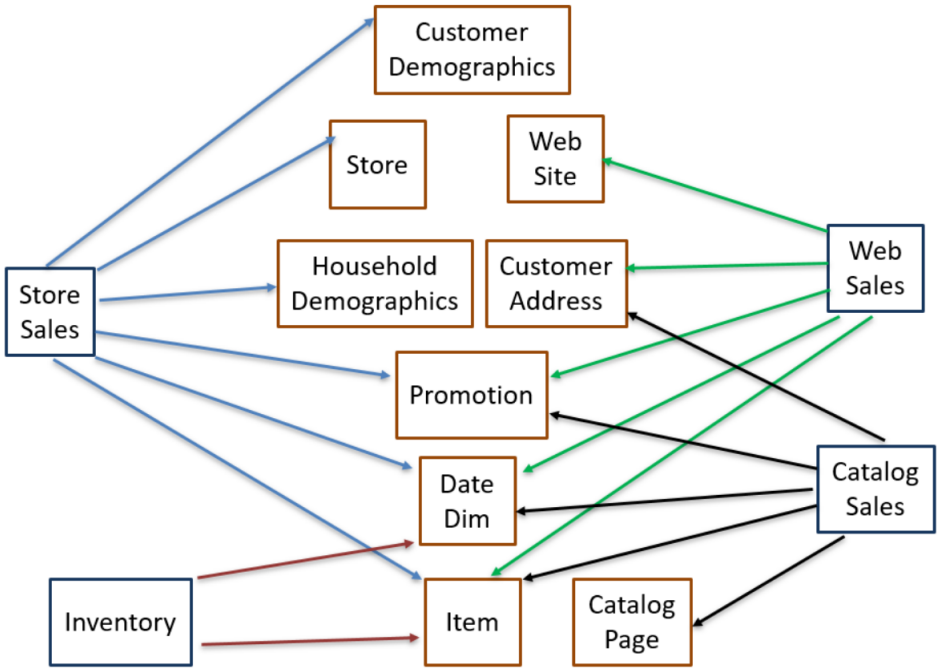


Figure 6.7: Schema Graph

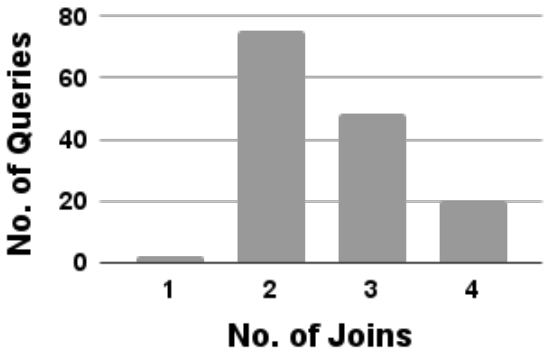


Figure 6.8: Distribution of Joins

also shown in Figure 6.9.

Sample SQL Query.

```
Select Distinct i_item_id
From store_sales SS, date_dim D, item I, customer_demographics CD, promotion P
Where ss_sold_date_sk = d_date_sk and ss_item_sk = i_item_sk
   and cd_demo_sk = ss_cdemo_sk and p_promo_sk = ss_promo_sk
   and d_year = 2001 and cd_gender = 'M' and cd_marital_status = 'M'
```

```
and cd_education_status = '4 yr Degree' and p_channel_email = 'N' ;
```

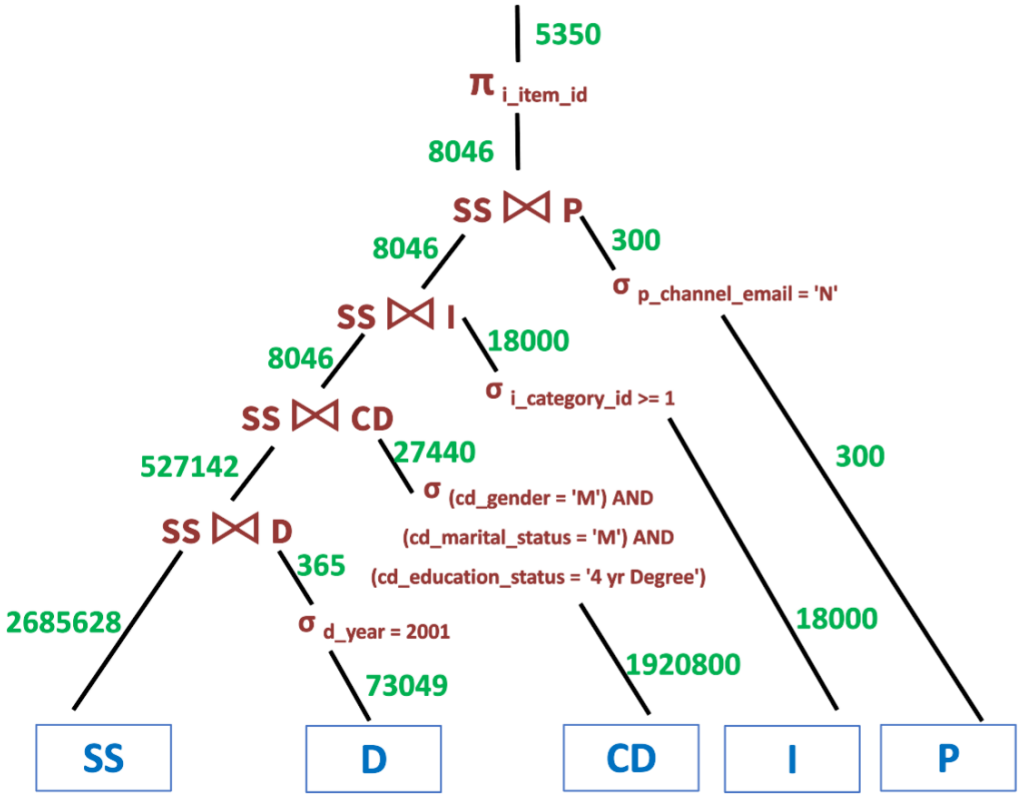


Figure 6.9: AQP of Sample Query

6.10.1 Workload Decomposition

We decompose the input workload into fifteen sub-workloads such that all the conflicts discussed are resolved. The complexity of these sub-workloads is quantitatively characterized in Table 6.3.

6.10.2 Constraint Accuracy

We ran our framework on the workloads mentioned before and the generated data satisfied all the constraints with **100% accuracy**. This is because – (a) additional constraints were included in the LP to ensure distinct cardinality relationships between fact tables and their corresponding dimension tables; (b) key curation ensured the explicit subset requirement, hence ensuring referential integrity.

6.10.3 Time and Space Overheads

We now turn our attention to the computational and resource overheads. The summary production times and sizes corresponding to all the sub-workloads are shown in Table 6.4. We see here

Table 6.3: Workload Characteristics (TPC-DS)

Workload No.	#Queries	#CCs	Max PAS Length	Max joins
1	13	56	2	5
2	10	45	4	5
3	8	33	10	4
4	8	26	3	4
5	11	44	5	4
6	10	38	2	5
7	9	22	3	3
8	10	32	4	3
9	10	47	4	5
10	9	41	6	5
11	8	30	2	5
12	7	20	2	4
13	8	27	6	5
14	14	39	2	4
15	10	40	5	5

that the end-to-end time for summary production for each sub-workload is less than a minute except for sub-workload 9 which is a couple of minutes. From a deployment perspective, these times appear acceptable since database generation is usually an offline activity. Moreover, the summary sizes, as shown in the table, are minuscule, within a few MBs. As mentioned earlier, these are independent of the data-scale of the original database.

To obtain a quantitative understanding, we report the sizes of the intermediate results at various pipeline stages also in Table 6.4 – specifically, the number of ARBs and CPBs created by Hydra are presented. The number of variables are simply the summation of these two quantities.

6.10.4 Performance of JOB Benchmark

We also evaluated the proposed solution on a workload of 35 SQL queries derived from the JOB benchmark [45, 5] based on IMDB dataset. This workload covered the three prominent fact tables – namely, MOVIE_KEYWORD (11 Queries), CAST_INFO (14 Queries), and MOVIE_COMPANIES (10 Queries), and their associated dimension tables. The number of joins in these queries range from 2 to 4. The workload led to a tally of 161 CCs.

The workload was split into 6 sub-workloads. The characteristics of these sub-workloads are shown in Table 6.5.

In this case as well, Hydra ensured 100% volumetric similarity. The computational and

Table 6.4: Overheads and Block Profile (TPC-DS)

Workload	#ARBs	#CPBs	#Variables	Summary	
				Time (s)	Space (MBs)
1	89	583	672	21	1.1
2	81	162	243	11	0.6
3	39	27	66	8	4.8
4	40	76	116	4	3.8
5	120	6485	6605	81	5.2
6	53	69	122	6	0.2
7	38	137	175	5	0.7
8	48	66	114	6	8.2
9	98	8356	8634	121	3.3
10	65	122	187	9	0.9
11	46	29	75	6	0.04
12	24	70	94	3	4.3
13	36	78	114	4	0.15
14	75	1473	1548	110	1.7
15	58	243	301	11	5.4

Table 6.5: Workload Characteristics (IMDB)

Workload No.	#Queries	#CCs	Max PAS Length	Max joins
1	5	27	3	3
2	6	33	3	4
3	7	30	2	3
4	6	24	3	3
5	6	28	2	3
6	5	19	2	3

resource overheads are tabulated in Table 6.6. We see that the time is at most a couple of minutes and the summary sizes are also typically within a few MBs.

6.10.5 Select-Join Workload

To showcase improvements in handling pure SJ workload, we also made another SJ-workload having 311 CCs derived from the TPC-DS benchmark. We compared Hydra against DataSynth for this workload. As mentioned earlier, in the SJ case, only a single database summary is produced because of the absence of conflicts that are induced by the projection operator.

We begin by investigating how closely the volumetric similarity, with regard to operator output cardinalities is achieved by the Hydra and DataSynth regenerators. This behavior is captured in Figure 6.10, which plots the percentage of CCs that are within a given relative error

Table 6.6: Time and Space Overheads (IMDB)

Workload	#Variables	Summary	
		Time (s)	Space (MBs)
1	94	1	0.4
2	66	1	0.3
3	1321	7	2.5
4	198	84	0.2
5	142	10	20
6	30	1	9.1

of volumetric similarity. From the plot it is evident that Hydra satisfies around 90 percent of the CCs with virtually no error, and the remaining CCs are also satisfied within a relative error of less than 10%. This is in contrast to DataSynth, which accurately satisfies around 80 percent of the CCs, but then incurs as much as 60% relative error to achieve complete coverage of the remaining CCs.

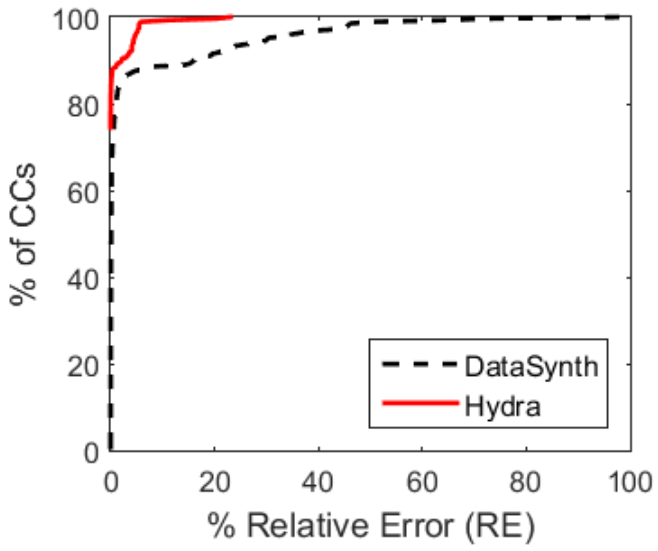


Figure 6.10: Quality of Volumetric Similarity

There are two reasons for the error-prone behavior of DataSynth: (1) the probabilistic sampling technique, and (2) the maintenance of referential integrity. While Hydra also is forced to insert additional tuples to maintain referential integrity, the number is substantially smaller than those injected by DataSynth. This is because the integrity errors are *amplified* by the impact of the sampling errors. This effect is quantified in Figure 6.11, where the number of extra tuples inserted is plotted on a log-scale for representative TPC-DS tables. We see here that Hydra is often an *order-of-magnitude* smaller with regard to the addition of these extra

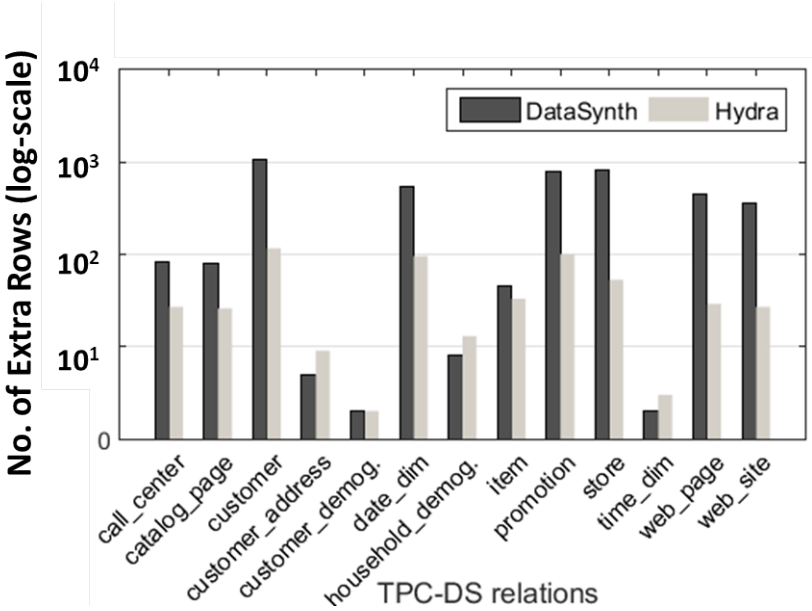


Figure 6.11: Extra Tuples for Referential Integrity

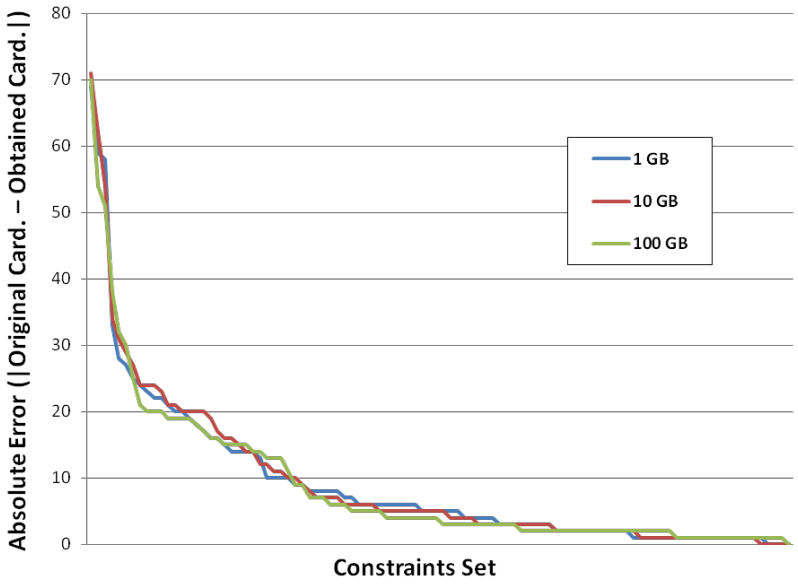


Figure 6.12: Distribution of Absolute Errors

tuples as compared to DataSynth. Recall that integrity errors in Hydra are independent of the data scale. Figure 6.12 shows the distribution of absolute volumetric errors for the constraints set at different database scales. We can see that this distribution is very similar for the three different scales. Hence, in the relative sense, the errors become minuscule as the data is scaled to large volumes.

6.11 Conclusion

Our work expands the scope of supported constraints to collectively include Select, Project and Join operators. The main challenge in handling joins in the CCs was to ensure referential integrity among tables. Our solution constructed a unified LP for the linked tables and added referential constraints that address the aforementioned challenge. Techniques of Align Refinement and Block Mapping aid in the process of adding these constraints. The experimental evaluation on workloads derived from TPC-DS and JOB benchmarks indicated that our solution accurately models the SPJ CCs and produces generation summaries within viable time and space overheads.

Chapter 7

Adding Robustness

7.1 Introduction

The applicability of the solution discussed so far is restricted to handling volumetric similarity on *seen* queries. Generalization to new queries can be a desirable requirement at the vendor site as part of the ongoing evaluation exercise. However, the Hydra design described thus far does not reflect this robustness objective because of (a) treating all candidate databases equally, (b) inserting an artificial skew in the database, (c) generating data that does not comply with the metadata statistics.

In this chapter, we extend Hydra to address these robustness-related limitations while retaining its desirable features of similarity, scalability and efficiency. As an initial effort, we describe the solution wrt a select-join input workload. Extending the ideas to include projection is an area of future work.

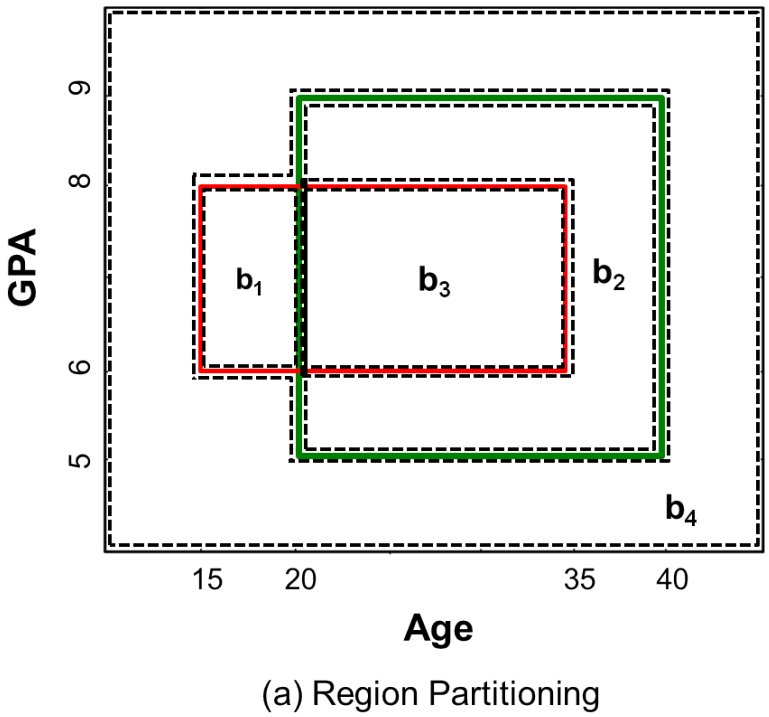
7.1.1 Limitations of Basic Hydra

In order to understand the limitations of the solution discussed thus far, we begin by briefly discussing the solution pipeline. Since, we are restricting to select-join workload, we omit the details of projection here. Hydra first constructs a denormalized version (without key columns) of each table, called a view. To generate a view, its data space is partitioned into a set of disjoint blocks determined by the filter predicates in the CCs. Further, a variable is created for each block, representing its row cardinality in the synthetic database. Next, a linear feasibility problem is constructed, where each CC is expressed as a linear equation in these variables. After solving the problem, a unique tuple within the block-boundaries is picked and replicated as per the block-cardinality obtained from the solution.

Example: Consider the following two CCs from Chapter 4 wrt the table *Std*.

$$c_1 : \langle f_1, 40000 \rangle \mid f_1 = (15 \leq Age < 35 \wedge 6 \leq GPA < 8)$$
$$c_2 : \langle f_2, 45000 \rangle \mid f_2 = (20 \leq Age < 40 \wedge 5 \leq GPA < 9)$$

We show these by a red box and green box in Figure 7.1(a). Accordingly, the LP problem constructed is shown in Figure 7.1(b).



(a) Region Partitioning

$$x_1 + x_3 = 40000$$
$$x_2 + x_3 = 45000$$
$$x_1 + x_2 + x_3 + x_4 = 50000$$
$$x_1, x_2, x_3, x_4 \geq 0$$

(b) Linear Program

Register		
...	RollNo	#Tuples
...	1	22000
...	45000	15000

Student	
Age, Scholarship, GPA	#Tuples
20, 20000, 5	5000
20, 20000, 6	40000
15, 20000, 5	5000

(c) Database Summary

Figure 7.1: Basic Hydra

The process of extracting relations from the views, while ensuring referential integrity (RI), forces the addition of some (spurious) tuples in the dimension tables. At the end, the output consists of concise constructors, together called as the *database summary*. An example summary is shown in Fig. 7.1(c).

As discussed above, the ability to generalize to new queries can be a useful feature for the vendor as part of the ongoing evaluation exercise. This is rendered difficult for the basic version of Hydra due to the following design choices:

No Preference among Feasible Solutions: There can be several feasible solutions to the linear feasibility problem. However, no particular solution is preferred over the others.

Moreover, due to the usage of Simplex algorithm internally, the solver returns a sparse solution, i.e., it assigns non-zero cardinality to very few blocks. This leads to very different *inter-block distribution* of tuples in the original and synthetic databases.

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

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

7.1.2 Our Contributions

In this chapter, we extend Hydra to address the above robustness-related limitations while retaining its desirable features of similarity, scalability and efficiency. Specifically, the linear *feasibility* problem is replaced with a linear *optimization* program that works towards finding a solution that is close to the metadata stats as well. Further, instead of generating just a few unique tuples, a more realistic spread of tuples over the domain are generated. Since we restrict to select and join operators, we also use the independent LP for each view strategy clubbed with referential integrity through spurious tuple addition.

The efficacy of the proposed solution is evaluated by comparing its volumetric similarity over unseen queries against the baseline of Hydra without the additional contributions of this chapter wrt robustness. Our results indicate a substantive improvement – for instance, the volumetric similarity on filter constraints of unseen queries improves by more than **30 percent**, as measured by the UMBRAE model-comparison metric [32]. Further, we also show that along with better generalization, the proposed solution also ensures metadata compliance and the generated data has no artificial skew. These experiments are conducted over a vanilla computing platform using the TPC-DS benchmark hosted on the PostgreSQL engine. Note that in this process, we also do not compromise on scale or efficiency - that is, we also support dynamic data generation using database summary, and the algorithm continues to be independent of the database size.

7.1.3 Notations

The main acronyms and key notations used in this chapter are summarized in Tables 7.1 and Table 7.2, respectively.

Table 7.1: Acronyms

Acronym	Meaning
AQP	Annotated Query Plan
CC	Cardinality Constraint

Table 7.2: Notations

(a) Database Related

Symbol	Meaning
T	Output Table
A	Attribute
V_T	View wrt T
F	Fact Table
D	Dimension Table
$\mathbb{B}$	Borrowed Attribute-Set
S	Sub-Table
$\mathcal{A}$	Attributes in a S
μ	No. of Metadata Constraints
$U(T)$	Set of attributes wrt the input table

(b) Workload Related

Symbol	Meaning
c	A CC
f	Filter Predicate
l	Output row card. after filter

(c) LP Related

Symbol	Meaning
b	filter-block
n	Number of filter-blocks
x_b	variable for b
C	LP Constraint

7.1.4 Organization

The remainder of this report is organized as follows: We present an overview of the proposed solution in Section 7.2, followed by its primary constituents – LP Formulation and Data Generation, in Sections 7.3 and 7.4, respectively. A detailed experimental evaluation is highlighted in Section 7.5. Our conclusions are summarized in Section 7.6.

7.2 Solution Overview

In this section, we provide an overview of the proposed solution. Using the metadata in the client input, the data space partitioning module constructs a refined partition, i.e. it gives finer blocks. Further, a linear program (LP) is constructed by adding an objective function to pick a *desirable* feasible solution. From the LP solution, the Summary Generator produces a richer database summary.

The overview of the modified components is described next.

7.2.1 Inter-block Distribution

Note that the original database also satisfies the feasibility problem, but there can be several assignments of cardinalities to the blocks such that all of them satisfy the constraints. For example, if we consider the constraints shown in Figure 7.1(b), both the following solutions satisfy the constraints:

$$\textbf{Solution 1:} \quad x_1 = 0, x_2 = 5000, x_3 = 40000, x_4 = 5000$$
$$\textbf{Solution 2:} \quad x_1 = 5000, x_2 = 10000, x_3 = 35000, x_4 = 0$$

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

7.2.2 Intra-Block Distribution

Within a block that gets a non zero cardinality assignment, we were generating a single unique tuple. As a result, it leads to inconsistent *intra-block distribution*. That is, even if the inter-

block distribution was matched between the original and synthetic databases, the distribution of tuples *within* a block can vary enormously. This is again not surprising since no efforts have been put to match the intra-block distribution. This not only affects the accuracy for unseen queries but single point instantiation per block also creates an artificial skew in the synthetic database. In the proposed solution, we instead try to obey the distribution based on the information extracted from metadata stats. At the finer granularity where volumes cannot be estimated any further, we resort to uniform distribution.

7.3 Inter-Block Distribution: LP Formulation

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

7.3.1 MDC: Optimization Function using Metadata Constraints

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

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

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

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

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

$$|\sigma_{A=u}(T)| = l_u$$

- Say for an attribute A, a bucket with a boundary $[\ell, h)$ and frequency B exists. Further, within this bucket, say two MCVs exist, namely u_1 and u_2 with frequencies l_{u_1} and l_{u_2}

respectively, then the following CC can be formulated:

$$|\sigma_{A\in[\ell,h)}(T)| = B + l_{u_1} + l_{u_2}$$

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

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

1. Run region partitioning using all the CCs, i.e. CCs from AQPs and metadata.
2. The CCs from AQPs are added as explicit LP constraints as before.
3. The CCs from metadata are modeled in the optimization function that minimizes the L1 norm of the distance between the output cardinality from metadata CCs and the cardinality from the sum of variables that represent the CCs. If there are n blocks (variables) together obtained from μ metadata CCs and Q AQP CCs, then the LP is as shown in Figure 7.2a. Here, $\mathbb{1}_{ij}$ is an indicator variable, which takes value 1 if block i satisfies the filter predicate in the jth metadata CC, and takes value 0 otherwise. Further, $C_1, C_2, ..., C_Q$ are the Q LP constraints corresponding to the Q AQP CCs. Finally, l_j represents the cardinality associated with the jth metadata CC.

7.3.2 OE: Optimization Function using Optimizer's Estimates

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

$$\text{minimize} \sum_{j=1}^{\mu} \epsilon_j \text{ subject to:}$$

1. $-\epsilon_j \leq \left(\sum_{i:\mathbb{1}_{ij}=1} x_i \right) - l_j \leq \epsilon_j, \quad \forall j \in [\mu]$
2. $C_1, C_2, ..., C_Q$
3. $x_i \geq 0 \quad \forall i \in [n], \quad \epsilon_j \geq 0 \quad \forall j \in [\mu]$

(a) MDC LP Formulation

$$\text{minimize} \sum_{i=1}^{n} \epsilon_i \text{ subject to:}$$

1. $-\epsilon_i \leq x_i - \tilde{x}_i \leq \epsilon_i \quad \forall i \in [n]$
2. $C_1, C_2, ..., C_Q$
3. $x_i, \epsilon_i \geq 0 \quad \forall i \in [n]$

(b) OE LP Formulation

Figure 7.2: Proposed LP Formulation

1. Same as Step 1 in MDC.

2. For each block obtained from (1), we construct an SQL query equivalent. Any query that can capture the block precisely is acceptable.

   As an example, the queries for the four blocks from Figure 7.1(a) are as follows:

```
Select * From Student Std
Where Age >= 15 and Age < 20 and GPA >= 6 and GPA < 8;
```

```
Select * From Student Std
Where Age >= 20 and Age < 35 and GPA >= 6 and GPA < 8;
```

```
Select * From Student Std
Where (Age >= 20 and Age < 40 and ((GPA >= 5 and GPA < 6)
or (GPA >= 8 and GPA < 9))) or (Age >= 35 and Age < 40 and
GPA >= 6 and GPA < 8);
```

```
Select * From Student Std
Where GPA >= 9 or GPA < 5 or Age >= 40 or Age < 15 or
(Age >= 15 and Age < 20 and ((GPA >= 5 and GPA < 6) or
(GPA >= 8 and GPA < 9)));
```

3. Once the block-based SQL queries are obtained, their estimated cardinalities are obtained from the optimizer. This is done by dumping the metadata on a dataless database using the CODD metadata processing tool [19], and then obtaining the compile-time plan (for example using EXPLAIN ⟨query⟩ command in Postgres).

4. We construct an LP that tries to give a feasible solution, i.e. that satisfies all AQP constraints, while minimizing the L1 distance of each block from its estimated cardinality as obtained from the previous step. Let the estimated cardinality for the n blocks be $\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_n$. Then the new LP that is formulated is shown in Figure 7.2b.

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

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

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

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

7.4 Intra-Block Distribution: Data Generation

We saw in the previous section how inter-block cardinality distributions were represented. In this section, we go on to discuss the strategy to represent the *intra-block* distributions.

The LP solution returns the cardinalities for various blocks within each sub-view. As discussed earlier, the blocks across sub-views need to be merged to obtain the solution at the view

level. After merging, the views are made consistent to ensure referential integrity (RI) and finally a database summary is constructed from these views. This summary is used for either on-demand tuple generation or, alternatively, for generating the entire materialized database. We briefly discuss these stages now.

7.4.1 Merging Sub-Views

Say two sub-views S_1 and S_2, with attribute-set $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, are to be merged. If the two attribute-sets are non-overlapping (i.e. $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$) then we can directly take the *cross* of the blocks in S_1 and S_2 to obtain the blocks in the view space. But when the intersection is non-empty, the merging of blocks happen among *compatible pair*. A block from S_1 is compatible with a block form S_2 if the two have an identical projection along the intersecting columns $\mathcal{A}_1 \cap \mathcal{A}_2$. Several blocks in S_1 and S_2 can have identical projection along the intersecting columns. Therefore, we define compatibility at a set level. That is a compatible pair is a set of blocks from S_1 and S_2 such that all these blocks have identical projections along the intersecting columns. For such a compatible pair, let $b_1^1, b_2^1, ..., b_{n_1}^1$ be the blocks from S_1 and $b_1^2, b_2^2, ..., b_{n_2}^2$ be the blocks from S_2. In the proposed solution, we merge each block b_i^1 with each block b_j^2 from a compatible pair and assign it a cardinality $\frac{|b_i^1||b_j^2|}{\sum_{i\in[n_1]}|b_i^1|}$. Note that $\sum_{i\in[n_1]}|b_i^1| = \sum_{j\in[n_2]}|b_j^2|$, which is ensured by consistency constraints in the LP. Hence in each pair, $n_1 \times n_2$ blocks are constructed. Merging of two blocks can be thought of as taking a join of the two blocks along $\mathcal{A}_1 \cap \mathcal{A}_2$. A sample sub-view merging is shown in Figure 7.3 where two sub-views (Age, GPA) and (Age, $Scholarship$) have five and four blocks respectively. Further, there are three compatible pairs that finally lead to eight blocks after merging.

Age	GPA	NumTuples
[15,20)	[8,10)	3000
[20,35)	[6,8)	40000
[20,35)	[5,6)	3000
[35,40)	[5,9)	2000
[15,20)	[0,6)	2000

×

Age	Scholarship	NumTuples
[15,20)	[4K,12K)	3000
[15,20)	[1K,4K)	2000
[20,35)	[8K,35K)	43000
[35,40)	[12K,70K)	2000

=

Age	GPA	Scholarship	NumTuples
[15,20)	[8,10)	[4K,12K)	1800
[15,20)	[8,10)	[1K,4K)	1200
[20,35)	[6,8)	[8K,35K)	40000
[20,35)	[5,6)	[8K,35K)	3000
[35,40)	[5,9)	[12K,35K)	1000
[35,40)	[5,9)	[35K,70K)	1000
[15,20)	[0,6)	[4K,12K)	1200
[15,20)	[0,6)	[1K,4K)	800

Figure 7.3: Sub-view Merging

7.4.2 Ensuring Referential Integrity

Since each view is solved separately, it can lead to RI errors. Specifically, when V_F, the fact table view, has a tuple where the value combination for the attributes that it borrows (say $\mathbb{B}$) from V_D, the dimension table view, does not have a matching tuple in V_D, then it causes an RI violation. Our algorithm for ensuring referential integrity is as follows:

1. For each block b^F of V_F, project the block along $\mathbb{B}$. Let the projected block be $\pi_{\mathbb{B}}(b^F)$.

2. Iterate on the blocks in V_D that have non zero cardinality and find all the blocks that have an intersection with $\pi_{\mathbb{B}}(b^F)$.

3. For each block b^D of V_D obtained from (2), split b^D in two disjoint sub-blocks b_1^D and b_2^D such that b_1^D is the portion of b^D that intersects with $\pi_{\mathbb{B}}(b^F)$ and b_2^D is the remaining portion. Cardinality of b^D is split between b_1^D and b_2^D using the ratio of their domain volumes. A corner case to this allocation is when the cardinality of b^D is equal to 1 – in such a case, we replace b^D with b_1^D.

4. If no block is obtained from (2), then we add $\pi_{\mathbb{B}}(b^F)$ in V_D and assign it a cardinality 1. This handles RI violation but leads to an additive error of 1 in the relation cardinality for dimension table. Collectively, these errors can be considered negligible. Also, they are independent of the scale of the database we are dealing with, and therefore, as the database size grows, the relative error keeps shrinking.

As we saw that the algorithm takes projections of blocks along borrowed set of dimensions. Since, the blocks neither have to be hyper rectangles nor have to be continuous, they may not be *symmetric* along the borrowed attribute(s), further adding to the complexity of ensuring consistency. If the set of the attributes in a fact table F is $U(F)$ and the set of attributes that are borrowed in V_F are $\mathbb{B}$, then a block b^F in V_F is symmetric along $\mathbb{B}$ iff:

$$b^F = \pi_{\mathbb{B}}(b^F) \times \pi_{U(F)}(b^F)$$

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

7.4.3 Constructing Relation Summary

Once we get the consistent summary for each view, where for each view we have the set of (symmetric) blocks and their corresponding cardinalities, we need to replace the borrowed

attributes in a view with appropriate FK attributes. Here the challenge is to remain in the summary world and still achieve a good span among all the FK values within a block. To handle this, instead of picking a single value in the FK column attribute for a block, we indicate a range of FK values.

Before discussing how the FK column value ranges are computed, let us discuss how the PK columns are assigned values. Each block in any view has an associated block cardinality. PK column values are assumed to be auto-number so, given two blocks b_1 and b_2, the PK column value ranges for the two blocks are 0 to $|b_1|-1$ and $|b_1|$ to $|b_1|+|b_2|-1$, respectively. Now, to compute the FK column values, for each block b^F in the fact table view V_F, the corresponding matching blocks from the dimension table view V_D are fetched. Once these blocks are identified, their PK column ranges are fetched and the union of these ranges is assigned to the FK column for b^F.

Recall that a block in its most general form can be represented as a collection of multi-dimensional arrays, where each multi-dim. array has an array of intervals for its constituent dimensions (relation attributes). For example, Figure 7.4 shows the structure of a block that is symmetric along dimension A. It can be expressed in words as the domain points where

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

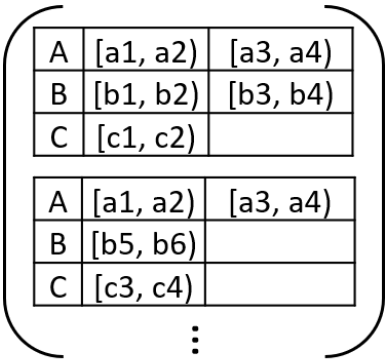


Figure 7.4: Block Structure

Now, we split the block into sub-blocks, where each sub-block is a single multi-dimensional array. Further, we divide the parent block's cardinality among the sub-blocks in the ratio of their volumes. In this way summary for each relation collectively gives the database summary.

7.4.4 Tuple Generation

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

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

Dynamic Generation. With dynamic generation, we cannot generate values randomly because the resultant tables will not be consistent across multiple query executions. Therefore, we generate values in a deterministic way. Based on the lengths of the intervals that are contained for an attribute in the block, the ratio of tuples to be generated from each interval is computed. Now, if ω values have to be generated within an interval, the interval is split into ω equal sub-intervals and the center point within each interval is picked. If the range does not allow splitting into ω sub-intervals, then it is split into the maximum possible sub-intervals, followed by a round-robin instantiation.

7.4.5 Comparison with Basic Hydra

For a compatible pair where we construct $n_1 \times n_2$ blocks (as described above), the basic Hydra may generate as few as $max(n_1, n_2)$ blocks. Therefore, it leads to several "holes" (zero-cardinality blocks) in the view, further leading to poor generalizations to unseen queries.

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

RI violations lead to addition of spurious tuples as we discussed. In the proposed solution, these violations are sourced primarily from *undesirable* solution, where the solver assigns a non-zero cardinality to a block in fact table for which the corresponding block(s) in dimension table has a zero cardinality. In such a case, then no matter how we distribute tuples within the blocks, it will always lead to a violation. However, earlier the choice of the single tuple that it instantiates within a block itself was not done carefully. Therefore, it has additional sources of RI violations.

Finally, the single point instantiation per block also generates an artificial skew and gives an unrealistic appeal to the data.

7.5 Experimental Evaluation

In this section, we empirically evaluate the performance of the proposed solution as compared to the baseline, i.e. basic Hydra. In these experiments, the Z3 solver was used to compute solutions for the linear programs. The performance is evaluated using a 1 GB version of the TPC-DS decision-support benchmark. The database is hosted on a PostgreSQL v9.6 engine, with the hardware platform being a vanilla standard HP Z440 workstation.

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

Table 7.3: Row Cardinality Distribution in Test CCs

We constructed a workload of 110 representative queries based on the TPC-DS benchmark query suite. This workload was split into *training* and *testing* sets of 90 and 20 queries, respectively. The corresponding AQPs for the training query set resulted in 225 cardinality constraints, while there were 51 such CCs for the testing query set. The distribution of the cardinalities in these AQPs covered a wide range, from a few tuples to several millions. The distribution of tuples for the CCs from test queries are shown in Table 7.3, with a separation into two groups – CCs that are pure selection filters on base relations, and CCs that involve such filters along with 1 to 3 joins. 2622 CCs were derived from metadata statistics, such as MCVs, MCFs and histogram bounds.

7.5.1 Volumetric Similarity on Unseen Queries

For evaluating the volumetric accuracy on the constraints derived from the unseen queries, we use the **UMBRAE** (Unscaled Mean Bounded Relative Absolute Error) model-comparison metric [32], with basic Hydra serving as the reference model. Apart from its core statistical soundness, UMBRAE's basis in the absolute error is compatible with the L1 norm used in the proposed optimization functions. UMBRAE values range over the positive numbers, and a value $\mathbb{U}$ has the following physical interpretation:

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

The value $\mathbb{U}$ is computed using the formula:

$$\mathbb{U} = \frac{MBRAE}{1 - MBRAE} \text{ , where } MBRAE = \frac{1}{Q}\sum_{j=1}^{Q}\frac{|e_j^P|}{|e_j^P|+|e_j^B|}$$

where $|e_j^P|$ and $|e_j^B|$ denote the absolute error for proposed solution and baseline respectively with respect to jth constraint; Q denotes the total number of constraints.

The UMBRAE values obtained by the proposed solution variants over the 20 test queries in our workload are shown in Figure 7.5, with basic Hydra serving as the reference baseline ($\mathbb{U} = 1$). For clearer understanding, the results for base filters, join nodes, and metadata statistics are shown separately. We see that the proposed solution delivers more than **30%** better performance on filters, while also achieving an improvement of over **20%** with regard to joins. The reason for the greater improvement on filters is that metadata is typically maintained on a per-column basis, making it harder to capture joins that combine information across columns.

On drilling down we found that as the number of joins increase, the improvement keeps reducing, as should be expected due to the progressively reduced quality of the input statistics. Also, we found that among the two proposed techniques, OE did better than MDC for constraint with multiple join predicates. Specifically, we found that OE did **33%** better than MDC for 2

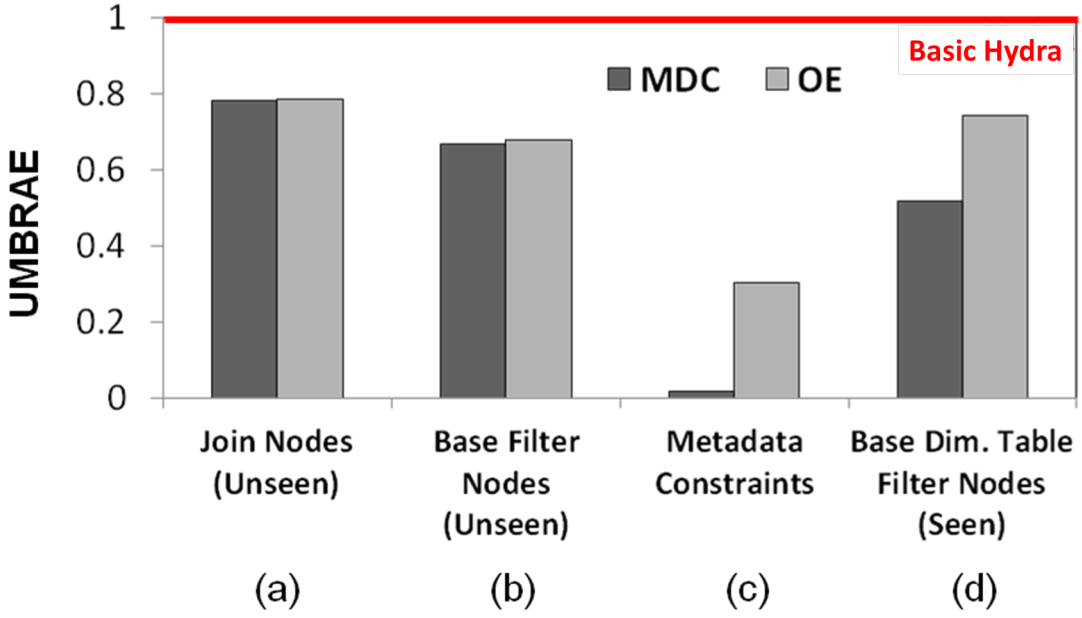


Figure 7.5: Volumetric Similarity on Unseen Queries and Constraints

join cases and **13%** better for 3 join cases. This is again expected since MDC is not sensitive to join constraints beyond first level joins while OE, due to its operations on the blocks directly, uses the optimizer's estimates for all the join constraints.

7.5.2 Metadata Compliance

We now turn our attention to evaluating the compatibility of the generated data wrt the metadata constraints. This is quantitatively profiled in Figure 7.5(c), and again we observe a very substantial improvement over the baseline – **98%** and **70%** for MDC and OE, respectively. Further, MDC outperforming OE is as expected because the former *explicitly* minimizes the errors in satisfying metadata CCs in its optimization function.

7.5.3 Database Summary Overheads

A key difference between the proposed solution and the baseline is with regard to the structure of the database summary. Firstly, the former has many more blocks in the summary. Secondly, instead of picking a single point per block, the summary retains the entire block and generates a wider spread of tuples intra-block. Due to these changes, a legitimate concern could be the impact on the size of the database summary and the time taken to generate the database from the summary. To quantitatively evaluate this concern, the space and time overheads are enumerated in Table 7.4. We see here that there is certainly a large increase in summary size, going from kilobytes to megabytes – however, in *absolute* terms, the summary size is still small enough to be easily viable on contemporary computing platforms. When we consider the time aspect, again there is an expected increase in the generation time from a few seconds to several tens of seconds, but here too the absolute values are small enough (sub-minute) to make the solution usable in practice.

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

Table 7.4: Space and Time Analysis

7.5.4 Data Scale Independence

The summary sizes and the time taken to generate the summary are independent of the client database size. We evaluate this by taking two instances of TPC-DS benchmark as the client database, namely 1 GB and 10 GB, and generating summary from both strategies MDC and OE. The results are enumerated in Table 7.5. While going from 1× to 10× with the database

size, the summary sizes and the generation time do not vary much. Also note that although the summary generation time looks large for OE, most of the time is incurred in getting a feasible solution from the LP Solver.

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

Table 7.5: Data Scale Experiment Analysis

7.5.5 Data Skew and Realism

Finally, to showcase the difference of skew, we drill down into the "look" of the data produced. For this purpose, a sample fragment of the TPC-DS Item relation produced by the baseline is shown in Figure 7.6a, and the corresponding fragment generated by the proposed solution is shown in Figure 7.6b. It is evident from the former figure has heavily-repeated attribute values – for instance, all the REC_START_DATE values are the same. In contrast, the proposed solution delivers more realistic databases with significant variations in the attribute values.

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

(a) Database (Item) from Baseline

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

(b) Database (Item) from Proposed Solution

Figure 7.6: Data Skew

7.6 Conclusion

In this chapter we looked into the problem of generating databases that are robust to unseen queries. In particular, we extended the existing Hydra framework by bringing the potent power of metadata statistics and optimizer estimates to bear on the generation exercise. In particular,

this information was captured in the form of additional cardinality constraints and optimization functions. The resulting fidelity improvement was quantified through experimentation on benchmark databases, and the UMBRAE outcomes indicate that proposed solution successfully delivers high fidelity databases.

Chapter 8

Hydra Architecture and Prototype Implementation

In the previous chapters, we saw the technical details of various contributions with respect to regenerating databases using SPJ cardinality constraints. We also discussed how Hydra incorporates the metaphor of "dataless databases" (proposed in [73]), whereby database environments are simulated without persistently generating and/or storing the contents. This is achieved by dynamically generating databases during query execution by using the minuscule database summary.

This dynamic generation provides Hydra a way to model two facets [2] of Big Data, namely, *volume* and *velocity*, which are of primary interest in the context of enterprise relational warehouses. Since the data is generated in memory, not just the large volumes of data can be easily generated, the velocity of data generation can be closely regulated, as compared to disk-resident databases. To complement dynamic database regeneration, Hydra also ensures that the process of summary construction is *data-scale-free*. Specifically, the summaries for complex Big Data client scenarios are constructed within just a few minutes.

On the implementation front, Hydra is completely written in Java, running to over 50K lines of code, and is currently operational on the PostgreSQL v9.6 engine [7]. Most of the functionality was achieved by leveraging existing APIs, with only the dynamic generation component requiring a few modifications within the engine. It has an intuitive user interface that facilitates modeling of enterprise database environments, delivers feedback on the regenerated data, and tabulates performance reports on the regeneration quality. The entire tool, including the source, is downloadable [1], and has been warmly received by both academic and industrial

[1] https://dsl.cds.iisc.ac.in/projects/HYDRA/index.html

organizations.

In the remainder of this chapter, we discuss the architecture, some implementation details, and the visual interface of the prototype tool.

8.1 Architecture

A pictorial view of it is presented in Figure 8.1 – in this picture, the green boxes represent the various modules of Hydra.

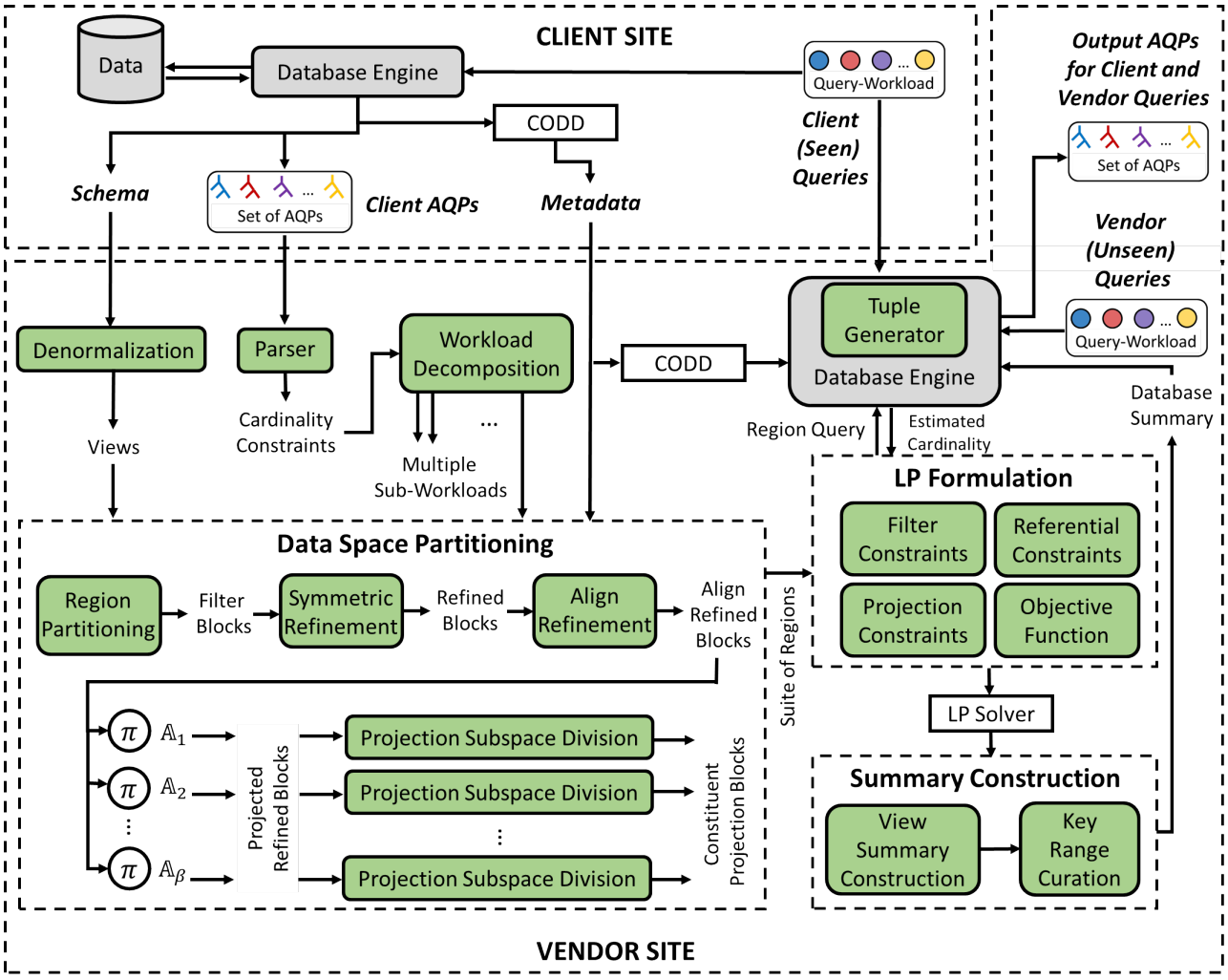


Figure 8.1: Hydra Architecture

8.1.1 Client Site

The information flow from the client to the vendor is as follows: At the client site, Hydra fetches the schema information, and the query workload with its corresponding AQPs obtained from

the database engine. The statistical metadata from the database catalogs is captured with the help of CODD [19] tool. All this information is then shipped to the vendor site.

To address the privacy concerns the above information can be passed through an appropriate anonymization layer at the client. For example, the table names, column names can be renamed. Likewise, the predicate constants can also be changed to any order preserving domain.

Note that if the client wishes to provide more than one AQP for the same query, that can also be done by giving all the AQPs for the query in the input. The volumetric constraints will be derived from all the nodes of each plan. The underlying assumption is that the AQPs will be mutually compatible, which is trivially true since the plans are derived from an original client deployment. Therefore, the constraints when modeled as an LP will also be feasible.

8.1.2 Vendor Site

Using the schema, views are constructed using the Denormalization module. The AQPs are converted to equivalent CCs using a Parser. The workload hence obtained is given to Workload Decomposition, which returns the set of non-conflicting CCs. Subsequently, the rest of the pipeline, comprising of Data Space Partitioning, LP Formulation and Summary Construction, is executed independently for each of these sub-workloads.

The Data Space Partitioning for a View and sub-workload begins with Region Partitioning followed by Symmetric Refinement algorithm. This gives the set of refined-blocks. Further, due to Align-Refinement across Fact and Dimension tables, the refined-blocks are split and give Align Refined Blocks (ARBs) in the output.

Let β be the total number of PASs across all the constraints, as indicated in Figure 8.1. For each PAS across all CCs, the PRBs are computed using the ARBs. These PRBs and sub-workload are then used by the Projection Subspace Division module to construct the set of CPBs.

Next, at the LP Formulation stage, an LP is constructed using variables representing the cardinalities of ARBs and CPBs. Specifically, Filter Constraints and Projection Constraints are modeled for each view. Subsequently, Referential Constraints are added between each pair of Fact and Dimension table. Finally, based on the cardinality estimation module of the database engine, an estimate of size of various blocks is obtained. Using these estimates, an Objective Function is added to the LP. This construction is then given as the input to the LP Solver. We have used the Z3 solver [14] from Microsoft for this purpose.

From the solution produced by the LP solver, a comprehensive table summary is constructed using the Summary Construction module. Specifically, it constructs View Summary and then replaces borrowed attributes in the fact tables with the corresponding foreign key columns using

the Key Curation module.

The summary is used by the Tuple Generation module to synthesize the data dynamically during query execution. Dynamic generation is useful if we are interested in analysis of intermediate operators in the query plan. If the focus is on scan operations, then materialized database instance should be generated from the summary. For experimental evaluation, we restricted the query plans to only have sequential scans.

8.2 Implementation Details

In this section we discuss the data structure and implementation detail of certain key concepts.

8.2.1 Domain Representation

The domain of a table is represented using a two dimensional vector as shown in Figure 8.2. Each row here represents a dimension (attribute) of the domain space. The vector corresponding to a dimension represents the split points inserted to break the dimension into intervals. Specifically, if the jth value in ith row is represented using $v_i[j]$, then the interval corresponding to it is $[v_i[j], v_i[j+1])$ [1] For example, from the figure we can conclude that dimension A_1 is broken into intervals $[0, 20), [20, 30), [70, 90), [90, A_1^{max})$, where A_1^{max} is the max value of the domain of A_1.

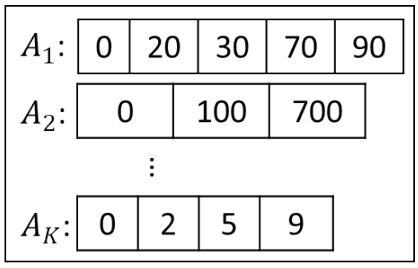


Figure 8.2: Domain Representation

The split points are inserted using the constants appearing in the filter predicates of the CCs. Specifically, if a predicate is represented as $\langle col \quad op \quad val \rangle$ then the split point is added based on the following condition:

- If $op =$ '$\geq$' or $op =$ '$<$', then val is inserted in the vector.
- If $op =$ '$>$' or $op =$ '$\leq$', then $(val + 1)$ is inserted in the vector.
- If $op =$ '$=$', then both val and $(val + 1)$ are inserted in the vector.

[1] If j is the last index then the $v_i[j+1]$ represents the max value of the domain.

Note that each filter predicate can be represented as a disjunction or conjunction of the sub-predicates of the $\langle col \quad op \quad val \rangle$ form. Therefore, the above split point insertion is done for each such sub-predicate. For example, Between SQL construct can be expressed using conjunction of two sub-predicates involving $\geq$ and $\leq$ operators; Likewise, In SQL construct can be expressed using disjunction of sub-predicates, each of which involves an $=$ operator.

The above procedure assumes that the columns are of integer type. In Hydra, we additionally deal with string, float, and date column datatypes. The predicate constants for such columns are mapped to an integer domain while preserving their relative order. Subsequently, post database summaries production, the column values can (optionally) be mapped to their original domain. Since there are finite number of predicates, this integer mapping is always feasible.

8.2.2 Region Data Structure

A primary data structure, used to represent a variety of objects in the implementation, is `Region` pictorially represented in Figure 8.3.

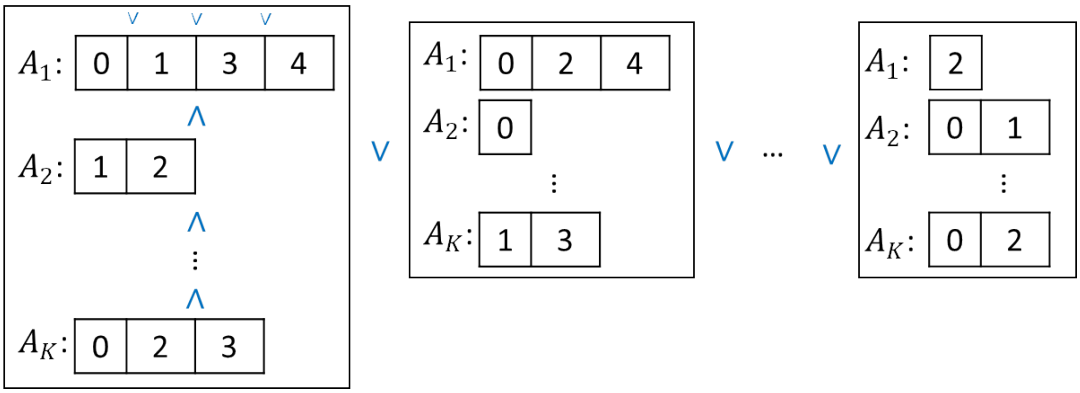


Figure 8.3: Region Data Structure

Each individual tabulation in this data structure is called as a `Bucket-Structure`. Like the domain representation, each `Bucket-Structure` is also represented as a two dimension vector where each row represents a dimension. The value in a cell of a row represents the index of the split point from the domain data structure. For example, the index value 1 for A_1 dimension will represent split point 20 as can be seen from Figure 8.2. As discussed earlier, each split point represents an interval. In a `Bucket-Structure`, these intervals are concatenated using disjunction intra-dimension and conjunction inter-dimensions. The first `Bucket-Structure` in Figure 8.3 shows the conjunction and disjunction with the help of the $\wedge$ and $\vee$ symbols, respectively. Further, the `Bucket Structures` are concatenated by disjunctions to represent the entire `Region`.

Note that a block in the partition need not be a contiguous hyper-rectangular region. It is simply a collection of points in the domain, which can be stored in a `Region` object. Therefore, this data structure is used to represent the various blocks (filter-block, refined-block, ARB, PRB, CPB). Hence, the output of the region partitioning algorithm is a collection of `Region` objects. Further, it is also used to represent the filter predicate region associated with a CC. At a constraint level, this modeling choice helps to handle DNF constraints easily.

8.2.3 Dynamic Tuple Generation Implementation

The summary is used by the Tuple Generation module to synthesize the data dynamically. This component resides inside the database engine, and needs to be explicitly incorporated in the engine codebase by the vendor. As a proof of concept, we have implemented it for the PostgreSQL engine by adding a new feature called *datagen*, which is included as a property for each table in the database. To enable this feature, the following command is executed on the Postgres engine:

```
alter table ⟨table-name⟩ set (datagen = 1);
```

Whenever this feature is enabled for a table, the scan operator for that table is replaced with the dynamic generation operator. As a result, during query execution, the executor does not fetch the data from the disk but is instead supplied by the Tuple Generator in an on-demand manner, using the available table summary.

8.2.4 Database Platform Portability

The work assumes relational data and the constraints including select, project and join operations. As long as the input constraints can be modeled to suit these requirements, there is no conceptual dependence on the database engine. To port to any other relational system, the implementation changes would be included in the following:

Plan Parser. To extract query plans standard APIs provided by the database engines are used. The Plan Parser in Hydra interfaces with these APIs and generates the cardinality constraints corresponding to the query plan.

Dynamic tuple generation. The module that replaces the traditional scan operation to give data on demand during query execution. This implementation has been done within PostgreSQL source code as it is open source.

8.3 Prototype: User Interface

In this section, we discuss the interface of the tool with a variety of visual scenarios that showcase the utility of the HYDRA tool.

8.3.1 Input from Client Site

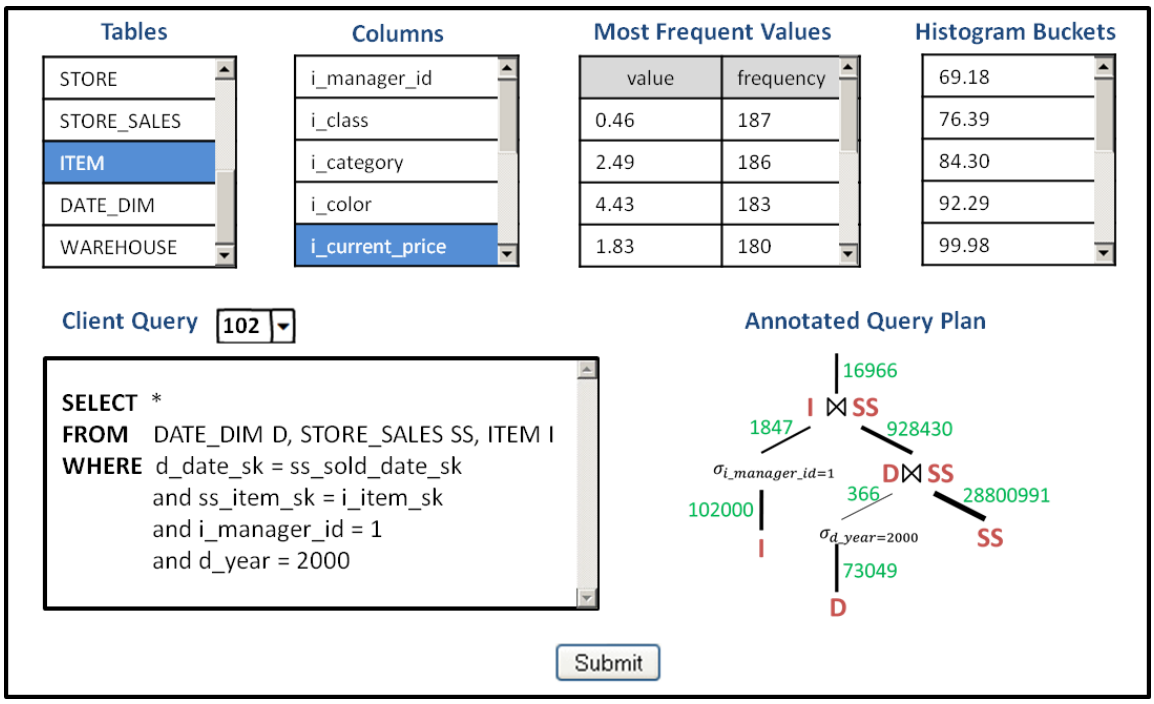


Figure 8.4: Client Site: Metadata, Queries and Annotated Query Plans

At the Client Site, the client supplies the query workload and the corresponding AQPs are obtained by optimizing and executing these queries on the client platform. Currently, the JSON format is supported to parse the execution plans. The next screen in the client interface is shown in Figure 8.4. In this figure, the top half profiles the metadata statistics. Specifically, the user can choose a specific table column, and the system presents the distribution of the most common values and the bucket boundaries of the equi-depth histogram for this column.[1] In the bottom half, the user can pick a query from the input workload (the figure shows a query on the TPC-DS schema), and the corresponding SQL text is displayed at the bottom left along with the associated AQP at the bottom right. The widths of the edges connecting the operators in the plan are scaled to visually indicate the volume of data flowing in each of these edges. Finally, once the user selects the SUBMIT button, all this information is transferred for

[1]The metadata visualization is customized for PostgreSQL [7].

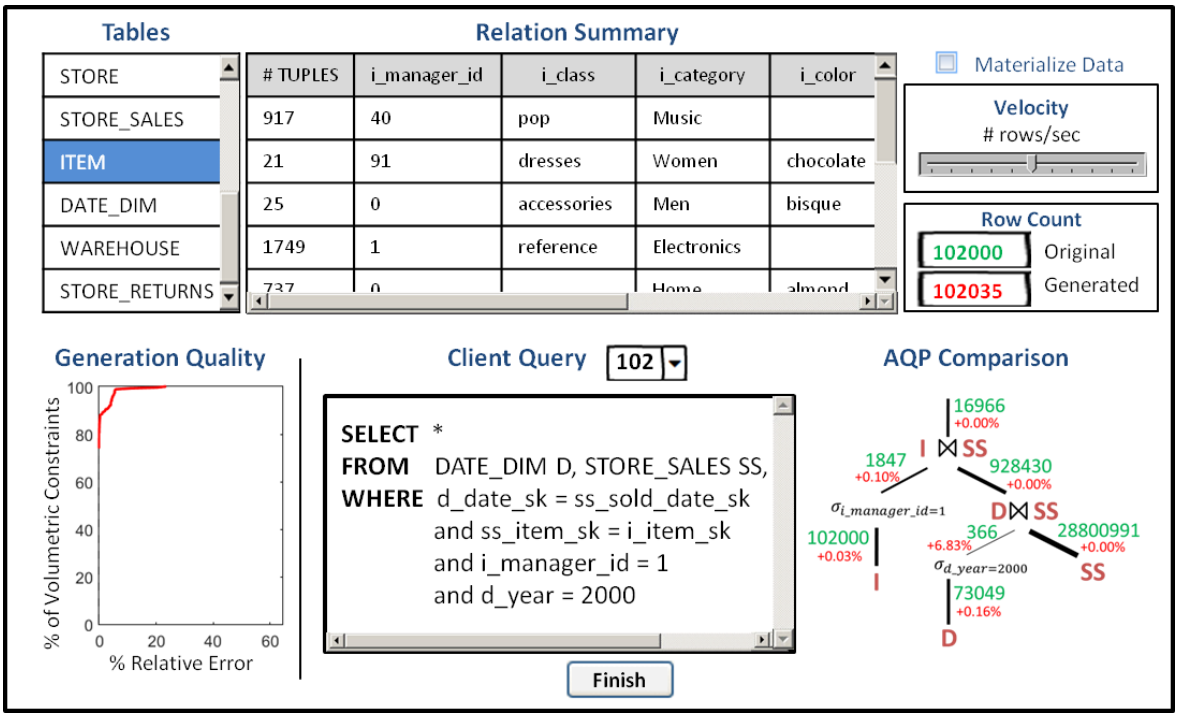


Figure 8.5: Vendor Site: Database Summary, Runtime Configuration Settings, Generation Quality and AQP Comparison

Vendor Site processing.

8.3.2 Vendor Site Processing

After receiving the above-mentioned information package from the client, the Vendor Site initiates the data regeneration process. Here, the primary interface during the LP solving stage tabulates the LP complexity in terms of their number of variables and run times. Subsequently, in the next screen, shown in Figure 8.5, the final database summary is displayed. The user can select an individual table, and the system shows its summary in the top middle panel. For instance, the first row in the item table summary in Figure 8.5 indicates that there are 917 rows with values `<40, pop, Music, ...>`. The PK columns are generated as auto-numbers.

Secondly, the top right panel shows the runtime configuration settings where the user can choose to either dynamically generate or optionally materialize the selected table. Also, for dynamic generation, the desired velocity, measured in rows per second, can be set using the slider bar. The chosen table's row count for the original and synthetic database are shown below the bar. To assess the overall quality of the regenerated data, the bottom left graph plots the percentage of volumetric constraints that are satisfied within a given relative error. Finally, the user can also drill down to a query-specific AQP comparison by selecting a query

from the drop down menu. In this mode, the corresponding SQL text and AQP are shown in the bottom middle and right panels, respectively. The edges in the AQP are annotated with the original cardinality in green color, and the relative errors (typically minor) incurred as a result of the regeneration are shown in red color.

8.3.3 Dynamic Database Regeneration

We would like to reiterate that the regenerated database has absolutely *no data* stored in the physical tables – i.e. the "dataless" approach. Instead, using our tuple generator, data is generated and supplied on-demand during query execution. As an example of the final outcome, a few sample rows for the initial columns of the ITEM table (highlighted in Figure 8.5) are enumerated in Table 8.1.

Table 8.1: Sample Tuples

item_sk	**i_manager_id**	**i_class**	**i_category**	**...**
0	40	pop	Music	...
917	91	dresses	Women	...
938	0	accessories	Men	...
963	1	reference	Electronics	...

8.3.4 Scenario Construction

Finally, Hydra also facilitates the vendor to pro-actively simulate anticipated client environments, by constructing synthetic AQPs through injecting cardinality annotations into the original client AQPs. For such "what-if" scenarios, Hydra creates the regeneration summary after verifying the feasibility of the synthetic assignments. This feature is particularly useful for testing the ability of the vendor's engine to robustly handle boundary condition scenarios and stressed Big Data environments.

8.4 Discussion

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

Multiple Summaries. We would ideally like to produce a single summary instance that satisfies all the CCs. However, Hydra 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, Hydra 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 blocks (variables) needed for expressing CCs, they can still be exponential if the input CCs have high overlaps. Hydra 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. 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.

Chapter 9

Extensions

9.1 Introduction

So far the focus has been on generating synthetic data that exhibits *volumetrically similar* behavior to the original database on a given query workload. That is, for every query in the workload, the output row cardinalities of individual operators in the corresponding query plans are similar in the original and synthetic data. While volumetric similarity captures a critical data characteristic necessary for mimicking client data processing environments, it lacks information such as data-skew and ordering of values. These can be important for mimicking performance of hash and sort operations. For example, various SQL constructs such as `Join, Group By, Distinct, Union` routinely require a hash based computation internally. The complexity of a hash-probe operation extensively depends on the amount of *duplication* in the data, especially when a hash bucket leads to spilling. Likewise, `Order By, Group By, Distinct, Union` constructs often use sorting operation internally. A sort operation's complexity also depends on how much data was *pre-sorted* in the input. This is because the movement of tuples and number of comparisons required during the sort are dependent on this feature.

9.1.1 Our Contributions

Keeping the above in mind, in this chapter, we focus on extending volumetric similarity to include other data characteristics, namely

1. Duplication Distribution
2. Presortedness

Specifically, for the above characteristics, we discuss the following:

1. Motivation for capturing the characteristics
2. Modeling the characteristics mathematically
3. Extracting the characteristics during query execution
4. Simple strategies for mimicking the characteristics in synthetic data

9.1.2 Notations

The key notations used in this chapter are summarized in Table 9.1. We use the acronym TAS to refer to Target Attribute-Set.

Table 9.1: Notations

Symbol	Meaning
$\mathcal{D}$	Output Database
$\mathbb{T}$	TAS
T	Output Table
A	Attribute
d	Duplication Value
e	Frequency of a Duplication Value
z	Duplication Distribution
δ	Length of z
h	Hash Table
X	Index Array
Y	Sorted Index Array
λ	Linearization of z
Δ	Distance btw two Duplication Distributions
ρ	Sortedness

9.1.3 Organization

In the upcoming sections we discuss each of the aforementioned characteristics sequentially. Specifically, Section 9.2 discusses the Duplication Distribution in detail. Further, Section 9.3 discusses the Presortedness characteristic. Finally, we conclude in Section 9.4.

9.2 Duplication Distribution

A duplication distribution, for a target attribute-set (TAS) $\mathbb{T}$ in Table T, is expressed as a 2-D vector that stores the information of how many value combinations corresponding to $\mathbb{T}$ occur with a certain frequency in T. For example, for a column $A = [4, 2, 3, 1, 4]$, the Duplication

Distribution vector will be $\{(1,3)(2,1)\}$. This is because the number of elements appearing once is three (values 1, 2, 3), while number of elements appearing twice is one (value 4).

Note that Duplication Distribution already encapsulates the total row-cardinality information. Therefore, ensuring matching Duplication Distribution implies volumetric similarity as well.

In the rest of this section, we discuss the Duplication Distribution characteristic in detail. Specifically, we begin with a motivating example. Followed by it, we discuss the mathematical expression and properties of Duplication Distribution. Subsequently, we discuss how to extract this information during query execution, and finally discuss how to model the characteristic in synthetic data generation.

9.2.1 Motivation

As discussed in Section 9.1, Duplication Distribution is important for modeling various operations such as Hash Join. To illustrate this, we constructed two sample datasets $\mathcal{D}_1$ and $\mathcal{D}_2$. Each dataset has a single column for each table from our running example schema. Specifically, we have $Std.RollNo$ and $Reg.RollNo$ columns in both $\mathcal{D}_1$ and $\mathcal{D}_2$. Recall that $Reg.RollNo$ is a foreign key column - that is, it takes values from the corresponding reference table column $Std.RollNo$. The frequency distribution of this foreign key column in the two datasets is specified below:

$\mathcal{D}_1$: $Reg.RollNo$ has a uniform distribution over all the values present in $Std.RollNo$.
$\mathcal{D}_2$: $Reg.RollNo$ has all identical values.

Both the datasets have 655 million and 82 million rows in Reg and Std tables, respectively. Now, consider the following simple SQL query:

```
Select * From Register Reg, Student Std
Where Reg.RollNo = Std.RollNo;
```

We executed the above query on $\mathcal{D}_1$ and $\mathcal{D}_2$ with the same underlying hardware, database platform (a popular commercial engine), and system configuration. We found that the query optimizer picked identical physical query plans, comprising of hash join operation, and same output row cardinalities. However, in spite of satisfying volumetric similarity, they have significantly different running times, as shown in Table 9.2. As we can see, for $\mathcal{D}_1$ the time was 18 minutes which increased to 28 minutes for $\mathcal{D}_2$. As per our judgement, the primary source of the time difference is the spilling behavior that happens while computing the hash table, which is used for performing the join operation. The spilling invocation depends heavily on the

Duplication Distribution of the data. Hence, our focus is on capturing Duplication Distribution in the synthetic data.

Distribution Type	Running Time
$\mathcal{D}_1$	18 min
$\mathcal{D}_2$	28 min

Table 9.2: Query Execution Time

9.2.2 Duplication Distribution Characterization

Now, we formally describe Duplication Distribution with respect to a TAS $\mathbb{T}$. Further, we also give a measure of computing distance between a pair of Duplication Distributions.

Expression

A Duplication Distribution with respect to a TAS $\mathbb{T}$ in table T is expressed using a 2D vector $\{(d, e)\}$, where d represents the number of duplicates, and e denotes the number of $\mathbb{T}$ values having d duplicates in T. For the above example, the (d, e) vector for $Reg.RollNo$ in $\mathcal{D}_1$ is {(8, 82 million)} as all 82 million values were uniformly distributed, and for $\mathcal{D}_2$ is {(655 million, 1)}, since all values are identical.

Duplication Distribution contains volume (total cardinality) information as well, and can be expressed as

$$\|z\| = \sum_{i=1}^{\delta} (d_i \times e_i) = |T|$$

where $z = \{(d_1, e_1), (d_2, e_2), \ldots, (d_\delta, e_\delta)\}$, and δ is number of (d, e) elements in z.

Bound on size of Duplication Distribution

The number of entries in the Duplication Distribution vector is equal to the number of distinct duplication frequencies for values occurring wrt $\mathbb{T}$ in T. We express this number as δ. It is easy to see that δ is maximum when the duplication frequency distribution is of the type: $\{(1, 1), (2, 1), (3, 1), \ldots, (\delta, 1)\}$. This gives us the following condition:

$$1(1) + 2(1) + 3(1) + \ldots, \delta(1) = |T| \tag{9.1}$$

This gives us $\delta = \mathcal{O}(\sqrt{|T|})$. Hence even for a trillion rows relation, the duplication frequency distribution of each attribute can be captured using few MBs of data in the worst case.

We also verified this experimentally by computing Duplication Distribution wrt each non-key attribute of four tables from 1 GB TPC-DS benchmark. The total size of Duplication

Distribution vectors was less than 40 KB. The detailed results are tabulated in Table 9.3. (The min, max and average δ values are with respect to various columns in the corresponding table.)

Table (T)	**Min. δ size**	**Avg. δ size**	**Max. δ size**	$\mathcal{O}(\sqrt{\lvert T\rvert})$	**Total tuples (M)**
store_sales	6	257	924	1620	2.6
catalog_sales	6	194	864	1195	1.4
customer	5	24	37	317	0.1
inventory	1	3	5	3428	11.7

Table 9.3: Duplication Distribution vector Size

Now, before we discuss how to compute similarity between two Duplication Distributions, let us first understand a prerequisite concept of linearization.

Linearlization

Linearization of a Duplication Distribution vector z is a linear (one-dimensional array) representation of it, represented as $\lambda(z)$. It is computed such that for each element (d, e) in z, d is added e times to the array $\lambda(z)$. Further, the values in the array are sorted in decreasing order. Notice that the number of element in $\lambda(z)$ is equal to the number of distinct values wrt TAS $\mathbb{T}$ in table T, where each index stores the frequency of an $\mathbb{T}$ value in T. Therefore, $\lambda(z)$ is just an exploded version of z.

To compare two Duplication Distribution vectors, we require linearizing them first. Further, this linearization should result in equal size arrays. In order to do that, we append zero values in the smaller array. Adding a zero value to the array implies there is an additional value appearing with 0 frequency. In other words, the value does not exist. Therefore, this addition does not change the semantic meaning of the array.

Distance between two Duplication Distributions

Given two Duplication Distribution vectors z_1 and z_2, the duplication distance between z_1 and z_2 can be defined as the summation of absolute difference between corresponding elements of $\lambda(z_1)$ and $\lambda(z_2)$. That is,

$$\Delta(z_1, z_2) = \frac{1}{2|T|} \sum_{i=1}^{|\lambda(z_1)|} |\lambda(z_1)[i] - \lambda(z_2)[i]|$$

Here we are assuming $\lambda(z_1)$ and $\lambda(z_2)$ have been made of equal length, as described above.

Distance Computation Example

Consider the following two Duplication Distributions:

$$z_1 : [(5,1),(4,2),(3,1),(1,2)], \quad z_2 : [(4,4),(2,1)]$$

The vectors obtained after linearization are as follows:

$$\lambda(z_1) : [5,4,4,3,1,1], \quad \lambda(z_2) : [4,4,4,4,2,0]$$

Therefore, the distance between z_1 and z_2 is given as:

$$\Delta(z_1, z_2) = \frac{|5-4|+|4-4|+|4-4|+|3-4|+|1-2|+|1-0|}{2*18} = \frac{4}{36} = 0.11$$

The term in the denominator of Δ expression has been added to normalize the distance such that it ranges from 0 to 1, with 0 being the minimum distance obtained for identical Duplication Distributions. The derivation of the bound is given next.

Bound on Distance

The two most distant Duplication Distributions are $z_1 : (|T|, 1)$ and $z_2 : \{(1, |T|)\}$. Here, $\{(|T|, 1)\}$, represents the case having all $\mathbb{T}$ values same. In contrast, $\{(1, |T|)\}$, represents the case where all $\mathbb{T}$ values are distinct. For this case, the distance $\Delta(z_1, z_2)$ is given as:

$$\Delta(z_1, z_2) = \frac{(|T|-1) + 1*(|T|-1)}{2*|T|} = 1 - \frac{1}{|T|}$$

From the above, we can conclude that Δ is always less than 1.

9.2.3 Duplication Distribution Extraction

All the database platforms give the information of input/output row cardinality for each operator in the query execution plan. However, the Duplication Distribution information is not provided explicitly. Therefore, to compute it, we use either (a) a non-invasive offline algorithm, where for each target operator[1] in the plan tree, a corresponding SQL query is constructed that returns the required duplication distribution at that operator, or (b) an invasive online algorithm, which computes the duplication distribution for the operator during the query exe-

[1]A target operator refers to an operator whose input data is to be used for Duplication Distribution computation. For example, a hash operator in the query plan can be a target operator as its efficiency is predicated on the Duplication Distribution of the attribute-set over which the hash is being computed.

cution itself. We have implemented this approach for open-source PostgreSQLv9.6 [7] database engine. Further, to make the computation efficient, we also have an approximation variant of the algorithm that gives approximate duplication distribution. We now describe both the offline and online strategies in detail.

Offline Approach

Applying `GROUP BY` with aggregate on $\mathbb{T}$ gives frequency count of each distinct $\mathbb{T}$ value in T and applying `GROUP BY` on the top of this frequency distribution gives the duplication distribution for $\mathbb{T}$. Specifically, following query is executed:

```
Select dup, count(*)
From (Select 𝕋, count(*) as dup FROM T Group By 𝕋)
Group By dup;
```

If the Duplication Distribution is to be computed for an intermediate node in the query execution (e.g. join output), then an additional where clause with the corresponding condition(s) is added to the inner query.

Online Approach

In online approach, we monitor the (intermediate) tables created during query execution and compute the duplication distribution wrt $\mathbb{T}$. Specifically, first a frequency counter (FC-1) is added to have the frequency count of each $\mathbb{T}$ value – this is same as computing aforementioned inner query result. Secondly, another frequency counter (FC-2) is applied on FC-1 for computing the Duplication Distribution.

To reduce the time and space overheads of the above solution, we also propose an *approximate* online approach. Here, instead of using frequency counter FC-1, we use Approximate Frequency Counter (AFC). AFC uses Lossy Counting [57] as an approximation algorithm, to keep track of most frequent items only. Thus, using AFC results in less number of elements after first frequency counting. This reduces space overheads. Further, second frequency counter, which runs over the result of AFC, gets benefited from this space reduction. This helps in reducing time overheads as well. However, since second frequency counter runs on AFC, it will give approximated Duplication Distribution. To make up for the loss in accuracy as a result of using AFC, the remaining cardinality is assumed to be filled by elements occurring once. From our observation, significant reduction in overheads were obtained as a result of using the approximation while only causing minor reduction in accuracy.

9.2.4 Duplication Distribution Mimicking

We now discuss the mechanism to mimic Duplication Distribution at the base table level. Depending on the constants appearing in the filter predicates, the domain of $\mathbb{T}$ can be divided into a set of intervals. Let this be the set $\mathbb{I}$. Further, let the Duplication Distribution vector $z = \{(d_1, e_1), (d_2, e_2), \ldots, (d_\delta, e_\delta)\}$. Now, let x_{ij} represent the fraction of e_i (with corresponding duplication value d_i) present in interval $I_j \in \mathbb{I}$. Therefore, following constraints need to be satisfied for each interval I_j:

$$|\sigma_{\mathbb{T} \in I_j}(T)| = \sum_{i=1}^{\delta} x_{ij} * d_i$$

Next, for each $e_i \in z$, following needs to be ensured:

$$|e_i| = \sum_{I_j \in \mathbb{I}} x_{ij}$$

The regions obtained from data space partitioning in Hydra can be split further so that the blocks do not span a split point along $\mathbb{T}$. Then, we can also say that the cardinality of rows inside each interval is equal to the summation of the row cardinalities of its constituent blocks.

If we wish to mimic duplication distribution for each histogram bucket maintained in the metadata stats, this can also be easily done by a simple extension of the above formulation. This is because the histogram buckets are mutually disjoint and can be mapped to the intervals easily.

9.3 Presortedness

In relational query processing, *sort* is a commonly used operator. It is not only used for handling *order by* SQL clause, but also used often as a component for various other operators like joins (sort-merge), or projection operator based constructs (group by, distinct, union). Sort operation in a query plan can have significant impact on the total execution. The execution time of sort operation depends on (a) number of comparisons among the data elements, and (b) movement of tuples while sorting. These aspects are influenced by the extent of ordering pre-existing in the input data – in other words, the Presortedness in the data.

In this section, we discuss Presortedness characteristic in detail. Specifically, we begin with a motivating example followed by modeling Presortedness mathematically. Subsequently, we discuss how to extract this information during query execution, and finally how to model it in synthetic database generation.

9.3.1 Motivation

To demonstrate the impact of Presortedness, we performed an experiment on the *INVENTORY* table of the TPC-DS [12] benchmark which has size 8.4 GB and 400M plus tuples. We picked an attribute (*inv_qty_on_hand*) from the table and created a new table ($T(A_1, A_2)$) with two attributes, both having the all values of *inv_qty_on_hand*, with one column having elements in sorted order, while the other column has the same values in random order. Then, we ran *ORDER BY ASC* and *ORDER BY DESC* query on both attributes and the time taken by queries is in Table 9.4. Here, Column Order represents the order from the data and Sort Order represents the order from the query. As can be seen from the table, in the first case of ascending sort and column order, the query took the least amount of execution time. This is because, unlike the other cases, here there was no tuple movement involved during the sort.

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

Table 9.4: Query Execution Time on Different Column Order and Sort Order

9.3.2 Presortedness Characterization

Let $\mathbb{T}$ be the TAS along which the order is being considered. Further, let the value-combination present in the ith row of input table T be represented by $T[i]$. Next, let $\tilde{T}$ be the sorted counterpart of T along $\mathbb{T}$. Now, let there be an *Index Array* X that stores the index of the corresponding value from T in $\tilde{T}$. That is, $X[i]$ stores the index value of $T[i]$ in $\tilde{T}$. Finally, let Y be the sorted counterpart of X. Our aim is to compute correlation between the arrays X and Y. Note that $|X| = |Y| = |T|$.

To compute correlation between the two arrays X and Y, we use the Pearson Correlation Coefficient [6], represented by ρ. Therefore, the Presortedness between T and $\tilde{T}$ is given by:

$$\rho = \frac{\sum_{i=0}^{|T|-1}(X[i] - \overline{X})(Y[i] - \overline{Y})}{\sqrt{\sum_{i=0}^{|T|-1}(X[i] - \overline{X})^2}\sqrt{\sum_{i=0}^{|T|-1}(Y[i] - \overline{Y})^2}} \tag{9.2}$$

Here, $\overline{X}$ and $\overline{Y}$ represent the mean of the arrays X and Y, respectively. Note that Y is an array of values $[0, 1, 2, .., (|T|-1)]$. Further, $\overline{X} = \overline{Y} = \frac{|T|(|T|-1)}{2}$.

Example

Let the input array of values be [40, 30, 20, 10]. Its sorted version is [10, 20, 30, 40]. Therefore, the index array (X) will be [3, 2, 1, 0]. Further, the Pearson's correlation coefficient ρ = -1.

Having $\rho = 1$ means the data is already sorted, and $\rho = -1$ means data is sorted in the opposite order, and having $\rho = 0$ means maximum randomness in the data. In general, positive value implies that there are a greater number of elements closer to their position in the sorted array and negative correlation means farther from it.

9.3.3 Presortedness Extraction

To compute the Presortedness of a TAS used by the `sort` operator in the query plan, we monitor the edge which gives input to the sort operator. Hashtable(H) maps the attribute values to their corresponding indices in the input list (unsorted) to the sort operator. Once the sort node is processed, each tuple comes in order and helps to create the Index Array for the input unordered list. Subsequently, the index array is passed to Pearson correlation coefficient calculator. This flow is shown in Figure 9.1. The shaded blocks in the figure are the modules added in the Postgres execution engine.

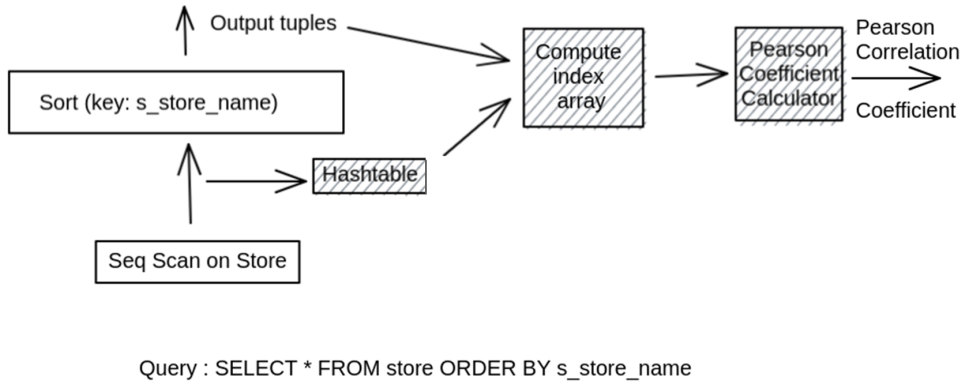


Figure 9.1: Presortedness Computation on a Query Execution

We now describe the details of each of these contributed modules.

Hashtable(h). The hashtable (h) is a data structure having *key-value* pair, where *key* is a TAS value-combination and *value* is the list of all indexes having the corresponding TAS value-combination. Hashing is performed on the input tuples to the sort operator. For

example, given TAS values: [30, 10, 20, 40, 30], then h will have (30,[0,4]), (10,[1]), (20,[2]), (40,[3]).

Compute Index Array (X). Once the sort operation is performed, the output tuples are ordered on TAS . So for each output tuple, we pick its TAS value-combination and using h, get the corresponding index values for unordered set and add them to the Index Array. For the above example, the sorted order is [10,20,30,30,40], and the Index Array X is [1, 2, 0, 4, 3].

Pearson Coefficient Calculator. Once the Index Array is constructed, it is passed to Pearson correlation coefficient calculator. Here, the Presortedness is computed using the formula in Equation 9.2.

Query#	$\|T\|$	**Execution Time Original**	**Time with ρ Computation**
1	12	0.1 ms	0.2 ms
2	50000	0.7 s	0.9 s
3	100000	0.9 s	0.9 s
4	2622614	9 s	10 s
5	11745000	28 s	29 s

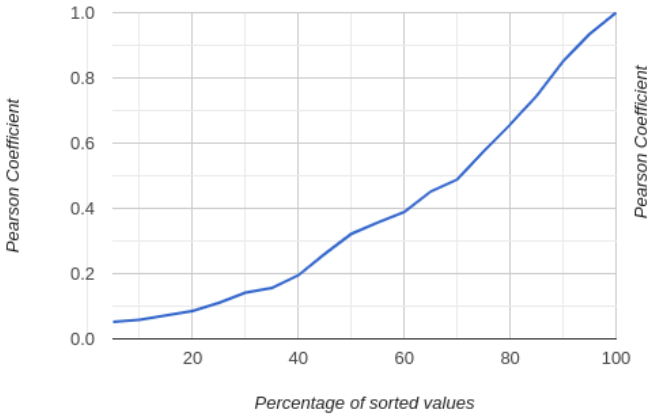
Table 9.5: Execution Time of Order By Queries

The time overheads incurred due to the additional code for Presortedness computation in PostgreSQL are shown in Table 9.5. We can see that the overheads are viable. Further, they can be further reduced by using the native data structures of PostgreSQL.

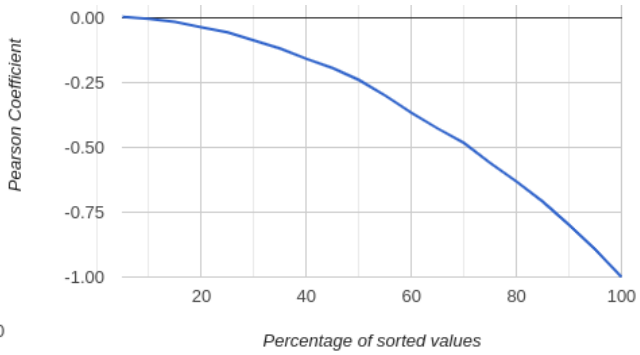
9.3.4 Presortedness Mimicking

We begin by establishing a relationship between percentage of sorted tuples vs Presortedness value ρ. We start with an array of $|T|$ tuples [0 to ($|T|-1$)] and shuffle it to get its ρ value close to 0. Now, we fetch different percentage of data from the array, sort it, and replace them back at the same locations in the array, but in sorted order. That is, if the tuples were fetched from locations $i_i, i_2, \ldots, i_Z$, the first tuple in the sorted order is inserted on location i_1; second on i_2, and so on.

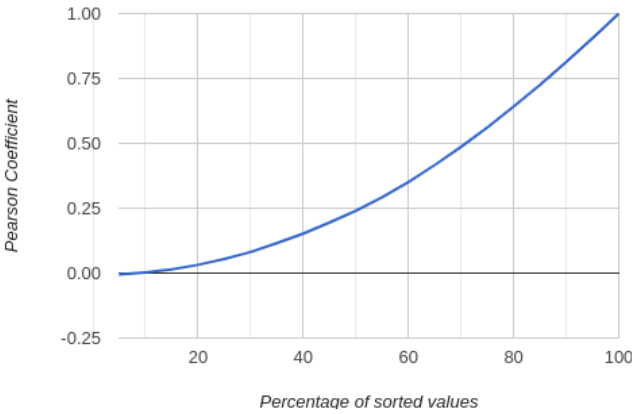
The above exercise is performed for varying percentage of tuples and $|T|$ value. Also, both ascending and descending order is considered. The relationship obtained is depicted in Figure 9.2. In these figures, the Pearson coefficient for each percentage is obtained by averaging over different set of fetched tuples.



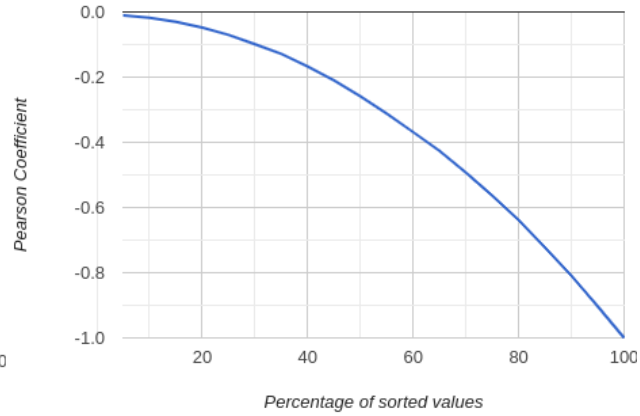
(a) Ascending, $|T|= 100$



(b) Descending, $|T|= 100$



(c) Ascending, $|T|= 10000$



(d) Descending, $|T|= 10000$

Figure 9.2: Presortedness vs. Percentage Presorted Tuples

The next step is to order the table T such that it gives the desired Presortedness value ρ^*. For this, the required percentage of tuples is sorted in T. To compute the required percentage, we use the inverse of the aforementioned relation of percentage of sorted tuples vs Presortedness value. For a specific percentage, the tuples are chosen at random. We performed some preliminary experiments with the implementation of the ideas presented. The results comparing the desired vs obtained Presortedness are shown in Table 9.6. We can see that the computed correlation coefficient is quite close to the actual correlation coefficient, thus this algorithm gives a promising direction.

9.4 Conclusion

In this chapter we went beyond volumetric similarity and discussed two other data characteristics – namely, Duplication Distribution and Presortedness, which are important for mimicking

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

Table 9.6: Comparing Expected vs Obtained Presortedness

client data processing environments. Specifically, we discussed their mathematical characterization, mechanism of extraction and simple strategies for modeling them for data generation.

Chapter 10

Conclusion and Future Directions

The ability to synthetically regenerate databases that accurately conform to the volumetric behavior on queries at client sites is of crucial importance to database vendors, especially with the advent of Big Data applications. Existing database regeneration frameworks are crippled by limitations on several fronts such as inability to (a) handle queries based on core SPJ relational algebra operators; (b) scale to big data volumes; (c) scale to large input workloads; and (d) provide high accuracy on unseen queries. The contribution of this thesis is a new data regeneration framework, namely Hydra, that materially addresses these challenges by adding functionality, dynamism, scale, and robustness.

In this final chapter, we summarize our conclusions, followed by outlining a number of potential areas for further research to achieve the larger goal of scalable, efficient and robust database regeneration.

10.1 Conclusions

In this thesis, we provide a comprehensive solution to support queries based on SPJ relational algebra operators. The volumetric constraints are modeled using an LP, in which each variable represents the volume of a region of the data space. The regions are computed using a suite of data space partitioning strategies – namely, Region Partitioning, Symmetric Refinement, Align Refinement and Projection Subspace Division. Additionally, referential constraints are applied over denormalized equivalents of the tables. Along the way, we ensured that the partitioning algorithms at various stages have provable optimality guarantees to keep the complexity of the LP low, thereby providing workload scalability.

We also proposed a *dynamic* approach to data regeneration where we generated a minuscule database summary as the output rather than the static data itself. This summary can be

used for dynamically generating data during query execution. Therefore, the enormous time and space overheads incurred by prior techniques in generating and storing the data before initiating analysis are eliminated. Further, in order to retain the spirit of data scalability, we also ensured that the entire regeneration pipeline is *data-scale-free*.

To improve accuracy towards unseen queries, metadata statistics were exploited to facilitate careful selection of a desirable database from the candidate synthetic databases and also provide metadata compliance.

We also discussed the details of the Java-based prototype implementation of Hydra and showed its performance on both real-world and synthetic benchmarks.

Finally, we also discussed other plausible data characteristics that are useful for mimicking client data processing environments. Specifically, we discussed two potent characteristics, namely, Duplication Distribution and Presortedness.

Closing Statement. In closing, Hydra provides the novel approach of dynamic database regeneration which guarantees volumetric similarity while eschewing the time and space overheads typically incurred in the process. The tool has been warmly received by both academic and industrial communities.

10.2 Future Directions

We now turn our attention to some interesting future directions that can further data regeneration goals and solutions.

1. **Using Approximation Algorithms.** Hydra 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. These can include techniques such as:

   Constraints Merging. The number of variables grow exponentially with the number of constraints. Therefore, one promising way of reducing the complexity of the problem is by merging the constraints that are *similar*. One plausible similarity metric is the density of data. For instance, if two constraints are similar in their data density, then they can be merged by mildly compromising on the associated output cardinalities.

   Greedy Partitioning. 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.

2. **Directing LP Solution.** We currently use the solver as a black box to get a solution that satisfies the constraints and is close to the metadata constraints. An alternate setting could be to develop an invasive algorithm that guides the solver to prefer one solution over the other. For example, finding a solution that is at the centroid of the polytope of the feasible solutions. This can be an interesting direction for getting a more robust solution.

3. **Exploiting Cardinality Estimators.** We have used the native optimizer of the Postgres engine as our cardinality estimator to get the estimates for the SQL queries for the regions. In the recent years, several promising selectivity estimation algorithms based on machine-learning techniques (e.g., NARU [79], MSCN [42], SDV [60]) have been proposed. These techniques significantly enhance the quality of cardinality estimation. Therefore, we would like to incorporate some of these techniques into the Hydra framework with the hope to further improve the robustness of the synthetic database produced.

4. **Balancing Robustness and Workload Scalability.** To further improve robustness towards volumetric similarity on unseen queries, it is important to handle large training workloads in the input. With the increase in workload constraints, we may run into scalability issues, stemming from the inherent LP complexity. The load on the solver further increases with the addition of objective functions. To sidestep this challenge, a promising recourse is to update our workload decomposition module. Currently the module splits a workload to handle the conflicting constraints. But this can be expanded such that each sub-workload focuses on only a fraction of the data space, while being conflict free. Further, for a test query, we can forward it to the database summary corresponding to the nearest sub-workload. Recall that having multiple summaries does not impose a substantive overhead due to the minuscule size of each summary.

5. **Building Incremental Solutions.** Currently the entire constraint workload is assumed to be given as the input. An alternative scenario is where the constraints are incrementally provided. While, the data-scale-free summary creation permits rebuilding the solution from scratch cheaply, it's an interesting open problem to be able to modify the solution to satisfy additional constraints. Further, this can add in workload-scalability as well.

6. **Expand Scope of Supported SQL Constructs.** Currently Hydra supports filter, inner key-based joins and projection based constructs such as Distinct, Group By and Union. Further, Order By and Aggregate operators are trivially supported from the point of view of volumetric similarity. In the future, we would further like to expand

the scope to include other SQL operations such as Having, non-key joins, outer joins and nested queries.

7. **Exploit Duplication Distribution.** In Chapter 9, we discussed the duplication distribution data characteristic and a basic solution to integrate it with Hydra at the level of base relations. This can be expanded to include each target intermediate operator in query processing. For example, maintaining duplication distributions for the columns participating in hash joins or hash aggregates (used in projections), can help to mimic client environments more closely.

8. **Mimic Presortedness.** Also in Chapter 9, we discussed about the presortedness data characteristic and some initial modeling of it for base table columns. Again, this can be expanded to include intermediate operators in the query plan where it is relevant. For example, maintaining presortedness for the columns participating in sort as part of sort-merge join, or sort based projection, or simply an order by clause, can help to mimic client environments more closely.

9. **Other Characteristics.** Beside duplication distribution and presortedness, other characteristics can also be aimed for mimicking client environments. These can include the aspects of memory, encoding and storage. For example, (a) String lengths and precision for numeric data types play a role in comparison-based operations. Further, they impact how much data can be cached and affect the I/O calls. (b) Differences in value identities can also lead to changes in UDFs computation complexity. It also affects column encoding (for columnar databases), which can lead to difference in the physical size of the data. (c) Data layout on the disk plays a significant role if tables are scanned from the disk, especially in the case of index scans.

Bibliography

[1] American Fuzzy Lop. github.com/google/AFL 33

[2] Big Data. en.wikipedia.org/wiki/Big_data 7, 153

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

[4] General Data Protection Regulation. en.wikipedia.org/wiki/General_Data_Protection_Regulation 22

[5] JOB Benchmark. github.com/gregrahn/join-order-benchmark 8, 65, 97, 130

[6] Pearson Correlation Coefficient. en.wikipedia.org/wiki/Pearson_correlation_coefficient 171

[7] PostgreSQL. postgresql.org/docs/9.6 8, 61, 97, 127, 153, 159, 169

[8] SolarWinds. solarwinds.com/database-performance-monitoring-software 33

[9] SQLancer. github.com/sqlancer/sqlancer 33

[10] SQLSmith. github.com/anse1/sqlsmith 33

[11] TPC-H. tpc.org/tpch/ 1, 67

[12] TPC-DS. tpc.org/tpcds/ 1, 61, 97, 171

[13] USE PLAN SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/performance/apply-a-fixed-query-plan-to-a-plan-guide?view=sql-server-ver16 4

[14] Z3. github.com/Z3Prover/z3 18, 45, 61, 97, 127, 155

[15] D. Agrawal, and C. C. Aggarwal. On the Design and Quantification of Privacy Preserving Data Mining Algorithms. *Proc. of 20th PODS Conf.*, 2001, pgs. 247–255. 22

[16] A. Alexandrov, K. Tzoumas, and V. Markl. Myriad: Scalable and Expressive Data Generation. *PVLDB*, 5(12):1890–1893, 2012. 1, 20

[17] A. Arasu, R. Kaushik, and J. Li. Data Generation using Declarative Constraints. *Proc. of ACM SIGMOD Conf.*, 2011, pgs. 685–696. 2, 3, 4, 19, 23, 29, 46, 47, 97, 107

[18] A. Arasu, R. Kaushik, and J. Li. DataSynth: Generating Synthetic Data using Declarative Constraints. *PVLDB*, 4(12):1418–1421, 2011. 3, 29

[19] Ashoke S., and J. R. Haritsa. CODD: A Dataless Approach to Big Data Testing. *PVLDB*, 8(12):2008–2011, 2015. 9, 16, 143, 155

[20] M. K. Baowaly, C. Lin, C. Liu, and K. Chen. Synthesizing electronic health records using improved generative adversarial networks *JAMIA*, 26(3):228–241, 2019. 22

[21] B. Barak, K. Chaudhuri, C. Dwork, S. Kale, F. McSherry, and K. Talwar Privacy, Accuracy, and Consistency Too: A Holistic Solution to Contingency Table Release *Proc. of 26th PODS Conf.*, 2007, pgs. 273–282. 21

[22] V. Barany, B. Cate, B. Kimelfeld, D. Olteanu and Z. Vagena. Declarative Probabilistic Programming with Datalog. *TODS*, 42(4):1–35, 2017. 20

[23] K. Beedkar, D. Brekardin, J. Quiané-Ruiz, and V. Markl. Compliant Geo-distributed Data Processing in Action. *PVLDB*, 14(12):2843–2846, 2021. 22

[24] C. Binnig, D. Kossmann, and E. Lo. Reverse Query Processing. *Proc. of 23rd ICDE Conf.*, 2007, pgs. 506–515. xvi, 22, 23, 24

[25] 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. xvi, 1, 3, 4, 23, 25

[26] B. Bollobás and A. Thomason. Projections of Bodies and Hereditary Properties of Hypergraphs. *Bulletin of the London Mathematical Society*, 27(5):417–424, 1995. 39

[27] J. Brickell, and V. Shmatikov. The Cost of Privacy: Destruction of Data-Mining Utility in Anonymized Data Publishing. *Proc. of 14th KDD Conf.*, 2008, pgs. 70–78. 22

[28] N. Bruno and S. Chaudhuri. Flexible Database Generators. *Proc. of 31st VLDB Conf.*, 2005, pgs. 1097–1107. 19

[29] T. S. Buda, T. Cerqueus, J. Murphy and M. Kristiansen. ReX: Extrapolating Relational Data in a Representative Way. *Proc. of BICOD Conf.*, 2015, pgs. 95–107. 21

[30] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases*, 4(1-3):1–294, 2012. 22

[31] H. Chen, S. Jajodia, J. Liu, N. Park, V. Sokolov, and V. S. Subrahmanian. FakeTables: Using GANs to Generate Functional Dependency Preserving Tables with Bounded Real Data. *Proc. of 28th IJCAI Conf.*, 2019, pgs. 2074–2080. 22

[32] C. Chen, J. Twycross, J. M. Garibaldi. A new accuracy measure based on bounded relative error for time series forecasting. *PLoS ONE*, 12(3): e0174202, 2017. 16, 137, 148

[33] E. Choi, S. Biswal, B. A. Malin, J. Duke, W. F. Stewart, and J. Sun. Generating Multi-label Discrete Patient Records using Generative Adversarial Networks. *PMLR*, 68:286–305, 2017. 22

[34] J. Domingo-Ferrer. A Survey of Inference Control Methods for Privacy-Preserving Data Mining. *Privacy-Preserving Data Mining*, 2008, pgs. 53–80. 22

[35] C. Dwork. Differential Privacy. *ICALP*, 2006, pgs. 1–12. 22

[36] J. Fan, T. Liu, G. Li, J. Chen, Y. Shen, X. Du. Relational Data Synthesis using Generative Adversarial Networks: A Design Space Exploration. *PVLDB*, 13(11):1962–1975, 2020. 2, 21, 22

[37] A. Gilad, S. Patwa, and A. Machanavajjhala. Synthesizing Linked Data Under Cardinality and Integrity Constraints. *Proc. of ACM SIGMOD Conf.*, 2021, pgs. 619–631. xvi, 2, 3, 8, 22, 23, 31, 32, 97

[38] L. Gondara, and K. Wang MIDA: Multiple Imputation Using Denoising Autoencoders. *Proc. of PAKDD Conf.*, 2018, pgs. 260–272. 22

[39] 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. 19

[40] J. E. Hoag, and C. W. Thompson. A Parallel General-Purpose Synthetic Data Generator. *Proc. of ACM SIGMOD Conf.*, 2007, pgs. 19–24. 1, 19

[41] K. Houkjær, K. Torp, and R. Wind. Simple and Realistic Data Generation *Proc. of 32nd VLDB Conf.*, 2006, pgs. 1243–1246. 19

[42] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *Proc. of CIDR Conf.*, 2019. 178

[43] I. Kotsogiannis, Y. Tao, X. He, M. Fanaeepour, A. Machanavajjhala, M. Hay, and G. Miklau. PrivateSQL: A Differentially Private SQL Query Engine. *PVLDB*, 12(11):1371–1384, 2019. 22

[44] I. Leader, Z. Randelovic, and Eero Raty. Inequalities on Projected Volumes. *arXiv:1909.12858*, 2019. 39, 40, 69

[45] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204—215, 2015. 8, 65, 97, 130

[46] M. Lenzerini, and P. Nobili. On The Satisfiability of Dependency Constraints in Entity-Relationship Schemata. *Proc. of 13th VLDB Conf.*, 1987, pgs. 147–154. 39

[47] N. Li, T. Li, and S. Venkatasubramanian. t-Closeness: Privacy Beyond k-Anonymity and ℓ-Diversity. *Proc. of 23rd ICDE Conf.*, 2007, pgs. 106-115. 22

[48] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. ℓ-Diversity: Privacy Beyond k-Anonymity. *TKDD*, 1(1):3-es, 2007. 22

[49] R. McKenna, G. Miklau, M. Hay, and A. Machanavajjhala. Optimizing Error of High-Dimensional Statistical Queries under Differential Privacy. *PVLDB*, 11(10):1206–1219, 2018. 22

[50] C. Li, G. Miklau, M. Hay, A. McGregor, and V. Rastogi. The Matrix Mechanism: Optimizing Linear Counting Queries Under Differential Privacy. *The VLDB Journal*, 24(6):757–781, 2015. 22

[51] H. Li, L. Xiong, L. Zhang, and X. Jiang. DPSynthesizer: Differentially Private Data Synthesizer for Privacy Preserving Data Sharing. *PVLDB*, 7(13):1677–1680, 2014. 21

[52] Y. Li, R. Zhang, X. Yang, Z. Zhang, and A. Zhou. Touchstone: Generating Enormous Query-Aware Test Databases. *USENIX ATC*, 2018, pgs. 575–586. xvi, 2, 3, 23, 28

[53] E. Lo, C. Binnig, D. Kossmann. A framework for testing DBMS features. *The VLDB Journal*, 19(2):203–230, 2010. 19, 23, 25

[54] E. Lo, N. Cheng, W.-K. Hon. Generating Databases for Query Workloads. *PVLDB*, 3(1):848–859, 2010. 26

[55] 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. xvi, 2, 3, 23, 26, 27

[56] P. Lu, P. Wang, and C. Yu. Empirical Evaluation on Synthetic Data Generation with Generative Adversarial Network. *Proc. of 9th WIMS Conf.*, 2019, pgs. 16:1–6. 22

[57] G. S. Manku, and R. Motwani. Approximate Frequency Counts over Data Streams. *Proc. of 28th VLDB Conf.*, 2002, pgs. 346–357. 169

[58] Y. Park, and J. Ghosh. PeGS: Perturbed Gibbs Samplers that Generate Privacy-Compliant Synthetic Data. *Trans. Data Priv.*, 7(3):253–282, 2014. 21

[59] N. Park, M. Mohammadi, K. Gorde, S. Jajodia, H. Park, and Y. Kim. Data Synthesis based on Generative Adversarial Networks *PVLDB*, 11(10): 1071–1083, 2018. 22

[60] N. Patki, R. Wedge, and K. Veeramachaneni. In DSAA, pages 399–410, 2016. T. Rabl, M. Danisch, M. Frank, S. Schindler, and H. Jacobsen. The Synthetic Data Vault. *Proc. of IEEE DSAA Conf.*, 2016, pgs. 399–410. 21, 178

[61] 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. 2, 19, 21

[62] T. Rabl, M. Frank, H. M. Sergieh and H. Kosch. A Data Generator for Cloud-Scale Benchmarking. *Proc. of 2nd TPCTC Conf.*, 2010, pgs. 41–56. 1, 20

[63] M. Rigger, and Z. Su. Testing Database Engines via Pivoted Query Synthesis. *USENIX OSDI*, 2020, pgs. 667–682. 33

[64] M. Rigger, and Z. Su. Detecting Optimization Bugs in Database Engines via Non-optimizing Reference Engine Construction. *Proc. of the 28th ACM ESEC/FSE Conf.*, 2020, pgs. 1140–1152. 34

[65] M. Rigger, and Z. Su. Finding Bugs in Database Systems via Query Partitioning. *Proc. of the ACM on Prog. Lang.*, 4(OOPSLA):1–30, 2020. 34

[66] E. Shen, and L. Antova. Reversing statistics for scalable test databases generation. *Proc. of DBTest Workshop*, 2013, pgs. 1–6. 2, 19, 21

[67] A. Silberschatz, H. F. Korth, and S. Sudarshan. Database System Concepts, McGraw-Hill, New York, Seventh Edition, 2020. 68

[68] J. M. Stephens and M. Poess. MUDD: A Multi-dimensional Data Generator. *Proc. of 4th WOSP*, 2004, pgs. 104–109. 1, 19

[69] L. Sweeney. k-Anonymity: A Model for Protecting Privacy. *Intl. Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002. 22

[70] 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. 39

[71] Y. C. Tay, B. T. Dai, D. T. Wang, E. Y. Sun, Yong Lin and Yuting Lin. UpSizeR: Synthetically scaling an empirical relational database. *Inf. Syst.*, 38(8):1168–1183, 2013. 21

[72] S. Thirumuruganathan, S. Hasan, N. Koudas, and G. Das. Approximate Query Processing using Deep Generative Models. *arXiv:1903.10000*, 2019. 22

[73] R. S. Trivedi, I. Nilavalagan, and J. R. Haritsa. CODD: COnstructing Dataless Databases. *Proc. of DBTest Workshop*, 2012, pgs. 1–6. 4, 153

[74] F. Waas, and C. Galindo-Legaria. Counting, Enumerating, and Sampling of Execution Plans in a Cost-Based Query Optimizer. *Proc. of ACM SIGMOD Conf.*, 2000, pgs. 499–509. 33

[75] Q. Wang, Y. Li, R. Zhang, K. Shu, Z. Zhang, and A. Zhou. A Scalable Query-Aware Enormous Database Generator for Database Evaluation. *TKDE*, 10.1109/TKDE.2022.3153651, 2022. 28

[76] W. E. Winkler. Masking and Re-identification Methods for Public-use Microdata: Overview and Research Problems. *Privacy in Statistical Databases*, 2004, pgs. 231–246. 22

[77] X. Xiao, G. Wang, and J. Gehrke. Differential Privacy via Wavelet Transforms. *TKDE*, 23(8):1200–1214, 2011. 22

[78] L. Xu, M. Skoularidou, A. Cuesta-Infante, and K. Veeramachaneni. Modeling Tabular Data using Conditional GAN. *Proc. of 33rd NeurIPS Conf.*, 2019. 22

[79] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep Unsupervised Selectivity Estimation. *PVLDB*, 13(3):279–292, 2019. 178

[80] J. Yang, P. Wu, G. Cong, T. Zhang, X. He. SAM: Database Generation from Query Workloads with Supervised Autoregressive Models. *Proc. of ACM SIGMOD Conf.*, 2022, pgs. 1542–1555. xvi, 3, 23, 32, 33

[81] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, and X. Xiao. PrivBayes: Private Data Release via Bayesian Networks. *TODS*, 42(4):1–41, 2017. 21

[82] J. W. Zhang and Y. C. Tay. Dscaler: Synthetically Scaling A Given Relational Database. *PVLDB*, 9(14):1671–1682, 2016. 2, 21