Enhancing Coverage and Robustness of Database Generators

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Technology (Research)

IN THE

Faculty of Engineering

BY

Rajkumar S

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

November, 2021

Declaration of Originality

I, **Rajkumar S**, with SR No. **04-04-00-10-22-18-1-15761** hereby declare that the material presented in the thesis titled

Enhancing Coverage and Robustness of Database Generators

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2018 - 2021**.
With my signature, I certify that:

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

Date: 29/11/2021 Student Signature

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

Advisor Name: Prof. Jayant R. Haritsa Advisor Signature

DEDICATED TO

My Parents

Acknowledgements

I would like to express my gratitude to my research supervisor, Prof. Jayant R. Haritsa, for giving me the opportunity to work on this project. I am thankful to him for his valuable guidance and continuous support at every step.

I am equally thankful to Anupam Sanghi for mentoring me throughout my research journey, without whom this thesis would not have reached its present state. I would also like to thank my other Database System Laboratory inmates for their valuable suggestions and continuous encouragement.

Next, I would like to thank Indian Institute of Science and the Department of Computer Science and Automation for providing me the opportunity to pursue my master's degree in this world-class institution. I would also like to express my gratitude to all the office staff of both the departments of CSA and CDS.

Finally, I would like to thank both my parents, who have always loved me unconditionally and supported me in each and every phase of my life. Also, I want to thank all my friends, both from and outside of IISc for always being there by my side.

Abstract

Generating synthetic databases that capture essential data characteristics of client databases is a common requirement for enterprise database vendors. This need stems from a variety of use-cases, such as application testing and assessing performance impacts of planned engine upgrades. A rich body of literature exists in this area, spanning from the early techniques that simply generated data ab-initio to the contemporary ones that use a predefined client query workload to guide the data generation. In the latter category, the aim specifically is to ensure volumetric similarity – 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 similar in the original and synthetic databases.

Hydra is a recently proposed data regeneration framework that provides volumetric similarity. In addition, it also provides a mechanism to generate data dynamically during query execution, using a minuscule database summary. Notwithstanding its desirable characteristics, Hydra has the following critical limitations: (a) limited scope of SQL operators in the input query workload, (b) poor scalability with respect to the number of queries in the input workload, and (c) poor volumetric similarity on unseen queries. The data generation algorithm internally uses a linear programming (LP) solver that throttles the workload scalability. This not only puts a threshold on the training (seen) workload size but also reduces the accuracy for test (unseen) queries. Robustness towards test queries is further adversely affected by design choices such as a lack of preference among candidate synthetic databases, and artificial skew in the generated data.

In this work, we present an enhanced version of Hydra, called High-Fidelity Hydra (HF-Hydra), which attempts to address the above limitations. To start with, we expand the SQL operator coverage to also include the LIKE operator, and, in certain restricted settings, projection-based operators such as GROUP BY and DISTINCT. To sidestep the challenge of workload scalability, HF-Hydra outputs not one, but a suite of database summaries such that they collectively cover the entire input workload. The division of the workload into the associated sub-workloads is governed by heuristics that aim to balance robustness with LP solvability.

Abstract

For generating richer database summaries, HF-Hydra additionally exploits metadata statistics maintained by the database engine. Further, the database query optimizer is leveraged to make the choice among the various candidate databases. The data generation is also augmented to provide greater diversity in the represented values. Finally, when a test query is fired, HF-Hydra directs it to the database summary that is expected to provide the highest volumetric similarity.

We have experimentally evaluated HF-Hydra on a customized set of queries based on the TPC-DS decision-support benchmark framework. We first evaluated the specialized case where each training query has its own summary, and here HF-Hydra achieves perfect volumetric similarity. Further, each summary construction took just under a second and the summary sizes were just in the order of a few tens of kilobytes. Also, our dynamic generation technique produced gigabytes of data in just a few seconds. For the general setting of a limited set of summaries representing the training query workload, the data generated by HF-Hydra was compared with that from Hydra. We observed that HF-Hydra delivers more than forty percent better accuracy for outputs from filter nodes in the plans, while also achieving an improvement of about twenty percent with regard to join nodes. Further, the degradation in volumetric similarity is minor as compared to the one-summary scenario, while the summary production is significantly more efficient due to reduced overheads on the LP solver.

In summary, HF-Hydra takes a substantive step forward with regard to creating expressive, robust, and scalable data regeneration frameworks with immediate relevance to testing deployments.

Publications based on this Thesis

1. Anupam Sanghi, **Rajkumar Santhanam** and Jayant R. Haritsa
   Towards Generating HiFi Databases (short paper),
   In *Proc. of 26th International Conference on Database Systems for Advanced Applications (DASFAA), Taipei, Taiwan, April 2021*, pages 105-112, 2021.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

A rich body of literature exists on data regeneration, beginning with *workload-independent* techniques (e.g [16, 12]), which provide scalable and efficient solutions, but fail to retain complex statistical characteristics such as the sizes of intermediate relations created during the execution of a query plan. To address this problem, a particularly potent approach of *workload-dependent* database regeneration was introduced in QAGen [11], and has served as the foundation for many of the practicable systems proposed over the last decade [20, 7]. Workload-dependent techniques aim to generate synthetic data whose behavior is *volumetrically similar* to the client database on the input query workload. That is, assuming a common choice of query execution plans at the client and vendor sites (ensured through "plan forcing" [3] or "metadata matching" [24]), the output row cardinalities of individual operators in these plans are very similar in the original and synthetic databases. Our goal is to reduce the absolute error between the output row cardinalities of each operator in the plans obtained from the client as well as the regenerated database. This similarity helps to preserve the multi-dimensional layout and flow of the data, a

pre-requisite for achieving similar performance on the client's workload. As a case in point, the DataSynth [7, 8] tool from Microsoft expresses such volumetric constraints as a Linear Program (LP) whose solution is used to construct the synthetic database.

A common limitation of contemporary techniques (reviewed in detail in Chapter 2), is that they run into issues of *scale* or *efficiency* at one stage or the other in the regeneration pipeline. This is partly due to their focus on *materialized* static solutions, making them impractical at large volumes. Further, the ability to scale to large query workloads and data volumes has not been clearly established, and validations have been typically restricted to relatively simple and small benchmarks such as TPC-H [4].

Also, workload-aware data regeneration focuses on volumetric similarity for the client's input (training) query workload. But in database testing applications, generalization to new queries can be a necessary requirement at the vendor site. Hence, reasonable volumetric similarity on any unseen (testing) query workload is critical as well. This ability to provide *robustness* to test queries has not been explicitly addressed in generators proposed in the literature.

1.1 Prior Work

To materially address the above-mentioned challenges in the data regeneration pipeline, a potent database regenerator, **Hydra** ([25], [26]) was recently proposed. Hydra is based on a workload-aware declarative approach to database regeneration, and aims to create synthetic databases that achieve *volumetric similarity* on input query workloads. Specifically, it takes as input from the client, the metadata and query execution plans corresponding to a query workload. With this input, it generates a synthetic database at the vendor site such that a similar volume of data is processed at each stage of the query execution pipeline for the workload queries. Hydra consciously addresses data scale and efficiency issues through the entire regeneration pipeline. A special feature of Hydra is that the data generation can be done "on-demand", obviating the need for materialization. This functionality is achieved through the construction of a concise *database summary*.

In this section, we provide background information on the basic concepts – Annotated Query Plans [11] and Cardinality Constraints [7] – underlying workload-dependent regeneration that are relevant to our design of the Hydra system.

1.1.1 Annotated Query Plans

Consider a toy scenario (for ease of presentation) where the client has the following database schema:

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

where pk and fk refer to primary-key and foreign-key attributes, respectively. A sample client query on this schema is shown in Figure 1.1, with the corresponding query execution plan in Figure 1.2. Note that this execution plan has the output edge of each operator annotated with the associated row cardinality (as evaluated during the client's execution) – for instance, there are 50K rows resulting from the join of R and S. Such a plan is referred to as an "Annotated Query Plan" (AQP) in [11]. The goal now is to generate synthetic data at the vendor site such that when the above query is executed on this data, we obtain the identical, or very similar, AQP.

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

Figure 1.1: Example Query

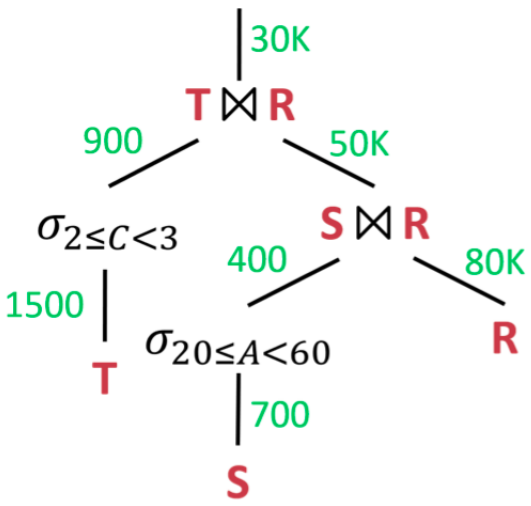


Figure 1.2: Example Annotated Query Plan

1.1.2 Cardinality Constraints

A unified and declarative mechanism for representing AQP data characteristics, called *cardinality constraints* (CCs), was proposed in [7]. A cardinality constraint dictates that the output of

a given relational expression over the generated database should feature the specified number of rows. For instance, the CCs expressing the AQP of Figure 1.2 are shown in Figure 1.3. The data regeneration technique takes the schematic information and the set of CCs from the client site and produces synthetic data that closely meets these CCs. To make the problem tractable, it is assumed that CCs consist of filters on only non-key attributes and that all joins are between primary keys and foreign keys.

$|R| = 80K$
$|S| = 700$
$|T| = 1500$
$|\sigma_{S.A \in [20,60)}(S)| = 400$
$|\sigma_{T.C \in [2,3)}(T)| = 900$
$|\sigma_{S.A \in [20,60)}(R \bowtie S)| = 50K$
$|\sigma_{S.A \in [20,60) \wedge T.C \in [2,3)}(R \bowtie S \bowtie T)| = 30K$

Figure 1.3: Cardinality Constraints (CCs)

1.1.3 Overview of Hydra

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

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

1. Satisfiability Modulo Theories (SMT) problem formulation
2. Database Generation

Let us look into each of the steps in the upcoming subsections. Prior to that, we present a brief introduction to SMT for better understanding.

1.1.3.1 Satisfiability Modulo Theories (SMT)

In computer science and mathematical logic, Satisfiability Modulo Theories (SMT) is the problem of determining whether a mathematical formula is satisfiable. It generalizes the Boolean satisfiability problem (SAT) to more complex formulas involving real numbers, integers, and/or various data structures such as lists, arrays, bit vectors, and strings. The name is derived from the fact that these expressions are interpreted within ("modulo") a certain formal theory in first-order logic with equality (often disallowing quantifiers). Formally speaking, an SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable. SMT solvers (e.g. Z3 [5, 15] and CVC4 [9]), which aim to solve the SMT problem for a practical subset of inputs, have been used as a building block for a wide range of tools across computer science, including in automated theorem proving, program analysis, program verification, and software testing.

1.1.3.2 SMT Problem Formulation in Hydra

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

For the view S an example constraint is shown by the red box (A in [20, 40) and B in [15000, 50000)) in Figure 1.4 has an associated cardinality 150 (say). Likewise, another green constraint is also shown - say with cardinality 250. Let the total number of tuples in the S be 700. These constraints can be expressed as the following linear constraints:

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

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

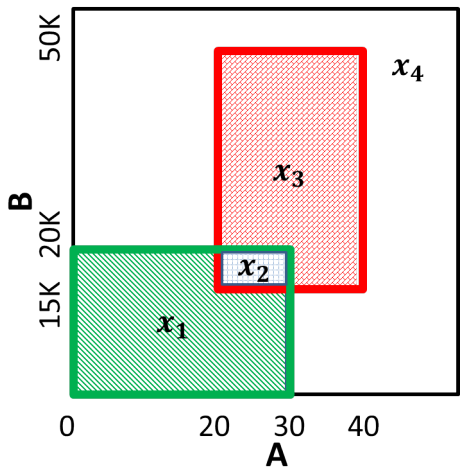


Figure 1.4: Region Partitioning in Hydra

1.1.3.3 Database Generation

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

An example database summary is shown in Figure 1.5. Here, entries of the type a - b in the PK columns (e.g. 101-250 for S_pk in table S), mean that the relation has $b - a + 1$ tuples with values $(a, a + 1, \ldots, b)$ for that column, keeping the other columns unchanged. Using the summary, *Tuple Generation module* (implemented inside the DBMS query executor module) dynamically generates tuples on demand during query execution.

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

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

T	
T_pk	C
1-902	2
903-1505	0

Figure 1.5: Example Database Summary in Hydra

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

1.2 Limitations of Hydra

While Hydra has the above-mentioned desirable characteristics, its applicability is restricted to handling volumetric similarity on *training* queries. Following are some of its key limitations:

Limited SQL Operator Coverage: Hydra supports input training queries that feature filters and joins. Among filters, Hydra is restricted to only supporting equality and range predicates. Queries that feature pattern matching operators such as LIKE, and projection based operators such as DISTINCT and GROUP BY are not supported by Hydra.

Inherent Scalability Limitation: Hydra can only handle a few hundred queries in its client workload owing to its *inherent scalability limitation*. The limitation is caused by the use of Z3 solver to solve the equations from the SMT formulation. For larger query workloads, with an increase in the number of variables and the complexity of the equations, the solver ends up crashing.

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

Restricted Diversity in Generated Data: Within a region that gets a non-zero cardinality assignment, Hydra picks up a single data point, and assigns it with the entire region's cardinality. Hence, the generated data ends up with featuring only very few unique data points, thus lacking diversity. Hence, accuracy to test queries is greatly affected.

1.3 Our Contributions

In this thesis, we present **High-Fidelity Hydra (HF-Hydra)**, an enhancement over Hydra, to address the above-mentioned limitations. The overview of each segment of our contributions are as follows:

Operator Coverage Expansion

In realistic client workloads, queries involving string attributes feature pattern matching filters, i.e., they involve the LIKE operator. Hence, in the first segment of the thesis, we discuss extending the ambit of Hydra's operator coverage which is restricted to only equality and range predicates in filter constraints, to also include the LIKE operator. LIKE predicates can be considered as regular expression strings, and for a workload of queries, there may be multiple predicates applied on the same attribute with possible intersections. In HF-Hydra, we propose a solution to discover and handle such intersections. We then transform the LIKE predicates into equivalent equality filter predicates, which can be eventually processed by the existing setup.

Moving on, to sidestep the challenge of workload scalability, HF-Hydra outputs not one, but a suite of database summaries, each constructed from a sub-workload such that they collectively cover the entire input workload. The division of the workload into the associated sub-workloads is governed by heuristics aiming to provide robustness to test queries while ensuring computational viability. A pictorial representation of this idea is illustrated in Figure 1.6. Let us consider the training workload to contain n queries. Firstly, we consider a special case, where the scope is restricted to providing volumetric similarity to only training queries. Here, we generate individual summaries for each query in the workload. Hence the total number of summaries m, would be equal to n. Although such a solution helps in bypassing the scalability limitation, this may not extend robustness to test queries. And so, secondly, we consider the general case, where the goal is to also provide good robustness to test queries. An ideal case here is when all the training queries are used to construct one summary ($m = 1$). This summary can be expected to provide good robustness, but for larger workloads, the inherent limitation of the solver prevents us from the queries at once. Hence, to meld robustness with scalability, a workload-division strategy is proposed to divide the workload into a few sub-workloads, and a summary is produced for each of them ($m < n$). At runtime, when a test query is fired, HF-Hydra directs it to the database summary that is expected to provide the highest volumetric similarity. The details of each of these contributions are discussed in the sequel.

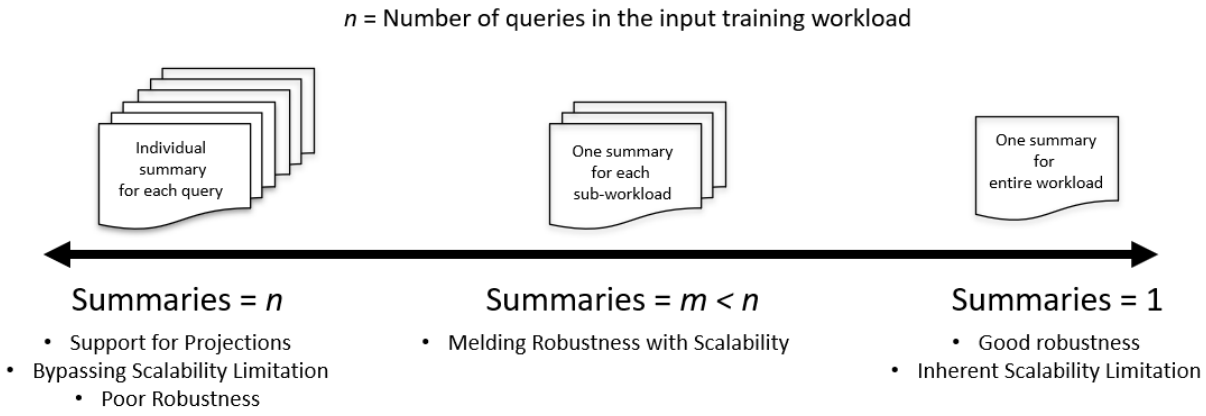


Figure 1.6: Overview of suite of summaries proposed in HF-Hydra

Managing Scalability for Training Workloads

In this next segment of the thesis, we look into our special case that focuses only on handling training queries. Here, HF-Hydra provides an extreme solution wherein, for each query in the workload, a different database summary is produced. The idea here is that with dynamic generation in action, when a training query is fired, the summary for that query is picked to generate tuples on the fly. With this approach, we can bypass the inherent scalability limitation and manage to handle any number of training queries. Also, in this specialized setting, we extend support for projection-based SQL operators.

Robustness to Testing Queries

In this subsequent segment of the thesis, we discuss the framework of HF-Hydra that addresses the general case, where the focus is also to provide robustness for test queries. To start with, in addition to the constraints from the training workload, we model the metadata statistics as constraints as well, and augment them to obtain a *refined* region-partitioning of the data space. Metadata statistics are usually not maintained up to date, and so these constraints could be inconsistent with the workload constraints. We frame an LP with the workload constraints, and the metadata constraints are included as an optimization function that tries to satisfy them with minimal error. From the LP solution, we correct the metadata constraints with their errors to resolve any inconsistency.

As the next step, an equivalent SQL query is generated for every region, and its cardinality estimate is obtained from the database query optimizer. Another LP is framed, with both workload and *error-corrected* metadata constraints, and an optimization function is added that

tries to get the region's cardinality to be assigned by the solver close to the region's estimate. This way, the optimizer is leveraged to make a focused choice among the candidate databases. Finally, a summary is constructed, and for each region therein with a non-zero cardinality assignment, we generate tuples adhering to the optimizer's distribution model, with greater diversity in the represented domain values.

Melding Robustness with Scalability

In the framework discussed for the general setting in the previous segment, with an increase in the workload constraints, and use of optimization functions, the load on the theorem prover is even more severe, and we may run into scalability issues. Hence, to sidestep this challenge, in the last segment of the thesis, we propose a *workload-division* strategy to produce a few sub-workloads, and a summary is produced for each sub-workload. The process starts with introducing two sub-workloads, by splitting the training domain interval for each attribute across the whole workload into two sub-domains, and associating each with either sub-workload. A workload constraint that has an intersection with a sub-domain in any of its participating attributes is assigned to the corresponding sub-workload. Each sub-workload is then checked for computational viability in the LP formulation process. The heuristic used for this check is whether the number of variables that are produced from the constraints in a sub-workload is under the limit that the solver can easily handle. For sub-workloads that fail the check, the splitting is repeated recursively for each one of them.

When a test query is fired, each of its participating attributes is checked for intersection with each sub-domain corresponding to every sub-workload. We then deterministically direct the test query to be executed on the data generated from the sub-workload where the intersection is found to be maximum.

We have experimentally evaluated HF-Hydra on a customized set of queries based on the TPC-DS decision-support benchmark. We start with the specialized case where each training query has its own summary, and show that HF-Hydra achieves perfect volumetric similarity in this setting. Further, the summaries are constructed in the order of seconds and the summaries occupy only a few tens of kilobytes. The dynamic generation technique produces data in gigabytes per second, comparable to data access speeds from materialized tables on permanent storage. For the general setting of a limited set of summaries representing the training query workload, the data generated by HF-Hydra was compared with that obtained from Hydra. The evaluation was done in two parts: First, we constructed a workload for which a single summary could be produced with viable LP complexity. In this case, HF-Hydra delivered **44 percent** better accuracy for filter nodes, while also achieving an improvement of **27 percent**

with regard to join nodes. Secondly, to test our workload-division strategy, we extended the workload to include a larger suite of queries, giving rise to a few sub-workloads. In this case, HF-Hydra delivered close to **30 percent** better accuracy for filter nodes, while also achieving an improvement of close to **20 percent** with regard to join nodes. Also, to evaluate our logic of directing a given test query to the appropriate summary, we showcase both the best and worst case possible assignment, and it was observed that the former outperformed the latter in terms of accuracy.

1.4 Thesis Organisation

The remainder of this thesis is organized as follows: We start with reviewing the related literature in Chapter 2. Moving to our contributions, firstly, we present the operator coverage expansion done in HF-Hydra to include the LIKE operator in Chapter 3. Subsequently, we present in detail our simplified approach to manage scalability for training queries in Chapter 4. Later, we present a detailed description of the design and implementation of the general framework of HF-Hydra to provide robustness to testing query workloads in Chapter 5. Lastly, we discuss our framework for melding robustness with scalability in Chapter 6. Finally, we dive into a detailed experimental evaluation of HF-Hydra for each scenario in chapter 7. We conclude the thesis and discuss the future directions in Chapter 8.

Chapter 2

Related Work

Over the last 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. [16, 12]), 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. [23, 28]). The other uses both the original database and the query workload to achieve statistical fidelity during evaluation (e.g [11, 7]). Both Hydra and HF-Hydra fall into this class.

In this chapter, we briefly review recent literature on the above research categories.

2.1 Ab Initio Generation

Descriptive languages for the definitions of data dependencies and column distributions were proposed in [12, 17, 22]. For example, [12] proposed a special-purpose language called Data Generation Language (DGL) that is used by the tool to generate synthetic data distributions by utilizing the concept of iterators. It supports a broad range of dependencies between relations but the construction of dependent tables always requires access to the referenced table, creating a bottleneck on the data generation speed.

In contrast to the above, MUDD[29] and PSDG[17] generate all related data at the same time. However, this can also be rendered inefficient if the referenced tables are large in size. MUDD proposes algorithms to parallelize the data generation process, and to efficiently generate dense-unique-pseudo-random sequences and derive nonuniform distributions. Both MUDD and PSDG decouple data generation details from data description, facilitating users to customize the tool for their needs.

In the distributed setting, a faster way of generating references is through recomputing since it eliminates the I/O costs incurred to satisfy referential constraints across relations that are present across different nodes. PDGF [22] was designed with this goal of achieving scalability and decoupling. In PDGF, the user specifies two XML configuration files, one for the data model and one for the formatting instructions. The generation strategy is based on the exploitation of determinism in pseudo-random number generators (PRNG) that allows regenerating the same sequences, hence eliminating the scan overheads. PDGF supports the generation of data with cyclic dependencies as well, but this incurs high computation cost for generating the keys. Finally, PDGF comes with a set of fixed generators for different datatypes and basic distribution functions.

A similar generator is Myriad [6], 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 nodes and ensures that they can run independently from each other, without imposing any restrictions on the data modeling language.

2.2 Database Dependent Regeneration

DBSynth[23] is an extension to PDGF, which 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, Markov chain generators are used if the text data contains multiple words. These help in analyzing the word combination frequencies and probabilities. Finally, after the model construction is complete, PDGF is invoked to generate the corresponding data.

Like DBSynth, RSGen[28] takes a metadata dump, including 1-D histograms, as the input, and generates database tables along with a loading script as the output. RSGen uses a bucket-based model at its core, which is able to generate trillions of records with minimum memory footage. However, the proposed technique works well only for queries with single attribute range predicates. Further, due to the inaccurate cost computations and statistical model in the query optimizers, the volumetric similarity is poor for queries involving predicates on correlated attributes.

UpSizeR [30] is a graph-based tool that uses attribute correlations extracted from an existing database to generate an equivalent synthetic database. A derivative work, Rex [13] produces an extrapolated database given an integer scaling factor and the original database, while maintaining referential constraints and the distributions between the consecutive linked tables. Dscaler [32] addresses the problem of generating a non-uniformly5 scaled version of a database using fine-grained, per-tuple correlations for key attributes, but such information is

typically hard to come by. Moreover, all these techniques only generate the key attributes, whereas the non-key values are sampled from the original database using these key values. Hence, the approach becomes impractical in Big Data and security-conscious environments. Finally, Dscaler fails to retain accuracy for some common query classes.

2.3 Workload Aware Regeneration

Apart from the above techniques, another line of work [10, 11, 20, 7] is based on workload dependence (as in the case of Hydra). Here the aim is to generate a database given a workload of queries such that volumetric similarity is achieved on these queries.

2.3.1 Reverse Query Processing (RQP)

Reverse Query Processing (RQP) [10] gets a query and a result as input and returns a possible database instance that could have produced that result for that query. Specifically, 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. In general, there are many different database instances that can be generated for a given Q and R. Depending on the application, some of these instances might be better than others. In order to generate test databases, it might be advantageous to generate a small D so that the running time of tests are reduced. While RQP tries to be minimal, the techniques do not guarantee any minimality.

The overall architecture of RQP is shown in Figure 2.1. In RQP, a query is (reverse) processed in four steps by the following components: (a) Parser: The SQL query is parsed into a query tree which consists of operators of the relational algebra. 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. In fact, 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. (b) 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^+ . That way, it can be guaranteed that a top-level operator of the reverse query tree does not generate any data that violates one of the database integrity constraints. (c) Query Optimization: In the last

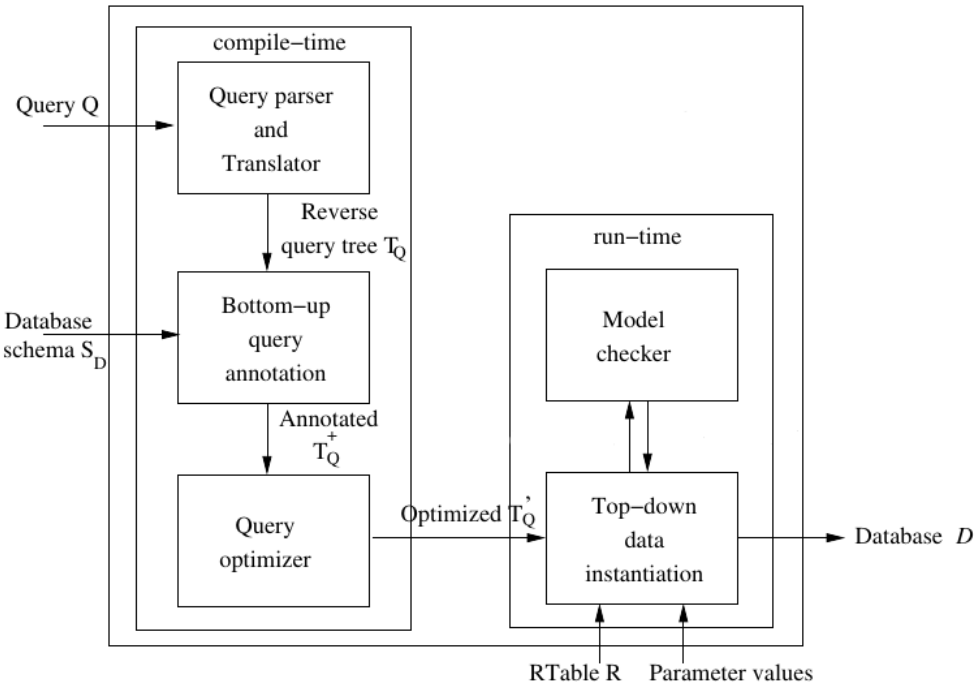


Figure 2.1: Architecture of RQP

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. (d) Top-down Data Instantiation: At run-time, the annotated reverse query tree is interpreted using the RTable R as input. Just as in traditional query processing, there is a physical implementation 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 (more precisely, the decision procedure of a model checker) is used in order to generate data.

In many applications, queries have parameters (e.g., bound by a host variable). In order to process such queries, values for the query parameters must be provided as input to Top-down data instantiation phase. The choice of query parameters again depends on the application; for test database generation, for instance, it is possible to generate several test databases with different parameter settings derived from the program code. In this case, the first three phases of query processing need only be carried out once, and the Top-down data instantiation can use the same optimized annotated reverse query tree for each set of parameter settings. It is also possible to use variables in the RTable. That way, it is possible to specify tolerances. Specifying such tolerances has two important advantages. First, depending on the SQL query, it might not be possible to find a test database that generates a report with the exact value.

Second, specifying tolerances (if that is acceptable for the application) can significantly speed up reverse query processing because it gives the model checker more options to find solutions.

The equivalence of two SQL queries is undecidable if the queries include the relational minus operator and if the queries do not follow a distinct operator order. As a result, RQP for SQL is also undecidable; that is, in general it is not possible to decide whether a D exists for a given R and Q if Q contains a minus operator. Furthermore, there are obvious cases where no D exists for a given R and Q (e.g., if tuples in R violate basic integrity constraints). The approach presented in this paper, therefore, cannot be complete. It is a best-effort approach: it will either fail (return an error because it could not find a D) or return a valid D.

2.3.2 Query Aware Generation (QAGen)

The idea of using cardinalities from a query plan tree was first introduced in Query Aware Generation (QAGen) [11]. They start by constructing a *symbolic database* which is like a regular database, but its attribute values are symbols (variables), not constants, and then translate the input AQPs to constraints over the symbols in the database.

The data generation process of QAGen consists of two phases: (a) the symbolic query processing phase, and (b) the data instantiation phase. The goal of the symbolic query processing phase is to capture the user-defined constraints on the query into the target database. To process a query without concrete data, 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 instantiates a database that contains a set of symbols instead of concrete data. Inside each symbolic tuple, the values are represented by symbols rather than by concrete values. Since the symbolic database is a generalization of relational databases and provides an abstract representation for concrete data, this gives room to QAGen to control the output of each operator of the query. The symbolic query processing phase leverages the concept of traditional query processing.

The general architecture of QAGen is shown in Figure 2.2. First, the database schema M and the input query Q_P are analyzed 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. Then, the user specifies her desired requirements on the operators of the query tree. Afterwards, the input

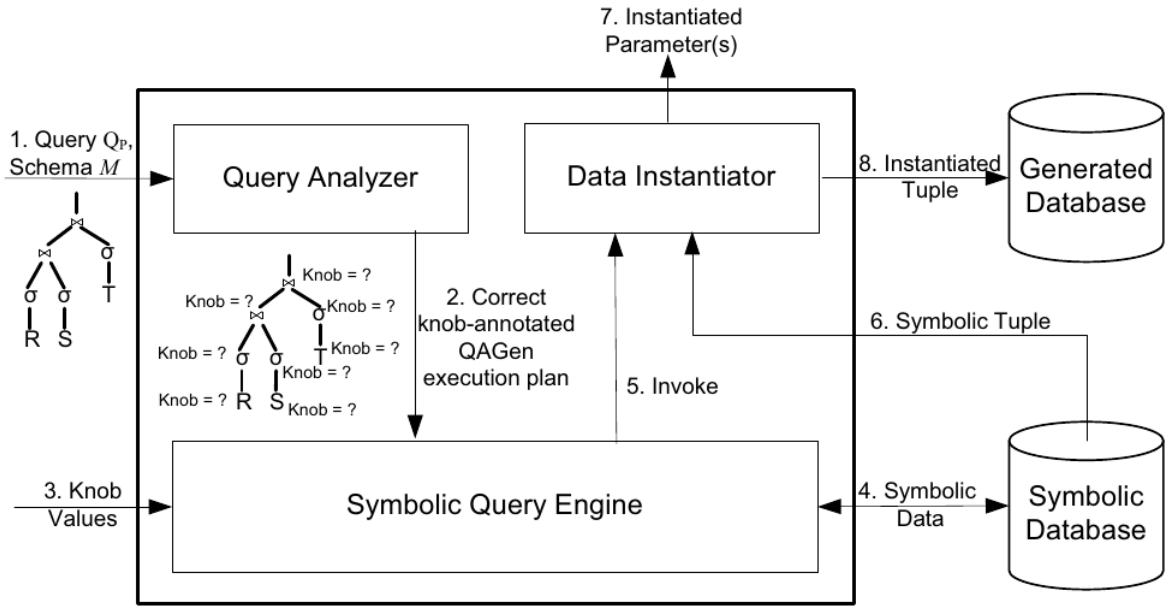


Figure 2.2: Architecture of QAGen

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.

However, unlike in traditional query processing, the symbolic execution of operators deals with symbolic data rather than concrete data. 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). The *Data Instantiation* phase follows the symbolic query processing phase. This phase reads in tuples from the symbolic database that are prepared by the symbolic query processing 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. To allow a user to define different test cases for the same query, the input query of QAGen is in the form of a relational algebra expression.

On the positive side, these generators are capable of handling complex operators as they use a general CSP, but the performance cost is huge since the number of CSP calls increases with the database size. Further, it requires operating on a symbolic database of matching size to the original database, and processing the entire database during the algorithm execution.

This makes it impractical for Big Data environments.

QAGen is similar to HF-Hydra's specialized setting discussed in Chapter 4, in the sense that both of them work on the level of individual input queries, and creates an exclusive database for that query.

2.3.3 MyBenchmark

MyBenchmark [20], an offline data generation tool, uses the Symbolic Query Processing (SQP) technique developed in [11] as a building block. However, MyBenchmark does not restrict itself to find a single database instance D for all input queries. Instead, 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 (a) 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 in which each query has to be executed on a separate generated database. Therefore, the goal of MyBenchmark is to minimize m, the number of generated databases, in best effort. Figure 2.3 shows the architecture for MyBenchmark.

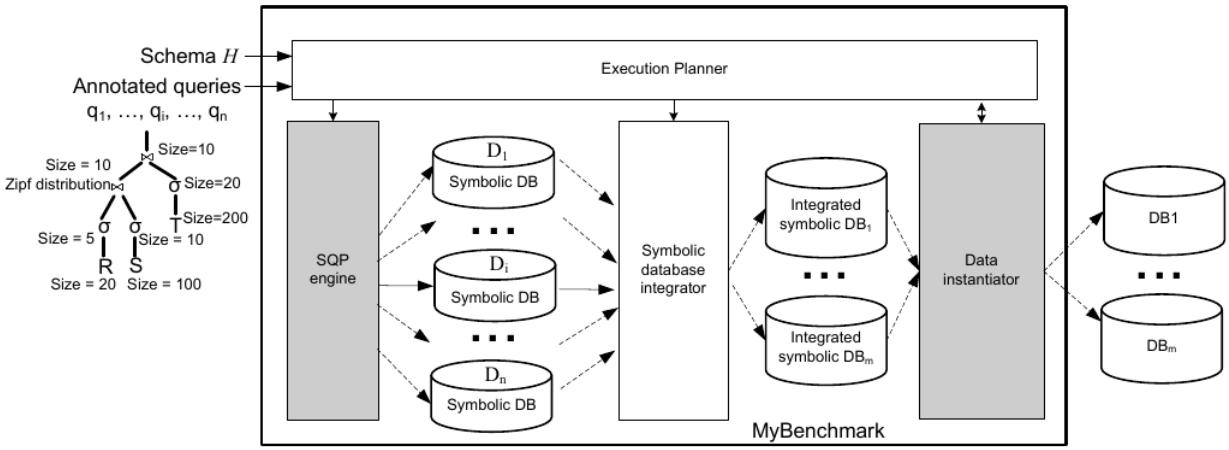


Figure 2.3: Architecture of MyBenchmark

SQP was designed to generate n separate databases for the n input annotated queries. If SQP is carried out on a "processed" symbolic database, SQP will generate many symbolic tuples with contradicting constraints (as different queries may impose different constraints on the same

symbolic tuple) and they will be unable to be instantiated with concrete values. 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 (SDBs) $D_1, D_2, \ldots, D_n$. 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 the SDBs. The integration algorithms are designed to minimize the number of symbolic tuples with contradicting constraints and the number of generated databases. Finally, we use the *Data Instantiator* of QAGen to instantiate each integrated SDB with concrete values. The major advantage of this architecture is that we can fully utilize the capability of SQP in processing a variety of SQL queries.

QAGen is similar to HF-Hydra's extended version of the general framework discussed in Chapter 6, in the sense that both of them take in an input workload of n queries, and produces a few (m) databases, and the number of databases m is tried to be kept minimum in best effort.

2.3.4 DataSynth

The goal of Datasynth [7] is to design efficient algorithms for generating synthetic databases that satisfy a given set of cardinality constraints. The set of constraints provided as input can be large. The queries in the constraints can be complex, possibly involving joins over multiple tables. DataSynth assumes that the input database schema is a snowflake schema with only joins between primary and foreign key attributes, and the filter selection predicates are added on the non-key attributes.

The Data Generation Problem (DGP) in DataSynth is defined as: Given a database schema and a collection of cardinality constraints $C_1, C_2, \ldots, C_m$ generate a database instance conforming to the schema that satisfies all the constraints. In the decision version of the problem, the output is Yes if there exists a database instance that satisfies all the constraints and No, otherwise. While the general data generation problem is hard, the algorithms are able to handle a large and useful classes of constraints. The algorithms are probabilistically approximate, meaning that they satisfy all constraints in expectation. The algorithms are also sensitive to the complexity of the input cardinality constraints in a precisely quantifiable way and use ideas from probabilistic graphical models.

DataSynth creates a denormalized relation for every relation from the input schema, which comprises non-key attributes of its own relation and all the non-key attributes of all the relations that it references. DataSynth performs *Grid-partitioning* over its data space. Grid-Partitioning first intervalizes the domain of each attribute based on the constants appearing in the CCs, and divides the domain into a grid aligned with the interval boundaries for each attribute. If

a denormalized relation has n attributes, and each attribute gets divided into l intervals, then the domain of the denormalized relation is partitioned into a grid of l^n cells. For each cell in the grid, a variable is created that represents the number of data rows present in that cell. The output of grid partitioning is a group of regions such that the domain points in each region satisfy the same set of CCs.

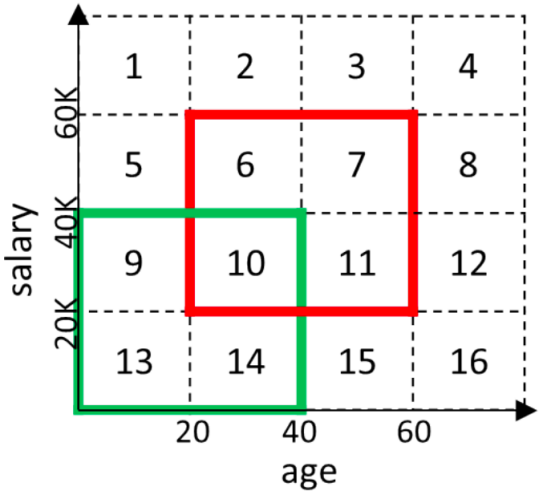


Figure 2.4: Grid Partitioning in DataSynth

The key difference between Hydra's region partitioning and DataSynth's grid partitioning can be seen in Figure 2.4. Each rectangular box (formed with dotted lines) forms a region in DataSynth, whereas only the coloured rectangles, and their intersections correspond to the region formed in Hydra. An important point to note is, the number of regions in grid-partitioning is not minimal, unlike Hydra, i.e., there can be more than one region that satisfies the same CCs. An LP is formulated after this with a variable for each region where the variable signifies the cardinality of the region. The solution of the LP is used to generate a denormalized table in a probabilistic manner, which is used to generate the base tables.

DataSynth severely suffers in scaling to a larger number of queries in the input workload, owing to its grid-partitioning approach, as it creates an explosion in the number of variables and hence becomes difficult for the solver to handle. The probabilistic way of generating tuples also introduces some errors, where the error is defined as the output cardinality mismatch in the CCs.

2.3.5 Touchstone

Touchstone [19] is a workload-aware data generator, that achieves fully parallel data generation, linear scalability, and austere memory consumption for supporting the generation of enormous

query-aware test databases. Touchstone generates one single database instance for multiple queries. In query-aware data generation, we need to handle the extremely complicated dependencies among columns which are caused by the complex workload characteristics specified on the target test queries, as well as the data characteristics specified on the columns. There are two core techniques employed by Touchstone beneath the accomplishments of all above enticing features. Firstly, Touchstone employs a completely new query instantiation scheme adopting the *random sampling* algorithm, which supports a large and useful class of queries, as well as the actual query execution trees from the running database system. Secondly, Touchstone is equipped with a new data generation scheme using the *constraint chain* data structure, which easily enables thread-level parallelization. This scheme allows Touchstone to easily scale up to dozens of nodes and hundreds of cores for parallelized database generation and thus achieves a 1000× performance improvement against [20].

The input to Touchstone contains the database schema H, data characteristics of non-key columns D, and workload characteristics of test queries W. These characteristics W are obtained by running actual queries over the client database, thereby obtaining result cardinality for each edge of the query plan, and then masking the constraints with parameters to obtain parametric constraints. This parameterization is done for privacy reasons. The output of Touchstone consists of two parts, namely the instantiated queries Q and the generated Database instance (DBI). To support the generation procedure, the infrastructure of Touchstone is divided into two major components, which are responsible for query instantiation and data generation, respectively, as shown in Figure 2.5.

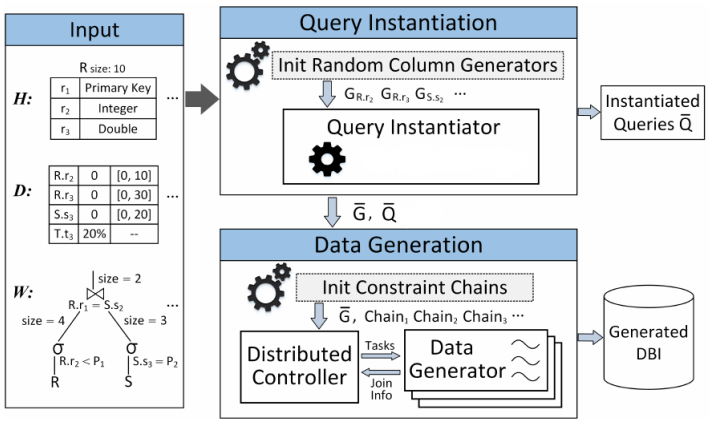


Figure 2.5: Architecture of Touchstone

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 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. The 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.

The data generation module in Touchstone consists of two steps. In the first compilation step, Touchstone orders the tables as with a generation sequence. This is done by constructing a topological order over the tables. In order to decouple the dependencies among columns and facilitate parallelization, Touchstone decomposes the query trees annotated with constraints into constraint chains. A constraint chain with respect to a table is defined as the sequences of constraints with a descendant relationship in the query trees. In the second assembling step, the working threads in Touchstone independently generate tuples for the tables based on the result order from the compilation step. For each tuple, the working thread fills values in the columns by calling the random column generators independently and incrementally assigns a primary key, while leaving the foreign keys blank. By iterating the constraint chains associated with the table, the algorithm identifies the appropriate candidate keys for each foreign key based on the maintained join information of the referenced primary key, and randomly assigns one of the candidate keys to the tuple. Touchstone maintains the join information table to track the status of joinability of primary keys based on a bitmap representation. There are two attributes in the entry of join information table, i.e., bitmap and keys, indicating the status of joinability and the corresponding satisfying primary key values. Note that the keys in the entry may be empty (such entries will not be stored in practice), which means there is no primary key with the desired joinability status.

The data fidelity of synthetic database is evaluated by a relative error on cardinality constraints, and maximum errors are in the range of 1-4%. A few key limitations of Touchstone are the following : (a) No support for filters on key columns, (b) Equality constraints over filters involving multiple columns are not supported, (c) Equi-joins on columns with no reference constraint are not supported and (d) No support for database schema with cyclic reference relationship. Touchstone is similar to HF-Hydra's general framework discussed in Chapter 5, in the sense that both of them take in metadata constraints as additional input other than the query workload, and generates a single synthetic database.

2.3.6 Hydra

We have already seen a brief introduction to Hydra in Chapter 1. However, in this section, we present a detailed overview of Hydra's architecture, along with a summary of its various components and their interactions with the database engine. Figure 2.6 captures the overall architecture of Hydra.

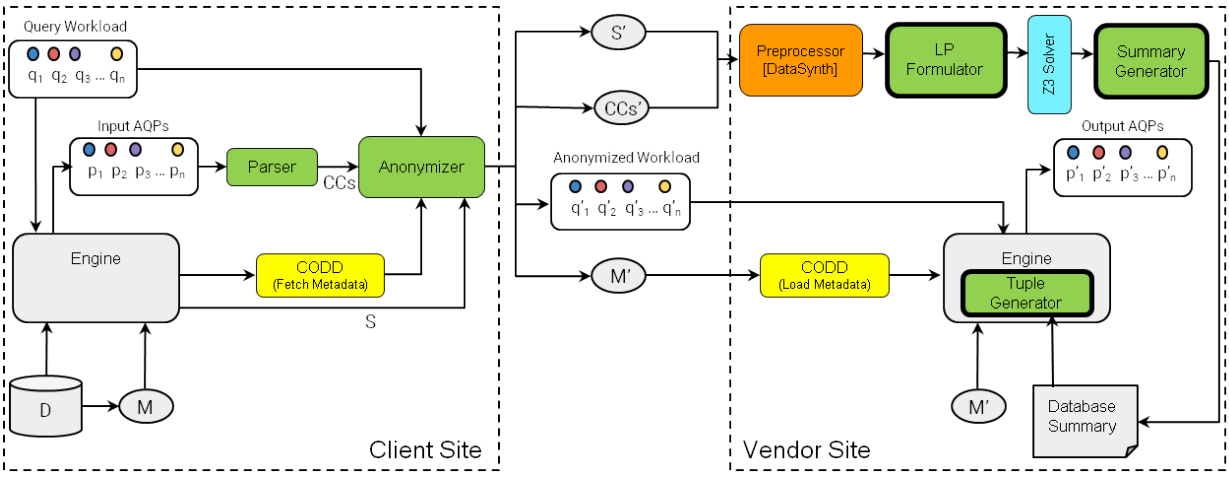


Figure 2.6: **Hydra Architecture**

In this picture, the green boxes show the new components designed specifically for Hydra. Among these, the primary components are the LP Formulator, the Summary Generator, and the Tuple Generator, all shown with thick borders. The other modules have been sourced from the literature, including the preprocessor (orange) from DataSynth [8], the CODD metadata processor (yellow) [24], and the Z3 LP solver (blue) [15]. Now we drill down into each component as follows:

Client Site The information flow from the client to the vendor is as follows: At the client site, Hydra fetches the schema information (S), and the query workload ($q_1, q_2, q_3, ..., q_n$) with its corresponding AQPs ($p_1, p_2, p_3, ..., p_n$) obtained from the database engine. The AQPs are converted to equivalent cardinality constraints (CCs) using a Parser. The metadata from the database catalogs (M) is captured with the help of CODD. In order to address client security concerns, all this information (schema, metadata, queries and CCs) is passed through an Anonymizer that suitably masks the information before shipping it to the vendor site. In this process, non-numeric constants appearing in the queries and plans are mapped to numbers to facilitate LP formulation at the vendor site. Due to this mapping, the final database summary generated at the vendor site also consists of only numeric datatypes. It is possible to reverse

this mapping to get back the original datatypes, but this is not an important consideration for satisfying the cardinality constraints.

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

Hydra's *region-partitioning* strategy is in marked contrast to the grid-partitioning strategy used in [7]. Recall that grid-partitioning first intervalizes the domain of each attribute based on the constants appearing in the CCs, and divides the domain into a grid aligned with the interval boundaries for each attribute. If a sub-view has n attributes, and each attribute gets divided into ℓ intervals, then the domain of the sub-view is partitioned into a grid of ℓ^n cells. For each cell in the grid, a variable is created which represents the number of data rows present in that cell. In contrast, region-partitioning strategy divides the domain into only as few regions as is necessary to precisely write out each cardinality constraint, and assigns one variable corresponding to each region. Typically, this leads to far fewer variables than grid-partitioning, which has one variable per cell in the grid. As the cardinality constraints get more complex, the difference in complexity of the LPs produces by region-partitioning and grid-partitioning become more pronounced. This LP is passed on to the Z3 [15] solver, which provides one of the feasible solutions as the output.

Summary Generator: This module generates the database summary from the LP solutions obtained on the views. Since partitioning is carried out at a sub-view level, the LP solution, which is expressed in terms of sub-view variables, needs to be mapped to equivalents in the original view space. A sampling-based approach was proposed in [7] for this purpose – for example, say a view (A,B,C) is split into a pair of sub-views (A, B) and (B, C), the algorithm computes the distributions $Prob(A, B)$ and $Prob(C|B)$. Then, each tuple is generated by first sampling a point from the former distribution, and then sampling a point from the latter conditioned on this outcome.

However, Hydra chooses not to take this approach since the computational overhead incurred is enormous and the sampling process introduces errors. Instead, Hydra uses an alternative deterministic alignment algorithm, which works purely using summaries. This component is also responsible for ensuring that the generated summary obeys referential integrity.

Tuple Generator: The Tuple Generator resides in the database engine. It ensures that whenever a query is fired, the executor does not need to fetch data from the disk, but instead gets data *on-demand* from the Tuple Generator, using the database summary.

A unique feature of Hydra's 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, depends only on the query workload and *not* on the database scale. It can be used either for *dynamically* generating data during query execution, or optionally for materializing static relations when parameters related to I/O processing need to be monitored. This summary-based approach helps eliminate the enormous time and space overheads incurred by prior techniques in generating and storing data before initiating analysis.

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.

Chapter 3

Operator Coverage Expansion

Hydra does not support the LIKE operator in its input training workload. In HF-Hydra, we extend the ambit of Hydra to include the LIKE operator as well. We achieve this by converting the LIKE predicate into one or more equivalent equality filter predicates. The detailed mechanism is discussed in this chapter.

To start with, from the entire input query workload, we collect all the predicate constants that occur with the LIKE operator, and segregate them on the basis of the attribute they are applied upon. Depending on the number of predicate constants an attribute has, the handling mechanism is classified as follows. To make things clear, we take a toy example with queries on the *Ship_mode* relation from the TPC-DS benchmark schema in each case and explain the process.

3.1 Single LIKE Predicate

The attributes which have only one LIKE predicate applied upon them in the entire training workload fall in this category. LIKE predicate constants can be considered as *regular expressions*. 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.

Consider the following example query Q_1 and its corresponding AQP shown in Figure 3.1.

Q_1: Select * from ship_mode where
sm_code like 'C%';

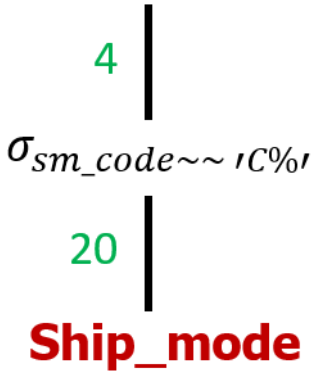


Figure 3.1: AQP for Query Q_1

The cardinality constraint corresponding to the LIKE predicate in Q_1 is:

$$CC_1\text{: } |\sigma_{sm\_code \sim\sim 'C\%'} \text{ (Ship\_mode)}| = 4$$

The LIKE predicate is applied on sm_code, and the predicate constant is 'C%', which we take as our regular expression. 'C' is a valid representative string that satisfies 'C%'. Now the CC can be now transformed as :

$$CC_1\text{: } |\sigma_{sm\_code \sim\sim 'C\%'} \text{ (Ship\_mode)}| = 4$$
$$\rightarrow \mathbf{|\sigma_{sm\_code = 'C'} \text{ (Ship\_mode)}| = 4}$$

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

3.2 Multiple LIKE Predicates

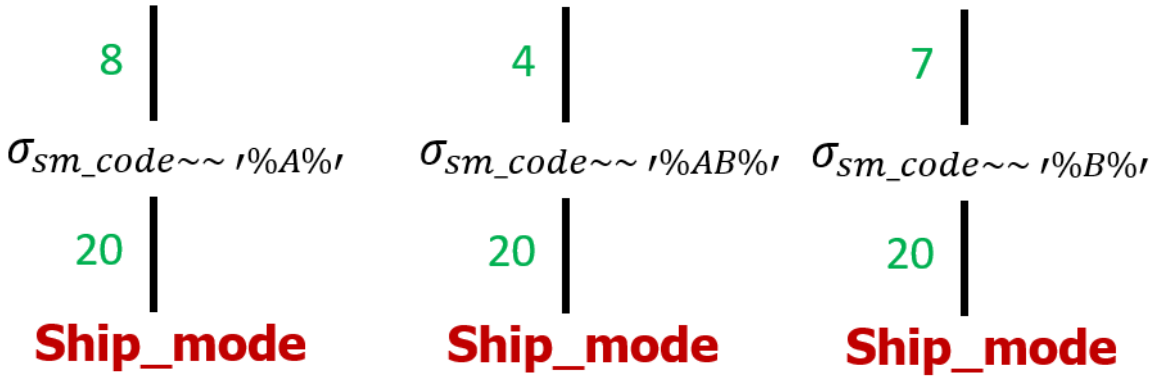
The attributes which have more than one LIKE predicate applied upon them in the entire training workload fall in this category. The data generation mechanism in such cases needs to be aware of any possible intersections among the different predicates that are featured on the same attribute. If no care is taken, then the resultant synthetic database will fail to satisfy the CCs. Let us take an example to understand this better.

Consider the following queries Q_2, Q_3, Q_4, and their corresponding AQPs shown in Figures 3.2a, 3.2b, 3.2c.

Q_2: Select * from ship_mode where
sm_code like '%A%';

Q_3: Select * from ship_mode where
sm_code like '%AB%';

Q_4: Select * from ship_mode where
sm_code like '%B%';



(a) AQP for Q_2 (b) AQP for Q_3 (c) AQP for Q_4

Figure 3.2: AQPs for Multiple LIKE predicate case

The cardinality constraints corresponding to the LIKE predicate for Q_2, Q_3, Q_4 are:

$$CC_2\text{: } |\sigma_{sm\_code \sim\sim '\%A\%'} \text{ (Ship\_mode)}| = 8$$

$$CC_3\text{: } |\sigma_{sm\_code \sim\sim '\%AB\%'} \text{ (Ship\_mode)}| = 4$$

$$CC_4\text{: } |\sigma_{sm\_code \sim\sim '\%B\%'} \text{ (Ship\_mode)}| = 7$$

Let's first try to handle multiple LIKE predicates the same way as the previous subsection, and so for the regular expressions '%A%', '%AB%', and '%B%', let the representative strings be 'A', 'AB', 'B'.

Also, let us consider that Q_2, Q_3, Q_4 are the only queries that form our training workload. Then the database generated from the resultant summary, for the attribute sm_code is shown in Table 3.1.

sm_code	Count
A	8
AB	4
B	7

Table 3.1: Sample synthetic database for sm_code

Now, if we run Q_2, Q_3, Q_4 over this synthetic database, the resulting CCs for LIKE predicates would be:

$$CC_2\text{: } |\sigma_{sm\_code \sim\sim '\%A\%'} \text{ (Ship\_mode)}| = 12$$

$$CC_3\text{: } |\sigma_{sm\_code \sim\sim '\%AB\%'} \text{ (Ship\_mode)}| = 4$$

$$CC_4\text{: } |\sigma_{sm\_code \sim\sim '\%B\%'} \text{ (Ship\_mode)}| = 11$$

Clearly, the CCs from the synthetic data do not match the CCs from the client AQPs, and so we need a different strategy to handle multiple LIKE predicates. This incorrectness has risen because of the overlaps in the regular expressions from the CCs. The regular expressions from the first two CCs have a non-null intersection, and so do the regular expressions from the last two CCs. If there were no such overlaps, or in other words, if the regular expressions were *disjoint*, we can then use the same transformation strategy as used in the previous subsection. So, firstly, we need to identify the *existence* of such overlaps by performing intersections on the individual regular expressions and secondly, come up with a methodology to form *disjoint spaces* within the regular expressions.

3.2.1 Discovering Intersections

Performing intersection on regular expression strings directly is *non-trivial*, and so, we adopt an *indirect approach*. We make use of concepts from *automata theory* to convert **Regular expression → Epsilon-NFA → NFA → DFA**. Note that DFAs are closed under intersections. Once we have equivalent DFAs for our regular expressions, we can now easily perform intersections on every pair of DFA and produce another DFA, which would be then equivalent to the intersection of the regular expressions. This exercise essentially gives us a Venn diagram denoting the powerset of intersections. An example Venn diagram for three regular expressions R_1, R_2, R_3 can be seen in Figure 3.3. In our toy example, for 3 regular expressions, there will be a total of seven spaces in the Venn Diagram. In general, for k number of regular expressions, we will have $2^k - 1$ number of spaces in the Venn Diagram.

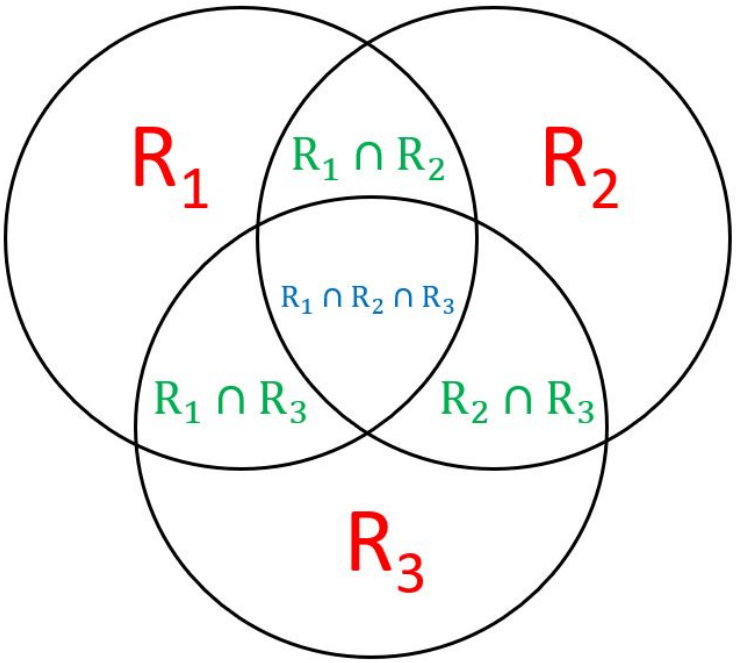


Figure 3.3: Venn diagram showing intersections for all DFAs

Recall that the goal was to generate disjoint individual spaces so as to completely avoid any possible overlaps among the regular expressions. The seven spaces in Figure 3.3 are still *not disjoint*. To make each space truly disjoint, we can intersect each space with its corresponding *complement* in the diagram. Note that DFAs are closed under complement as well so, at the end of this exercise, we would get seven *disjoint* spaces and a corresponding DFA for each of them. The final disjoint Venn Diagram for three regular expressions R_1, R_2, R_3 looks as shown in Figure 3.4.

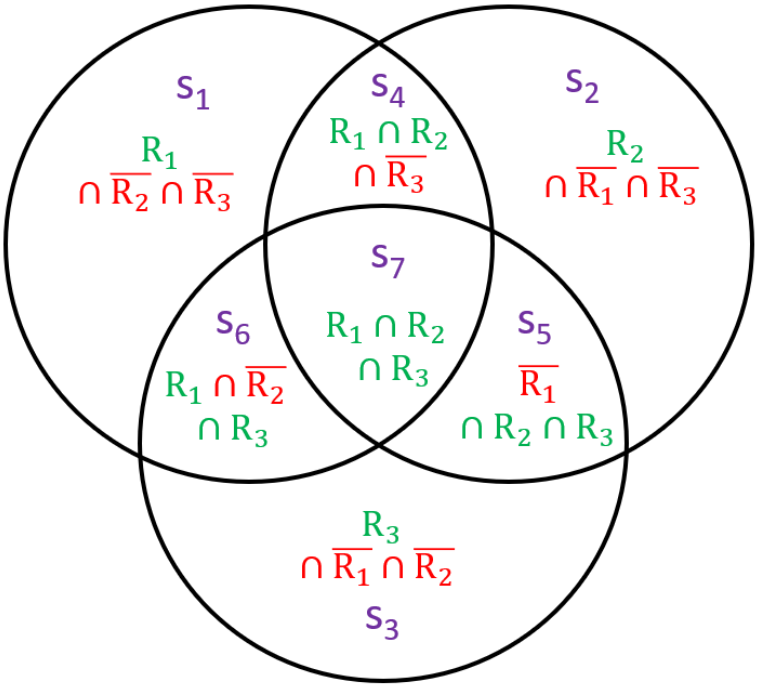


Figure 3.4: Venn diagram showing all disjoint spaces

For each of the seven spaces, we have a corresponding DFA from the Venn Diagram. Each of these DFA can be converted back to a regular expression. Now the initial regular expressions from the CCs that formed the training workload can be expressed as a conjunction of these disjoint constituent regular expressions. For example, R_1 is now broken down into four disjoint spaces s_1, s_4, s_6, s_7. From the DFAs corresponding to s_1, s_4, s_6, s_7, we can get four regular expressions r_1, r_4, r_6, r_7, and equivalently express R_1 as

$$R_1 = r_1 \cup r_4 \cup r_6 \cup r_7$$

Likewise, R_2 and R_3 can be expressed as

$$R_2 = r_2 \cup r_4 \cup r_5 \cup r_7$$
$$R_3 = r_3 \cup r_5 \cup r_6 \cup r_7$$

3.2.2 Representative strings for disjoint spaces

Now that we have achieved disjointness in the regular expressions, we can proceed to generate representative strings for each space. For each of the seven spaces $s_1, s_2, \ldots, s_7$, the corresponding regular expressions $r_1, r_2, \ldots, r_7$, and their representative strings are listed in Table 3.2.

Note that there may be cases in which the intersection of a pair of DFA resulted in NULL, and so the DFA for such spaces and their corresponding regular expression would simply be NULL. In our toy example, spaces x_2, x_4, x_5 are NULL. Such spaces can be simply ignored in the transformation process.

Disjoint Regular Expression		Representative String
r_1	$(\%A\%) \cap (\overline{\%AB\%}) \cap (\overline{\%B\%})$	A
r_2	$(\overline{\%A\%}) \cap (\%AB\%) \cap (\overline{\%B\%})$	NULL
r_3	$(\overline{\%A\%}) \cap (\overline{\%AB\%}) \cap (\%B\%)$	B
r_4	$(\%A\%) \cap (\%AB\%) \cap (\overline{\%B\%})$	NULL
r_5	$(\overline{\%A\%}) \cap (\%AB\%) \cap (\%B\%)$	NULL
r_6	$(\%A\%) \cap (\overline{\%AB\%}) \cap (\%B\%)$	BA
r_7	$(\%A\%) \cap (\%AB\%) \cap (\%B\%)$	AB

Table 3.2: Representative strings for disjoint spaces

3.2.3 Cardinality assignment for disjoint spaces

To generate data for each disjoint space using its representative string, we need to split the cardinality of each CC to its constituent disjoint spaces. For this purpose, we frame each CC into a linear equation. Each disjoint space is assigned a variable. Collectively, we then get a system of linear equations in these variables. These equations are solved to yield the cardinalities for the constituent disjoint spaces.

For our toy example, the system of linear equations would be:

$$R_1\colon\ x_1 + x_6 + x_7 = 8$$
$$R_2\colon\ x_7 = 4$$
$$R_3\colon\ x_3 + x_6 + x_7 = 7$$

Note that the spaces which turned out to be NULL are ignored in the equations. A feasible solution for the above system of linear equations is:

$$x_1 = 2, x_3 = 1, x_6 = 2, x_7 = 4$$

3.2.4 Final Transformation

The LIKE predicates of CC_2, CC_3, CC_4 from Q_2, Q_3, Q_4 are finally transformed using their constituent representative strings and corresponding cardinalities, as shown in Table 3.3. Note that collectively, there could be some duplicate CCs, which stemmed out from the intersection among the CCs. However, we take into consideration the unique ones only (highlighted in boldface). These transformed CCs can be handled the same way other equality filter predicates are handled with the existing setup.

Original CC		Transformed CC(s)
CC_2	$\lvert\sigma_{sm\_code \sim\sim '\%A\%'}$ (Ship_mode)$\rvert = 8$	$\boldsymbol{\lvert\sigma_{(sm\_code = 'A')}\ \text{(Ship\_mode)}\rvert = 2}$ $\boldsymbol{\lvert\sigma_{(sm\_code = 'BA')}\ \text{(Ship\_mode)}\rvert = 2}$ $\boldsymbol{\lvert\sigma_{(sm\_code = 'AB')}\ \text{(Ship\_mode)}\rvert = 4}$
CC_3	$\lvert\sigma_{sm\_code \sim\sim '\%AB\%'}$ (Ship_mode)$\rvert = 4$	$\lvert\sigma_{(sm\_code = 'AB')}$ (Ship_mode)$\rvert = 4$
CC_4	$\lvert\sigma_{sm\_code \sim\sim '\%A\%'}$ (Ship_mode)$\rvert = 7$	$\boldsymbol{\lvert\sigma_{(sm\_code = 'B')}\ \text{(Ship\_mode)}\rvert = 1}$ $\lvert\sigma_{(sm\_code = 'BA')}$ (Ship_mode)$\rvert = 2$ $\lvert\sigma_{(sm\_code = 'AB')}$ (Ship_mode)$\rvert = 4$

Table 3.3: Final Transformation of CCs with LIKE Predicates

3.2.5 Optimization Measures

The intersection of two DFAs could result in an explosion in the number of states of the resultant DFA, and so, it could get complex to compute subsequent intersections up the pipeline. Hence as an optimization measure, we minimize the resultant DFA after each individual intersection and then proceed with subsequent intersections.

The algorithm discussed here has exponential time complexity. But, in realistic workloads,

we can expect most intersections to end up in NULL, and so the subsequent intersections up the pipeline would be totally avoided. Hence, our proposed algorithm for handling intersections among regular expressions combined with minimization of resultant states at each step of the process can be expected to ensure pragmatic computational overheads.

3.3 Multiple LIKE predicates in a single CC

In our toy example for multiple LIKE predicates, we had looked at multiple LIKE predicates accumulated from CCs over a workload for a given attribute, but in all of those CCs, we had a single LIKE predicate within each CC. But if a single training query has multiple LIKE predicates applied in it, then the corresponding filter CC will have multiple LIKE predicates in it as well. The CC might have the multiple predicates either in disjunction or in conjunction, depending on the actual training query. We discuss the procedure to handle such cases here.

3.3.1 Multiple predicates in conjunction

For multiple predicates from a single CC in conjunction, we can take up the individual predicates and compute intersections with other predicates from the other CCs in the workload and create disjoint spaces as discussed already. In the transformation phase of converting the LIKE predicates into equality predicates, instead of taking up the representative strings from individual spaces, we can directly take up the representative string corresponding to the intersection of all the predicates that form the conjunction in the CC under consideration.

As an example, let us consider the workload to have only two CCs, namely CC_4 from the earlier section and the following CC_5:

$$CC_4\text{: } |\sigma_{sm\_code \sim\sim \prime\%B\%\prime} \text{ (Ship\_mode)}| = 11$$

$$CC_5\text{: } |\sigma_{((sm\_code \sim\sim \prime\%A\%\prime) \wedge (sm\_code \sim\sim \prime\%AB\%\prime))} \text{ (Ship\_mode)}| = 4$$

We take up regular expressions '%A%', '%AB%', '%B%' and proceed to get disjoint spaces. For the transformation of CC_5, we can directly generate a representative string from the regular expression obtained from DFA corresponding to the highest granularity for CC_5. i.e. '%A%' $\cap$ '%AB%'. Let the representative string be 'AB'. The transformation of CC_5 would then be:

$$CC_5\text{: } |\sigma_{(sm\_code \sim\sim \prime\%A\%\prime) \wedge (sm\_code \sim\sim \prime\%AB\%\prime)} \text{ (Ship\_mode)}| = 4$$
$$\rightarrow \boldsymbol{|\sigma_{((sm\_code = \prime AB\prime))} \textbf{ (Ship\_mode)}| = 4}$$

3.3.2 Multiple predicates in disjunction

For multiple predicates from a single CC in disjunction, we can simply choose to retain any one of the available predicates and drop the rest. From there, we can proceed with the rest of the workload by computing intersections, creating disjoint spaces, and transforming them back into equality predicates.

As an example, let us consider we have the following CC:

$$CC_6\text{: } |\sigma_{((sm\_code \sim\sim \text{'\%A\%'}) \vee (sm\_code \sim\sim \text{'\%B\%'}))} \text{ (Ship\_mode)}| = 20$$

Here we have two predicates with regular expressions '%A%' and '%B%'. We can choose to retain any one of them, and drop the other. The transformation would then be as follows:

$$CC_6\text{: } |\sigma_{((sm\_code \sim\sim \text{'\%A\%'}) \vee (sm\_code \sim\sim \text{'\%B\%'}))} \text{ (Ship\_mode)}| = 20$$
$$\rightarrow \boldsymbol{|\sigma_{sm\_code \sim\sim \text{'\%A\%'}} \text{ (Ship\_mode)}| = 20}$$
and eventually,
$$\rightarrow \boldsymbol{|\sigma_{((sm\_code = \text{'A'}))} \text{ (Ship\_mode)}| = 20}$$

Chapter 4

Managing Scalability For Training Queries

In this chapter, we look into our specialized case that focuses only on handling training queries. We discuss in detail our simplified approach to bypass the scalability limitation and manage to handle any number of training queries in HF-Hydra. We achieve this by working on the level of each individual query. Also, in this specialized setting, we extend support to include projection-based operators in our training query.

We begin the discussion with an overview of our simplified approach. We then proceed to look into different scenarios of training queries and how relation summaries are generated in each case. To make things concrete, in each case, we present a query derived from the TPC-DS benchmark suite, as a running example. Subsequently, we look into the special case of handling LIKE predicates when we deal with individual queries and also queries with projections. Finally, we talk about how dynamic generation is performed and the overall picture of managing scalability for training queries.

4.1 Overview

Hydra takes in a workload of training queries and produces a summary that ensures volumetric similarity. As we discussed in section 1.1.3, Hydra carries out view formation, region partitioning, and then extracts individual relations back from the views by ensuring no referential integrity violations and eventually constructs the individual relation summaries.

However, for the AQP from a single query, all the corresponding CCs are essentially subtrees, and so they have a superset-subset relationship. Hence the regions corresponding to each individual participating relation from these CCs readily come out to be disjoint. By virtue of

this, when working at the level of individual queries, we can *directly handcraft* a data point adhering to this region and add it to our relation summary similar to the ones seen in Figure 1.5, thus skipping all the above-discussed processing that is critical while dealing with a workload of queries.

Each data point we fabricate for a given CC of a relation will consist of a row containing values for all the attributes (except primary key (PK)) of the relation, according to its schema. But the row will have actual values for only the attributes that take part in the query. The rest of the attributes can be assigned any values, or even simply left out as null. A numerical cardinality is associated with each row. During dynamic generation, this row is replicated the specified cardinality number of times, with consecutive natural numbers used as the primary key value. The next row in summary takes on from the next natural number as its PK value where the previous row left off.

Note that only PK is left out from our fabrication, but the Foreign key (FK), if any, for a relation is fabricated as part of the rows in the summary itself. Hence, for a given fact relation, it is critical to know the PK values of the dimension relation with whom it is about to join (borrow PK) during query execution. Hence we process the relations in a topological order of their PK-FK dependencies.

For a single relation, with presence/absence of filter predicates and projection attributes, we have four possible scenarios. Similarly, for multiple relations, we have four more possible scenarios. However we look at only the scenarios which are comparatively complex in the upcoming sections. The solutions for other scenarios can be trivially extended. In each section, we will look into how HF-Hydra handles queries of increasing complexity with filters and projections and generates the individual relation summaries.

4.2 Query with Single relation

The query template under consideration is as follows :

Select distinct $P_1, P_2, ..., P_n$
from R
where $F_1, F_2, ..., F_n$;

Here, $P_1, P_2, ..., P_n$ are the attributes projected from Relation R
and $F_1, F_2, ..., F_n$ are the filter predicates that are applied on R.

As an example, let us consider Query Q_1 and its corresponding AQP shown in Figure 4.1.

Q_1: Select distinct d_month_seq
from date_dim where
d_year = 2001;

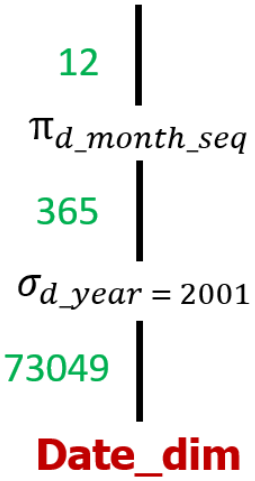


Figure 4.1: AQP for Q_1

The CCs from the AQP for Q_1 would be :

$|\pi_{d\_month\_seq}$ (Date_dim)$| = 12$
(Let us term the CCs of this type as **Projection CC**, with the attributes that appear in the CC as $\mathbf{P_{Attr}}$ and the cardinality as $\mathbf{P_{Card}}$)
$|\sigma_{d\_year=2001}$ (Date_dim)$| = 365$
(Let us term the CCs of this type as **Filter CC**, with the attributes that appear in the CC as $\mathbf{F_{Attr}}$ and the cardinality as $\mathbf{F_{Card}}$)
|Date_dim| = 73049
(Let us term the CCs of this type as **Relation CC**, and the cardinality as $\mathbf{R_{Card}}$)

The procedure to generate the relation summary for R is as follows:

1. Create a template row with placeholders for each attribute in the relation. Fabricate the corresponding value at the index of $\mathbf{F_{Attr}}$ with a **constant that satisfies the filter predicate(s).** We can populate the other attributes in the row with any legal value from the corresponding datatype for each attribute.

2. Create $\mathbf{P_{Card} - 1}$ number of rows from the template obtained from Step 1, and fabricate unique values for $\mathbf{P_{Attr}}$ in every row. If any of the attributes in $\mathbf{P_{Attr}}$ is also in $\mathbf{F_{Attr}}$, then it is made sure the unique fabricated value still satisfies the filter predicate. This can be done by choosing different predicate values that feature in the filter CC to fabricate each row for the common attribute(s) in $\mathbf{P_{Attr}}$ and $\mathbf{F_{Attr}}$. Each of these rows is added to the relation summary with cardinality 1.

3. Create a row from the template obtained from Step 1, and choose a different value than the one used in Step 2, and fabricate that value for $\mathbf{P_{Attr}}$. This row is added to the relation summary with cardinality $\mathbf{F_{Card} - P_{Card} + 1}$. Step 3 will satisfy the Filter CC. Steps 2 and 3 together will satisfy the Projection CC, since, among the total $\mathbf{P_{Card}}$ unique values needed to satisfy projection post the application of filter predicate, $\mathbf{P_{Card} - 1}$ unique values come from Step 1, and the last unique value comes from Step 2.

4. Create a row with values for each attribute in the schema filled with any legal value from the corresponding datatype for each attribute. This row is added to the database summary with cardinality $\mathbf{R_{Card} - F_{Card}}$. Steps 2, 3, 4 together satisfy the Relation CC, as the total rows required for this relation together come from all the three steps.

Since we deal with a single relation here, the relation summary for R forms our database summary, which is subsequently used for database generation.

4.3 Query with Multiple relations

The query template under consideration is as follows :

Select distinct $P_1, P_2, ..., P_n$ from $R_1, R_2, ..., R_n$ where <Join clause for $R_1, R_2, ..., R_n$ > and $F_1, F_2, ..., F_n$;

Here, $P_1, P_2, \ldots, P_n$ are the attribute(s) projected
together from all relations $R_1, R_2, \ldots, R_n$
and $F_1, F_2, \ldots, F_n$ are the filter predicate(s)
that are applied on relations $R_1, R_2, \ldots, R_n$.
Let us now discuss some sub cases of this template to understand.

4.3.1 Query with multiple relations and filters

Let us consider Query Q_2 and its corresponding AQP shown in Figure 4.2.

Q_2: Select * from
web_sales, customer, customer_address, date_dim where
ws_bill_customer_sk = c_customer_sk and
c_current_addr_sk = ca_address_sk and
ws_sold_date_sk = d_date_sk and
c_birth_year > 1950 and
c_birth_month in (6,5,12,4,3,7) and
ca_state in ('TN','IL','GA','MO','CO','OH','NM') and
d_year = 2001;

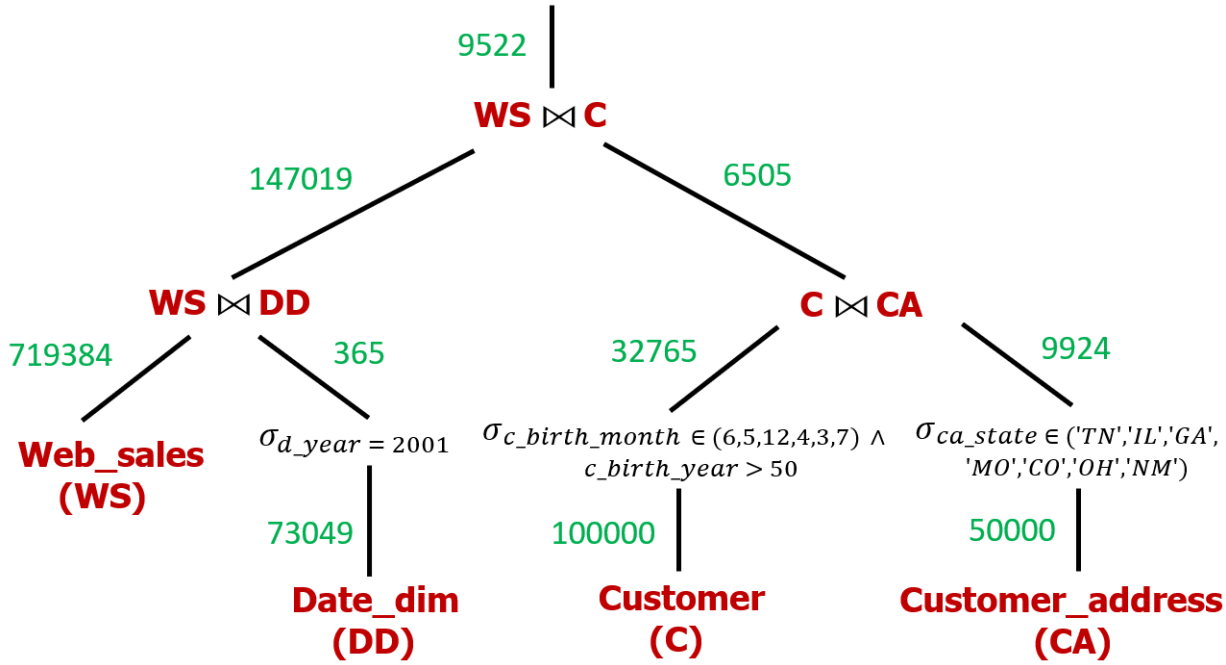


Figure 4.2: AQP for Q_2

The CCs for the query would be :

$$|\sigma_{d\_year=2001 \wedge c\_birth\_year>50 \wedge c\_birth\_month\in(6,5,12,4,3,7) \wedge ca\_state\in('TN','IL','GA','MO','CO','OH','NM')}$$
$$(\text{Customer} \bowtie \text{Customer\_address} \bowtie \text{Date\_dim} \bowtie \text{Web\_Sales})| = 9522$$
$$|\sigma_{d\_year=2001}(\text{Date\_dim} \bowtie \text{Web\_Sales})| = 147019$$

$$|\sigma_{c\_birth\_year>50 \wedge c\_birth\_month\in(6,5,12,4,3,7) \wedge ca\_state\in('TN','IL','GA','MO','CO','OH','NM')}$$
$$(\text{Customer} \bowtie \text{Customer\_address})| = 6505$$
$$|\sigma_{c\_birth\_year>50 \wedge c\_birth\_month\in(6,5,12,4,3,7)}\ (\text{Customer})| = 32765$$
$$|\sigma_{ca\_state\in('TN','IL','GA','MO','CO','OH','NM')}\ (\text{Customer\_address})| = 9924$$
$$|\sigma_{d_{year}=2001}\ (\text{Date\_dim})| = 365$$
$$|\text{Web\_sales}| = 719384,\ |\text{Customer}| = 100000$$
$$|\text{Customer\_address}| = 50000,\ |\text{Date\_dim}| = 73049$$

For queries with multiple relations, the fabrication is done in two steps: (a) Firstly, we fabricate data directly for non-key attributes for individual relations, so that the Filter CCs that are applied on the relations are satisfied. (b) Secondly, we fabricate values for the remaining CCs which correspond to join nodes. The join nodes are handled by proper FK value fabrication in the borrowed key attributes of the relations. The number of tuples flowing through each of the intermediate join nodes depend on the actual PK-FK value match among the joining relations, and hence by properly fabricating the FK values, we control the flow of tuples at each stage to satisfy each CC with join node.

The procedure to generate relation summaries for all the participating relations is as follows:

1. For every relation with **both Filter and Relation CCs**, the fabrication process is as per the following steps:

   In Q_2, Date_dim, Customer, and Customer_address fall under this category.

   (a) Create a row with placeholders for each attribute in the relation. Fabricate the corresponding value at the index of $\mathbf{F_{Attr}}$ with a **constant that satisfies the filter predicate(s).** We can populate the other non-key attributes in the row with any legal value from the corresponding datatype for each attribute.

   (b) For the row from Step (a), the FK attributes are fabricated as follows :

      i. For every FK attribute whose corresponding dimension relation **has Filter CC** applied on them, we need to make sure that we choose a legal value from the range of the primary key values that **satisfy the Filter CC.**

      ii. For every FK attribute whose corresponding dimension relation **does not have Filter CC** applied on them, we can choose any legal value from the range of the primary key values.

This row is added to the relation summary with cardinality $\mathbf{F_{Card}}$. This will satisfy the Filter CC. We term such fabricated rows as **Filter satisfying rows.**

(c) Create a row with placeholders for each attribute in the relation. Fabricate the corresponding value at the index of $\mathbf{F_{Attr}}$ with **any constant that does not satisfy the filter predicate(s).** We can populate the other non-key attributes in the row with any legal value from the corresponding datatype for each attribute.

(d) For each row from Step (c), we can populate the FK attributes with any legal value from the range of the primary key values of their corresponding dimension relations. This row is added to the relation summary with cardinality $\mathbf{R_{Card}} - \mathbf{F_{Card}}$. We term such fabricated rows as **Filter non-satisfying rows.** Rows from Step (a) and (c) will together satisfy the Relation CC.

2. For every relation with **only Relation CC**, the fabrication process is as per the following steps:

   In Q_2, Web_Sales fall under this category.

   (a) Create a row with placeholders for each attribute in the relation. We can populate the non-key attributes in the row with any legal value from the corresponding datatype for each attribute.

   (b) For every row from Step (a), the FK attributes are fabricated in the same way as discussed in **Step 1(b)**. This row is added to the relation summary with cardinality $\mathbf{R_{Card}}$. This will satisfy the Relation CC. We term such fabricated rows as **Trivial rows.**

4.3.2 Query with multiple relations and projection

Let us consider query Q_3 and its corresponding AQP as shown in Figure 4.3.

Q_3: Select distinct ws_net_paid, c_birth_year, c_birth_month, ca_zip, ca_state from
web_sales, customer, customer_address, date_dim where
ws_bill_customer_sk = c_customer_sk and
c_current_addr_sk = ca_address_sk and
ws_sold_date_sk = d_date_sk;

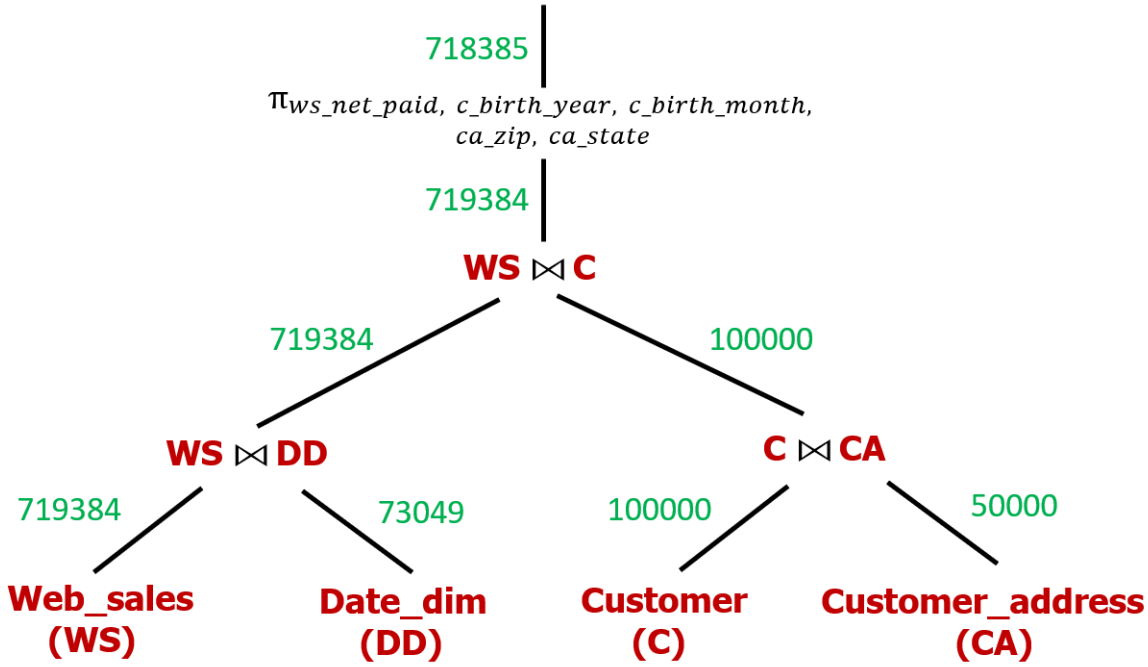


Figure 4.3: AQP for Q_3

The CCs for the query would be :

$$|\pi_{\substack{ws\_net\_paid,\\ c\_birth\_year,\\ c\_birth\_month,\\ ca\_zip,\\ ca\_state}} (\text{Customer} \bowtie \text{Customer\_address} \bowtie \text{Date\_dim} \bowtie \text{Web\_Sales})| = 718385$$

$$|\text{Customer} \bowtie \text{Customer\_address} \bowtie \text{Web\_sales} \bowtie \text{Date\_dim}| = 719384$$

$$|\text{Customer} \bowtie \text{Customer\_address} \bowtie \text{Web\_sales}| = 719384$$

$$|\text{Customer} \bowtie \text{Customer\_address}| = 100000$$

$$|\text{Web\_sales}| = 719384,\ |\text{Customer}| = 100000$$

$$|\text{Customer\_address}| = 50000,\ |\text{Date\_dim}| = 73049$$

In the case of projections applied in a query with multiple relations, the projection node appears as the topmost node in the AQP. For the case of multiple relations, we make a finer adjustment in our classification of Projection CC as follows:
(a) We consider Projection CC to be applied only on the final fact relation in the join sequence. (Here, only Web_sales is considered to have Projection CC).
(b) Also, we classify the attributes that appear in the Projection CC into two groups:
$\mathbf{P_{Attr}}$ are the attributes from the final fact table in the join. (Here, it is ws_net_paid)
Borrowed projected attributes are the attributes projected from other relations. (Here,

they are c_birth_year, c_birth_month, ca_zip, ca_state)

The procedure to generate relation summaries for all the participating relations is as follows:

1. For the relation with **both Projection and Relation CC**, the fabrication process is as per the following steps:

   In Q_3, Web_sales fall under this category.

   (a) Create $\mathbf{P_{Card} - 1}$ number of rows with placeholders for each attribute in the relation. Fabricate the corresponding value at the index of $\mathbf{P_{Attr}}$ with **unique values** in every row. We term such fabricated rows as **Projection satisfied rows.**

   (b) For every row from Step (a), the FK attributes are fabricated as follows:

      i. For FK attributes, whose **dimension relations have borrowed projected attributes**, we need to make sure that we choose a **unique legal value** from the range of the PK values. In case we exhaust all the unique values in the range of PK values, then we can sort to using repetition.

      ii. For FK attributes, whose **dimension relations do not have borrowed projected attributes**, we can choose **any** legal value from the range of the PK values of the corresponding dimension table.

   We can populate the other non-key attributes in the rows from Step (a) with any legal value from the corresponding datatype for each attribute. Each of these rows is added to the relation summary with cardinality 1.

   (c) Create a row with placeholders for each attribute in the relation and fabricate $\mathbf{P_{Attr}}$ with a different value than all $\mathbf{P_{Attr}}$ from rows of Step (a). We term such fabricated rows as **Projection non-satisfied rows.**

   (d) For every row from Step (c), the FK attributes are fabricated as follows:

      i. For FK attributes, whose **dimension relations have borrowed projected attributes**, we must choose **the same** legal value from the range of the PK values of the corresponding dimension table.

      ii. For FK attributes, whose **dimension relations do not have borrowed projected attributes**, we can choose **any** legal value from the range of the PK values of the corresponding dimension table.

We can populate the other non-key attributes in the rows from Step (c) with any legal value from the corresponding datatype for each attribute. This row is added to the relation summary with cardinality $\mathbf{R_{Card} - P_{Card} + 1}$.

The rows from Step (a) and (c) together will satisfy the Projection and Relation CC.

2. For every relation with **only Relation CC and borrowed projected attributes**, the fabrication process is as per the following steps:

   In Q_3, Customer and Customer_address fall under this category.

   (a) Create $\mathbf{R_{Card}}$ number of **Projection satisfied rows** with unique values for all **borrowed projected attributes**. Each of these rows is added to the relation summary with cardinality 1. This satisfies the Relation CC. Projections are indirectly applied on relations of this type, i.e., when the fact table borrows these Projection satisfied rows, then they will participate in the projection that is directly applied on the fact table.

   (b) For every row from Step (a), the FK attributes are fabricated in the same way as discussed in **Step 1(d)**.

3. For every relation with **only Relation CC**, the fabrication process is as per the following steps:

   In Q_3, Date_dim fall under this category.

   (a) Create a **Trivial row**, and add it to the relation summary with cardinality $\mathbf{R_{Card}}$. This will satisfy the Relation CC.

   (b) For every row from Step (a), the FK attributes are fabricated in the same way as discussed in **Step 1(d)**.

4.3.3 Query with multiple relations, filters, and projection

Consider query Q_4 and its corresponding AQP shown in Figure 4.4.

Q_4: Select distinct ws_net_paid, c_birth_year, c_birth_month, ca_zip, ca_state from
web_sales, customer, customer_address, date_dim where
ws_bill_customer_sk = c_customer_sk and
c_current_addr_sk = ca_address_sk and
ws_sold_date_sk = d_date_sk and

c_birth_year > 1950 and
c_birth_month in (6,5,12,4,3,7) and
ca_state in ('TN','IL','GA','MO','CO','OH','NM') and
d_year = 2001 and
ws_net_profit > 1;

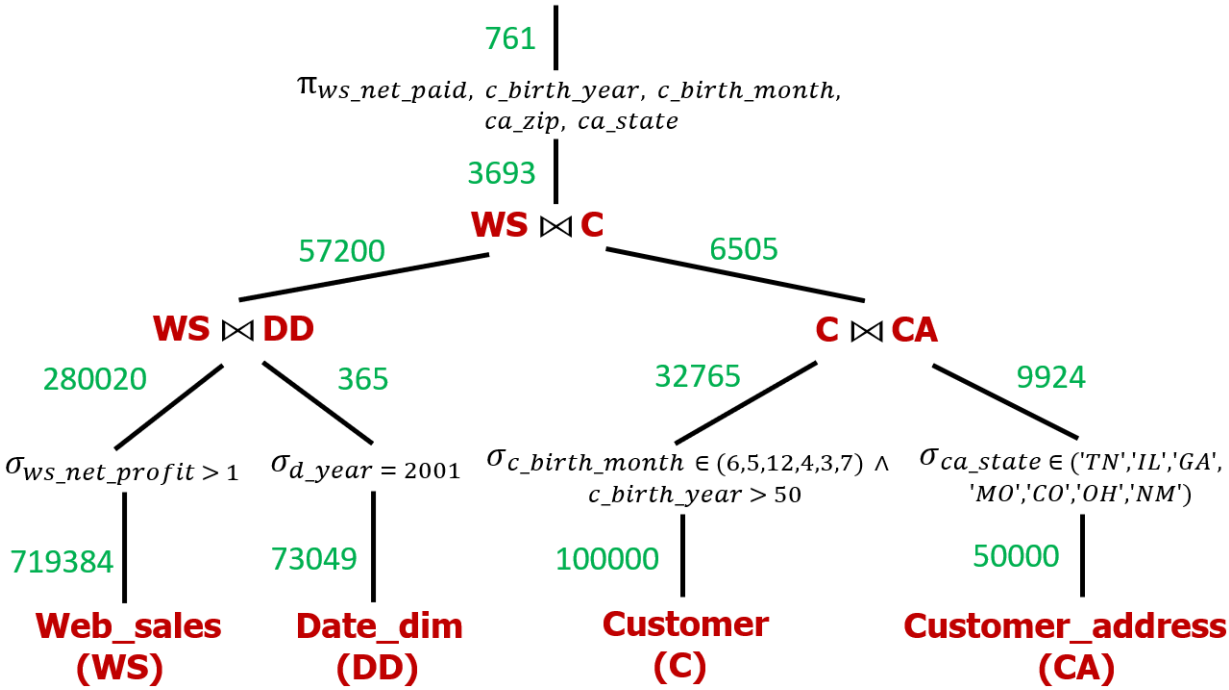


Figure 4.4: AQP for Q_4

The CCs for the query would be :

$$|\pi_{\substack{ws\_net\_paid,\\ c\_birth\_year,\\ c\_birth\_month,\\ ca\_zip,\\ ca\_state}} (\sigma_{\substack{d\_year=2001 \wedge c\_birth\_year>50\\ \wedge\, c\_birth\_month \in (6,5,12,4,3,7)\\ \wedge\, ca\_state \in ('TN','IL','GA',\\ 'MO','CO','OH','NM')\\ \wedge\, ws\_net\_profit>1}} (\text{Customer} \bowtie \text{Customer\_address} \bowtie \text{Date\_dim} \bowtie \text{Web\_Sales})| = 761$$

,

$$|\sigma_{\substack{d\_year=2001 \wedge ca\_state \in ('TN','IL','GA','MO','CO','OH','NM')\\ \wedge\, c\_birth\_year>50 \wedge c\_birth\_month \in (6,5,12,4,3,7) \wedge ws\_net\_profit>1}} (\text{Customer} \bowtie \text{Customer\_address} \bowtie \text{Date\_dim} \bowtie \text{Web\_Sales})| = 3693$$

,

$$|\sigma_{d\_year=2001 \wedge ws\_net\_paid>1}(\text{Date\_dim} \bowtie \text{Web\_Sales})| = 57200$$

,

$$|\sigma_{\substack{c\_birth\_year>50 \wedge c\_birth\_month\in(6,5,12,4,3,7) \\ \wedge ca\_state\in('TN','IL','GA','MO','CO','OH','NM')}} (\text{Customer} \bowtie \text{Customer\_address})| = 6505$$

$$|\sigma_{ws\_net\_profit>1} (\text{Web\_sales})| = 280020$$
$$|\sigma_{c\_birth\_year>50 \wedge c\_birth\_month\in(6,5,12,4,3,7)} (\text{Customer})| = 32765$$
$$|\sigma_{ca\_state\in('TN','IL','GA','MO','CO','OH','NM')} (\text{Customer\_address})| = 9924$$
$$|\sigma_{d_{year}=2001} (\text{Date\_dim})| = 365$$
$$|\text{Web\_sales}| = 719384, |\text{Customer}| = 100000$$
$$|\text{Customer\_address}| = 50000, |\text{Date\_dim}| = 73049$$

The procedure to generate relation summaries for all the participating relations is as follows:

1. For every relation with **Projection, Filter, and Relation CCs**, the fabrication process is as per the following steps:

   In Q_4, Web_sales fall in this category with the filter on ws_net_profit and ws_net_paid as its projected attribute.

   (a) Create a template **Filter satisfying row.**

   (b) Create $\mathbf{P_{Card} - 1}$ **Projection satisfying rows** from the template obtained from Step 1 with **unique values** for $\mathbf{P_{Attr}}$, as the same way discussed in **Step 2 of Section 4.2**. Each of these rows is added to the relation summary with cardinality 1. This will satisfy the Projection CC.

   (c) Create a **Projection non-satisfying row** from the template obtained from Step 1 with the **same values** for $\mathbf{P_{Attr}}$, as the same way discussed in **Step 3 of Section 4.2**, and add this row to the relation summary with cardinality $\mathbf{F_{Card} - P_{Card} + 1}$. Rows from Steps (b) and (c) here together will satisfy the Projection and Filter CC.

   (d) Create a **Filter non-satisfying row**, and add this row to the relation summary with cardinality $\mathbf{R_{Card} - F_{Card}}$. This will satisfy Relation CC.

   (e) For each row from Step (b), the FK attributes are fabricated as follows:

      i. For the FK attribute whose dimension relation **have Filter CC and borrowed projected attributes**, we need to make sure that we choose a **unique legal value** from the range of the primary key values which have **satisfied their Filter CC**. In case we exhaust all the unique values in the range of primary key values, then we can sort to using repetition.
      In Q_4, FKs of rows of Web_sales referenced from Customer and Customer_address would fall under this category.

ii. For the FK attribute whose dimension relation **has no Filter CC but has borrowed projected attributes**, we need to make sure that we choose any legal **unique value** from the range of the primary key values of the corresponding dimension relation.
In Q_4, FKs of rows of Web_sales referenced from Customer and Customer_address would have fallen under this category if those relations did not have any filters.

iii. For the FK attribute whose dimension relation **have Filter CC alone**, we need to make sure that we choose **any** legal value from the range of the primary key values that have **satisfied their Filter CC**.
In Q_4, FKs of rows of Web_sales referenced from Date_dim fall under this category.

iv. For the FK attribute whose dimension relation **do not have Filter CC** applied on them, we can choose **any** legal value from the range of the primary key values.

(f) For each row from Step (c), the foreign key attributes are fabricated as follows:

i. For the FK attribute whose dimension relation **have Filter CC and borrowed projected attributes**, we need to make sure that we choose **the same** legal value from the range of the primary key values which **satisfy their Filter CC**.

ii. For the FK attribute whose dimension relation **has no Filter CC but has borrowed projected attributes**, we need to make sure that we choose **the same** legal value from the range of the primary key values which have **do not satisfy their Filter CC**.

iii. For the FK attribute whose dimension relation **have Filter CC alone**, we need to make sure that we choose **any** legal value from the range of the primary key values that **satisfy the Filter CC.**

iv. For the FK attribute whose dimension relation **do not have Filter CC** applied on them, we can choose **any** legal value from the range of the primary key values.

(g) For each row from Step (d), we can populate the FK attributes with **any** legal value from the range of the primary key values.

2. For every relation with **Filter, Relation CC and has borrowed projected attributes**, the fabrication process is as per the following steps:

In Q_4, Customer and Customer_address relations fall in this category.

(a) Create a template **Filter satisfying row.**

(b) Create $\mathbf{F_{Card}}$ **Projection satisfied rows** with **unique values for borrowed projected attributes** from the template row obtained from Step 1. Each of these rows is added to the relation summary with cardinality 1. This will satisfy the Filter CC, while the borrowed projected attributes would be unique in each row.

(c) Create a **Filter non-satisfying row**, and add it to the database summary with cardinality $\mathbf{R_{Card}} - \mathbf{F_{Card}}$. This will satisfy Relation CC.

(d) For each row from Steps (b), the foreign key attributes are fabricated the same way as in **Step 1(e)**.

(e) For each row from Step (c), the foreign key attributes are fabricated as the same as in **Step 1(g)**.

3. For every relation with **Projection and Relation CC**, the fabrication process is done in the same way as discussed in **Step 1 of subsection 4.3.2.**
In Q_4, if Web_sales relation had no filters, it would have fallen in this category.

4. For every relation with **Relation CC and borrowed projected attributes**, the fabrication process is done as the same way as discussed in **Step 2 of subsection 4.3.2.**
In Q_4, if Customer and Customer_address relation had no filters, it would have fallen in this category.

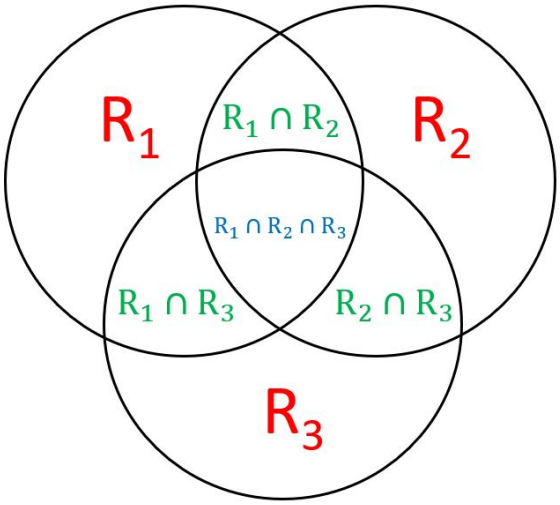
5. For every relation with only **Filter and Relation CC**, the fabrication process is done as the same way as discussed in **Step 1 of subsection 4.3.1**

6. For every relation with only **Relation CC**, the fabrication process is done as the same way as discussed in **Step 2 of subsection 4.3.1**.

4.4 Simplicity in Handling LIKE Operator

As we have only a single query into consideration in this context, we can make certain optimization to the general case of handling the LIKE operator discussed in Chapter 3. For an AQP from a single query, there can only be a maximum of one CC per relation for LIKE operators. In that one CC, we can still have one or multiple LIKE predicates that are applied. For the case of a single LIKE predicate, we follow the same procedure as discussed in subsection 3.1. However, for multiple LIKE predicates, the same procedure discussed in from subsection 3.3 is followed, but with the following optimization.

Let us consider that we have a CC with three LIKE predicates in conjunction and their regular expressions are R_1, R_2, and R_3. Recall Figure 3.3 from subsection 3.2.1:



Here, we can directly take the representative string corresponding to $R_1 \cap R_2 \cap R_3$, thus completely skipping the computation of disjoint spaces, and proceed with the transformation of like predicate into equality filter predicate.

4.4.1 Like Predicate participating in Projections

In our design, wherever projections were applied, we fabricated unique values in the template rows and added it to the corresponding relation summary. For queries wherein the attributes on which the LIKE operator is applied, also feature as a projected attribute, we follow an *indirect* approach. First, the LIKE predicate is converted into an equality filter predicate using the same procedure discussed in subsection 3.1. But a slight modification is done, wherein, instead of generating just one representative string, we generate $\mathbf{P_{card}}$ number of unique representative strings from the corresponding regular expression, and frame them all in disjunction, and form our equality filter predicate.

As an example, let us consider the following Projection CC and a CC with LIKE predicate applied on *Ship_mode* relation from the TPC-DS benchmark, both with sm_code as the common participating attribute.

$$|\pi_{sm\_code}\ (\text{Ship\_mode})| = 4$$
$$|\sigma_{(sm\_code \sim\sim '\%C\%')}\ (\text{Ship\_mode})| = 15$$

We need 4 distinct representative strings from '%C%', and say we generate 'CA', 'CB', 'CC', 'CD'. Then the LIKE predicate CC is transformed as follows :

$$|\sigma_{(sm\_code \sim\sim '\%C\%')}\ (\text{Ship\_mode})| = 15$$

$$\rightarrow |\sigma_{((sm\_code = 'CA') \vee (sm\_code = 'CB') \vee (sm\_code = 'CC') \vee (sm\_code = 'CD'))} \textbf{(Ship\_mode)}| = \textbf{15}$$

Now, with the Projection CC and this transformed filter CC, the problem reduces to handling Projection and Filter CCs together, which can be carried out by the procedures discussed earlier in this chapter.

4.5 Dynamic Tuple Generation

We built relation summaries on a per-query basis, and so the dynamic generation process becomes a function of the query chosen as well. Each time a query is invoked, the relation summaries corresponding to that query are chosen to perform dynamic generation. Now we look into how the actual dynamic generation procedure is carried out.

From the individual relation summaries we built in the previous sections, we saw that each row in the summary had a list of value combinations and an associated *cardinality* entry. The dynamic generation process involves generating each relation from the database summary on-the-fly as follows. We consider the primary key values to be the row numbers of the relation. Therefore, to get the rth tuple of a relation R, the primary key is chosen as r, and the rest of the attributes come from the relation summary. We iterate over the rows of relation summary and take the cumulative sum of the cardinality entries until the sum exceeds r. Say the summation crosses the value r in jth row of the relation summary. The rest of the values of the rth tuple are precisely the same as those present in the jth row of the relation summary. For each attribute for which we have metadata statistics obtained from the client, we take care to initialize a value from the domain captured in those statistics, rather than picking up any random value.

4.6 Managing Scalability

With the discussion of projection and like inclusive data generation procedure done in the previous sections, we finally look into how we can extend these to manage and handle any number of queries in the training workload.

In the LP formulation phase of Hydra, there are a set of linear equations formed and is solved by using Z3 Solver. When a workload of training queries are processed at once, the number of variables would be large, and so the inherent computational limitation of the solver prevented us from extending the workload to a very large number of queries.

On contrast, in this chapter, we have discussed an extreme case where we worked at the level of individual queries instead of the whole workload at once. Note that for an AQP from a single query, all the corresponding CCs are sub-trees, and hence they have a superset-subset

relationship. This gets reflected in the equations that are formed as well. As an example, let us recall the Filter and Relation CCs of Q_2 corresponding to Customer and Customer_address relation.

$$|\sigma_{c\_birth\_year>50 \wedge c\_birth\_month\in(6,5,12,4,3,7)}\ (\text{Customer})| = 32765$$
$$|\text{Customer}| = 100000$$
$$|\sigma_{ca\_state\in('TN','IL','GA','MO','CO','OH','NM')}\ (\text{Customer\_address})| = 9924$$
$$|\text{Customer\_address}| = 50000$$

If we frame equations for the above CCs, using Hydra's LP formulation module, then we end up with the equations as shown in Table 4.1.

Relation	Equations	Solution
Customer	$x_1 = 32765$	$x_1 = 32765$
	$x_1 + x_2 = 100000$	$x_2 = 67235$
Customer_Address	$x_1 = 9924$	$x_1 = 9924$
	$x_1 + x_2 = 50000$	$x_2 = 40076$

Table 4.1: Nature of equations while handling one query

We can clearly see that the equations can be solved with mere substitution. Hence, while working with a single query at a time, as we propose here, we *need not make use of the solver* at all. Hence, we can bypass the inherent scalability limitation of the solver and manage to handle any number of queries in the training workload.

In fact, it is precisely this substitution strategy we *emulate* in our relation summary generation process discussed in the previous subsections of this chapter, when we directly assigned the cardinalities using $\mathbf{P_{card}}$, $\mathbf{F_{card}}$ and $\mathbf{R_{card}}$, as the Projection, Filter and Relation CCs all follow a hierarchial superset-subset relationship, as they are derived from the same AQP.

Chapter 5

Robustness to Test Query Workloads

In this chapter, we take a detailed look into the general framework of HF-Hydra that enables us to achieve robustness to test query workloads. In this setup, we utilize all the queries from the training workload, to create a unified database summary. First, we start with the discussion of modeling metadata statistics into CCs. Then we talk about how the metadata CCs are made consistent with the CCs from the AQP. Later, we see the process of generating equivalent SQL queries for regions and extracting the optimizer's cardinality estimate and using it to obtain a better inter-region distribution from the solver. Subsequently, we see the summary generation process and how the intra-region distribution of tuples is done in HF-Hydra, leveraging the database optimizer's model to create more diversity in the data.

5.1 Modeling Metadata Statistics

Most database engines maintain metadata statistics which can be easily modeled as CCs. Here, we discuss some of the Postgres engine's stats and how they are modeled into CCs in HF-Hydra. This technique can be extended easily for other database engines as well. To distinguish these CCs from AQP CCs, we hereafter refer to them as metadata CCs.

5.1.1 Postgres Metadata Statistics

The metadata information in Postgres is stored in the *pg_statistic* system catalog. Entries in pg_statistic are always approximate even when freshly updated. Rather than looking at pg_statistic directly, it's better to look at its view *pg_stats* when examining the statistics manually. The pg_stats view is designed to be more easily readable and is restricted to show only rows about tables that the current user can read. We source information from three metadata statistics, specifically MCV, MCF, and Histogram bounds from pg_stats. We look into each one of them in detail in the following subsections.

5.1.2 MCVs and MCFs

Postgres maintains two lists for most attributes in a relation, namely, Most Common Values (MCVs) and Most Common Frequencies (MCFs). Let us see the definitions and an example for each from the Postgres documentation to understand better.

Most Common Values: A list of the most common values in the column. (Null if no values seem to be more common than any others.)

Most Common Frequencies: A list of the frequencies of the most common values, i.e., number of occurrences of each divided by the total number of rows. (Null when MCVs is.)

Example:

For an attribute A of a relation R, the frequency of an element stored at position i in the MCVs list will be stored at the matching position i in the MCFs array.

SELECT most_common_vals, most_common_freqs FROM pg_stats WHERE tablename = 'tenk1' AND attname = 'stringu1';	
most_common_vals	{FDAAAA, NHAAAA, ATAAAA, BGAAAA, EBAAAA, MOAAAA, NDAAAA, OWAAAA, BHAAAA, BJAAAA}
most_common_freqs	{0.00333333, 0.00333333, 0.003, 0.003, 0.003, 0.003, 0.003, 0.003, 0.00266667, 0.00266667}

In the example, the frequencies corresponding to strings – 'ATAAAA' is 0.003 and 'BHAAAA' is 0.00266667. We can express this information in the form of CCs as follows:

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

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

5.1.3 Histogram bounds

Postgres 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. An example from the Postgres documentation is given below.

Example :

SELECT histogram_bounds FROM pg_stats WHERE tablename = 'tenk1' AND attname = 'unique1';
{1, 970, 1943, 2958, 3971, 5069, 6028, 7007, 7919, 8982, 9995}

In this example, the attribute 'unique1' has 10 equi-depth histogram buckets.
We can express histogram bounds information in the form of CCs as follows:

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

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

5.2 Refined Partitioning using Metadata

We saw that Hydra uses all the AQP CCs to do region partitioning in subsection 1.1.3. In HF-Hydra, we additionally use the metadata CCs to do a *refined-region-partitioning* of the data space. The result of this activity is that we get regions at a much finer granularity. Note that we use the metadata CCs only to do refined region partitioning, but not include them in the SMT formulation. Instead, we further process these metadata CCs and eventually formulate an LP, and the details are discussed in the upcoming section.

An example representation of refined region partitioning is given in Figure 5.1.

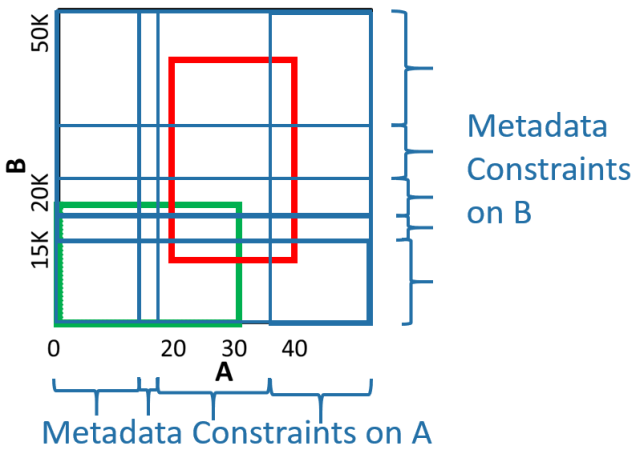


Figure 5.1: Refined partitioning in HF-Hydra using Metadata CCs

5.3 Architecture of HF-Hydra

The end-to-end pipeline of the general setting of HF-Hydra's data generation to provide robustness to test queries is shown in Figure 5.2. The client AQP and metadata CCs are given as input to *LP Formulator*. Using the inputs, the module constructs a refined partitioning of the data space. Further, an LP is constructed by adding an objective function to pick a *desirable* feasible solution. From the LP solution, which is computed using the popular Z3 solver [5, 15], the *Summary Generator* produces a richer database summary. We discuss the different modules in detail in the upcoming sections.

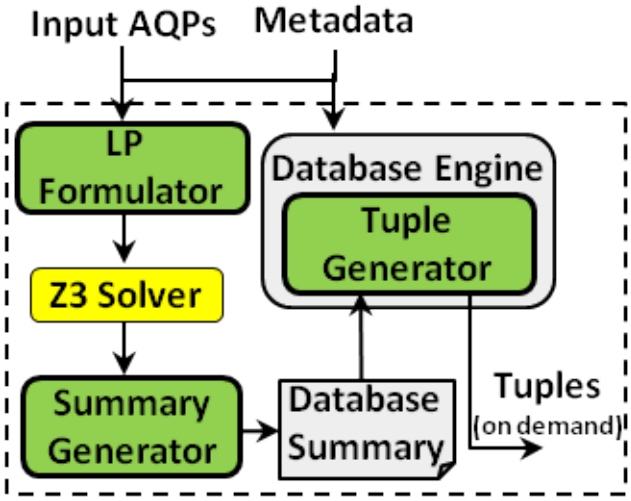


Figure 5.2: HF-Hydra Pipeline

5.4 LP Formulator

Here we discuss our LP formulator module to get a better inter-region tuple cardinality distribution. Note that the original database also satisfies the SMT problem that Hydra constructs, but there can be several assignments of cardinalities to the regions such that all of them satisfy the constraints. For example, if we recall the SMT equations constraints corresponding to the example shown in Figure 1.4, we have

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

Both the following solutions satisfy the constraints:

$$\textbf{Solution 1:} \quad x_1 = 250, x_2 = 0, x_3 = 150, x_4 = 300$$
$$\textbf{Solution 2:} \quad x_1 = 100, x_2 = 150, x_3 = 0, x_4 = 450$$

Hydra does not prefer any particular solution over the other. Another point to note about the equations that get formed in Hydra is that they are highly under-determined, i.e., there are more number of variables that participate in the equations than the number of equations that are formulated. The Z3 solver assigns non-zero cardinality to very few regions. This is because of the use of the simplex algorithm internally, which seeks a basic feasible solution, leading to a sparse solution. Further, not only is the solution sparse, but also, the assigned cardinalities may be very different from the original database since no explicit efforts are made to bridge the gap. For example, the two solutions illustrated above are both similar from the sparsity point of view but the values themselves are very different. It is very easily possible that one of the two is picked by the solver, and the other is actually the desired solution. Therefore, the *distribution* of tuples among the various regions in the original and synthetic databases can be very different, leading to the generation of a drastically different synthetic database. Hence, volumetric similarity for test queries can incur enormous errors.

In HF-Hydra, we propose a two-step process to enhance the inter-region distribution from our LP formulation:

1. Correct the inconsistencies in the values from the metadata CC with respect to the values from AQP CCs.

2. Frame an LP and obtain cardinality assignments that are compliant with the database optimizer's cardinality estimate for individual regions.

We discuss each step in detail in the following subsections.

5.4.1 Metadata Inconsistency Correction

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 be used with the AQP CCs while framing the LP constraints, we need to resolve the inconsistencies in the metadata CCs. For this purpose, we frame an LP with the AQP CCs, and add the metadata CCs in an optimization function that tries to satisfy the metadata CCs with minimal error.

The complete algorithm is as follows:

1. Run region partitioning (as discussed in Section 1.1.3) using all the CCs, i.e., CCs from both AQPs and metadata.

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

3. The CCs from metadata are modeled in the optimization function that minimizes the L1 norm of the distance between the output cardinality from metadata CCs and the cardinality from the sum of variables that represent the CCs. If there are n regions (variables) together obtained from m metadata CCs and q AQP CCs, then the LP formulation is as shown in Figure 5.3. Here, I_{ij} is an indicator variable, which takes value one if region i satisfies the filter predicate in the jth metadata CC, and takes value zero otherwise. Further, $C_1, C_2, \ldots, C_q$ are the q LP constraints corresponding to the q AQP CCs.

$$\text{minimize} \sum_{j=1}^{m} \epsilon_j \text{ subject to:}$$

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

Figure 5.3: LP Formulation for Metadata Inconsistency Correction

An example representation of this LP formulation is shown in Figure 5.4. Here, the highlighted portion shows a metadata CC from a histogram bucket along the attribute A, and the six refined regions that satisfy the filter predicate corresponding to this metadata CC.

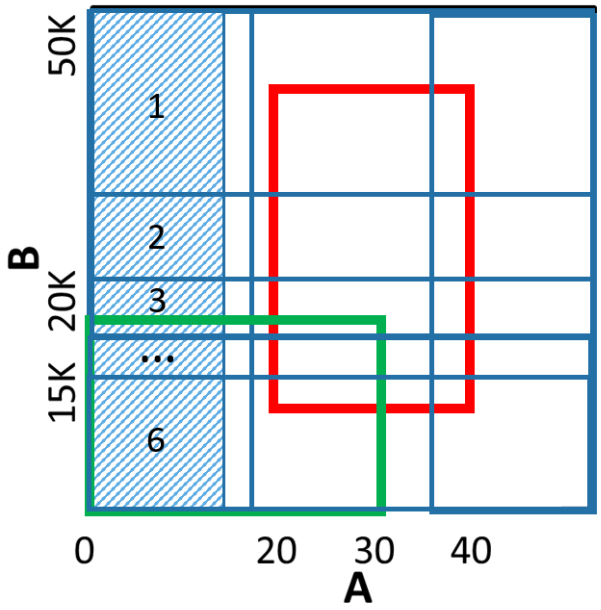


Figure 5.4: Metadata Inconsistency Correction in HF-Hydra

From the solution of the LP obtained from the solver, we get error values for each ϵ used in the optimization function. This error can be now adjusted against the corresponding metadata CC variable. Once this error correction is done, the metadata CCs will become consistent with the AQP CCs, and we can directly use both types of CCs as explicit constraints. These *error-corrected* metadata CCs are used in the second step of the process, as discussed in the upcoming subsection.

5.4.2 Optimizer Cardinality Estimate Compliance

While adding the metadata CCs explicitly with the AQP CCs can help to obtain a better LP solution, it can still suffer from inconsistent inter-region distribution from the solver – in fact, we may still get a sparse solution. This is because there is no explicit constraint that works on the individual regions to get a less-sparse cardinality assignment.

In this next step, we try to overcome this lacuna by trying to obtain an LP solution with explicit constraints on the level of individual regions. To begin with, the estimates for each region's cardinality are obtained from the database engine itself. Once the estimates are obtained, we find an LP solution that is close to these estimates while satisfying all the explicit CCs coming from the AQPs and *error-corrected* metadata CCs. Note that, similar to the previous subsection, the cardinality estimates cannot be expected to be accurate, and so the values may be inconsistent with the explicitly added CCs. Hence, we add the optimizer cardinality

estimate compliance constraint as an optimization function.

The complete algorithm is as follows:

1. Run region partitioning (as discussed in Section 1.1.3) using both AQP CCs and *error-corrected* metadata CCs.

2. For each region obtained from (1), we construct an SQL query equivalent. Any query that can capture the region precisely is acceptable. The query generation process is described in detail in section 5.4.3.

   As an example, the queries for the four regions from Figure 1.4 is as follows:

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

3. Once the region-based SQL queries are obtained, their estimated cardinalities are obtained from the optimizer. This is done by dumping the metadata on a dataless database using the CoDD metadata processing tool [1] and then obtaining the compile-time plan (for example, using EXPLAIN $\langle$query$\rangle$ command for the case of Postgres). Let the estimated cardinality for the n regions be $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n$.

4. We construct an LP that tries to give a feasible solution, i.e., that satisfies all AQP and error-corrected metadata CCs, while minimizing the L1 distance of each region from its estimated cardinality as obtained from the previous step. The LP formulation that is formulated is shown in Figure 5.5.

$$\text{minimize} \sum_{i=1}^{n} \epsilon_i \text{ subject to:}$$

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. $x_i, \epsilon_i \geq 0 \quad \forall i \in [n]$

Figure 5.5: LP Formulation for Optimizer Cardinality Estimate Compliance

An example representation of this LP formulation is shown in Figure 5.6. Here, the highlighted portion shows an example refined region. The cardinality estimate for such refined regions that take part in the LP formulation are obtained and used in the optimization function.

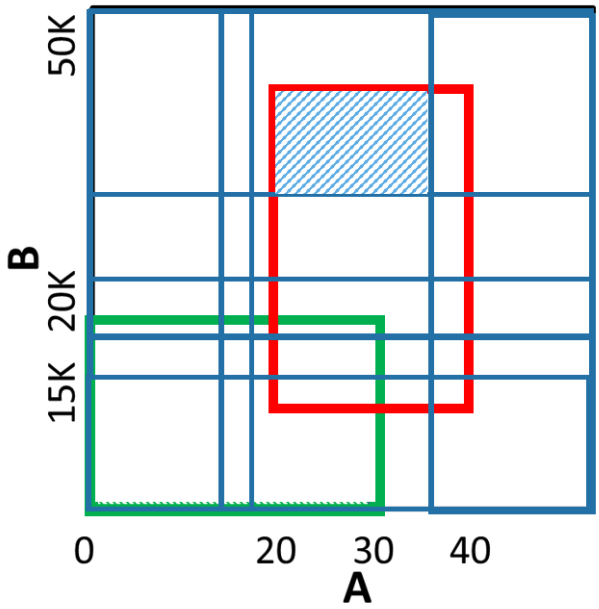


Figure 5.6: Optimizer Cardinality Estimate Compliance in HF-Hydra

Our choice of minimizing the L1 norm in these steps is reasonable because from the 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.

Given the Metadata Inconsistency Correction and Optimizer Cardinality Estimate Compliance steps, the following aspects are to be noted:

1. Metadata Inconsistency Correction step is comparatively simpler as it adds fewer constraints and hence is computationally more efficient. Optimizer Cardinality Estimate Compliance adds more constraints, as it works on every region. 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 Optimizer Cardinality Estimate Compliance 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 Metadata Inconsistency Correction is usually more reliable as they are derived as it is from the metadata.

5.4.3 SQL Query Generation for Regions

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

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

A region in its most general form can be represented as a collection of multidimensional arrays, where each multi-d array has an array of intervals for its constituent dimensions (relation attributes). We term each dimension as a **Bucket**, and each collection of such Buckets as a **BucketStructure**. One or more such BucketStructures constitutes the region. For example, Figure 5.7 shows the structure of a region. It can be expressed verbally as:

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

After refined-region-partitioning, all the intervals for each attribute corresponding to a region are captured in the corresponding buckets and bucketstructures of that region. Within each bucket, the split points may or may not be continuous. The convention in HF-Hydra is that every interval is in the form of $[a, b)$ i.e., the included values are from a to $b - 1$.

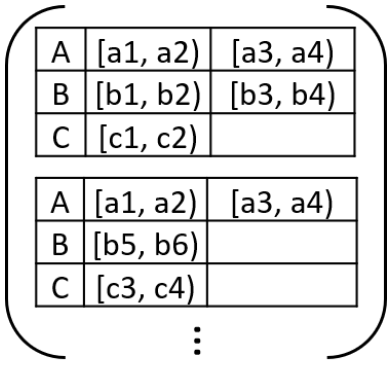


Figure 5.7: Region Structure

The filter predicates for our SQL query is generated as follows:

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

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

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

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

The final SQL query is generated as:

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

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

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

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

The filter predicates representing each bucket are:

BucketStructure 1	
Column X.A	(X.A $\geq a_1$ AND X.A $< a_5$)
Column Y.B	((Y.B $< b_1$) AND (Y.B $\geq b_3$ AND Y.B $< b_5$))
Column Z.C	(Z.C $\geq c_1$ AND Z.C $< c_3$)

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

The filter predicates representing each bucketstructure are:

BucketStructure 1
((X.A $\geq a_1$ AND X.A $< a_5$) **AND** ((Y.B $< b_1$) AND (Y.B $\geq b_3$ AND Y.B $< b_5$)) **AND** (Z.C $\geq c_1$ AND Z.C $< c_3$))

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

The filter predicate representing the whole region is:

Region R
((X.A $\geq a_1$ AND X.A $< a_5$) AND ((Y.B $< b_1$) AND (Y.B $\geq b_3$ AND Y.B $< b_5$)) AND (Z.C $\geq c_1$ AND Z.C $< c_3$)) **OR** ((X.A $\geq a_6$ AND X.A $< a_9$) AND (X.A $\geq a_{11}$ AND X.A $< a_{13}$)) AND (Y.B $\geq b_6$ AND Y.B $< b_7$) AND ((Z.C $\geq c_5$ AND Z.C $< c_8$) AND (Z.C $\geq c_{10}$))

The final SQL query equivalent generated for region R is:

SELECT * FROM X,Y,Z
WHERE (X.PK = Y.FK) AND (Y.PK = Z.FK)
AND
(((X.A $\geq a_1$ AND X.A $< a_5$) AND
((Y.B $< b_1$) AND (Y.B $\geq b_3$ AND Y.B $< b_5$)) AND
(Z.C $\geq c_1$ AND Z.C $< c_3$))
OR
(((X.A $\geq a_6$ AND X.A $< a_9$) AND (X.A $\geq a_{11}$ AND X.A $< a_{13}$)) AND
(Y.B $\geq b_6$ AND Y.B $< b_7$) AND
((Z.C $\geq c_5$ AND Z.C $< c_8$) AND (Z.C $\geq c_{10}$))));

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

5.5 Summary Generator

Within a region that gets a non-zero cardinality assignment, Hydra generates a single unique tuple. As a result, it leads to inconsistent *intra-region distribution*. That is, even if the inter-region distribution was matched between the original and synthetic databases, the distribution of tuples *within* a region could vary enormously. This is again not surprising since no efforts have been put to match the intra-region distribution. This single point instantiation per region creates a situation wherein only very few domain points are chosen to generate the entire database, and makes the data look very unrealistic. This heavily affects the accuracy of test queries. In HF-Hydra, we instead generate data within the region in compliance with the distribution of the database engine's estimation module. For instance, the Postgres engine assumes uniform distribution among the metadata stats. Hence in our implementation, we also resort to uniform distribution within each region. For other engines, this step can be modified accordingly to comply with other distribution models that is followed by that engine.

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

5.5.1 Merging Sub-Views

Say two sub-views R and S are to be merged, and they have a common set of attributes $\mathbb{A}$. If $\mathbb{A}$ is empty then we can directly take the *cross* of the regions in R and S to obtain the regions in the view space. But when $\mathbb{A}$ is non-empty, the merging of regions happen among *compatible pair*. A region from R is compatible with a region form S if the two have an identical projection along $\mathbb{A}$. Several regions in R and S can have identical projection along $\mathbb{A}$. Therefore, we define compatibility at a set level. That is, a compatible pair is a set of regions from R and S such that all these regions have identical projections along $\mathbb{A}$. For such a compatible pair, let $r_1, r_2, \ldots, r_m$ be the regions from R and $s_1, s_2, \ldots, s_n$ be the regions from S. In HF-Hydra we merge each region r_i with each region s_j from a compatible pair and assign it a cardinality $\frac{|r_i||s_j|}{\sum_{i\in[m]}|r_i|}$. Note that $\sum_{i\in[m]}|r_i| = \sum_{j\in[n]}|s_j|$, which is ensured by consistency constraints in the LP. Hence in each pair, $m \times n$ regions are constructed. Merging of two regions can be thought of as taking a join of the two regions along $\mathbb{A}$. A sample sub-view merging is shown in Figure 5.8 where two sub-views (A, B) and (A, C) have four and three regions, respectively. Further, there are two

compatible pairs that finally lead to six regions after merging.

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

×

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

=

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

Figure 5.8: Sub-view Merging

5.5.2 Ensuring Referential Integrity

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

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

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

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

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

As we saw that the algorithm takes projections of regions along borrowed set of dimensions. Since the regions neither have to be hyper rectangles nor have to be continuous, they may not

be *symmetric* along the borrowed attribute(s), further adding to the complexity of ensuring consistency. If the set of the attributes in a view V is A, and the set of attributes that it borrows are B, then a region r in V is symmetric along B iff:

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

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

5.5.3 Parallelization in Ensuring Referential Integrity process

The merge process described in subsection 5.5.1 can result in a lot of regions, and so the intersection process described in subsection 5.5.2 can take a lot of time to compute. Hence, in HF-Hydra, we deploy parallelization to perform intersections efficiently and are discussed in this subsection.

Recall that a region is a collection of multi-dimensional arrays called Bucketstructure, wherein the information about individual attributes that make up the region are captured. In Step 1 and 2 of sub-section 5.5.2, to compute the intersection of F with D, we need to iterate for every bucketstructure within each region f of F and intersect with every bucketstructure in each region d of D. Note that, within a region, a whole bucketstructure is unique, but individual buckets within them can be shared across bucketstructures, and even with other regions. Hence while projecting a region f of F along the attributes borrowed from D, we may end up having bucketstructures that are duplicates of each other. So, firstly, we make sure that all these duplicates are removed and so that the projected region has only unique bucketstructures within them. This reduces the total number of intersections to be computed.

Secondly, regions are disjoint, and so there is no dependency between regions $f_1, f_2, \ldots, f_n$ of F, and so, each of their intersections with regions $d_1, d_2, \ldots, d_n$ of D can be computed parallelly. For each pair of intersections, from regions of F and D to be made, we can create a new independent thread and perform the intersection. But this causes severe overheads and delays in thread creation, resource allocation, and thread deletion. Hence, we follow a different approach to parallelization as follows:

1. The maximum number of parallel threads the underlying system can support is determined, and let's call it T_{max}. This depends on the number of processors available in the system.

2. A fixed thread pool of T_{max} threads are created, and each pair of intersection to be computed is given as a list of jobs to the pool. In the case of a fixed thread pool, each

thread takes up a job and runs it to completion. If there are more number of jobs than the number of threads, then they are all placed in a waiting queue, and is dispatched to threads one after the other. This way, a thread pool reuses previously created threads to execute all the jobs and offers a solution to the above-mentioned thread overheads.

3. Although intersection for each pair of regions can be performed independently, the result of the intersections, for those pairs of regions which did have a non-zero intersection, are accumulated in one data structure per view. In order to avoid any race conditions, we make use of concurrent data structures for storing the results of the computation.

5.5.4 Generating Relation Summary

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

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

Recall that a region in its most general form can be represented as a collection of multi-dimensional arrays, where each multi-d array has an array of intervals for its constituent dimensions (relation attributes). We split the region into sub-regions, where each sub-region is a single multi-d array. Further, we divide the parent region's cardinality among the sub-regions in the ratio of their volumes. In this way, summary for each relation collectively gives the database summary. A sample database summary produced by HF-Hydra is shown in Table 5.1 ((analogous to Figure 1.5). As a sample, the second row in summary for table T means that the last 603 rows in the table have PK column value 903 to 1505, while the value for column C is the range $[0, 2)$ or $[3, 10)$.

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

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

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

Table 5.1: HF-Hydra Example Database Summary

5.5.5 Comparison with Hydra

Hydra's strategy for merging two sub-views is different. For a compatible pair where we construct $m \times n$ regions (as described in section 5.5.1), Hydra may generate as few as $max(m, n)$ regions. Therefore, it leads to several "holes" (zero-cardinality regions) in the view, further leading to poor generalizations to unseen queries.

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

5.6 Tuple Generator

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

5.6.1 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.

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

Chapter 6

Melding Robustness with Scalability

In the framework discussed for the general case in the previous segment, with increase in the workload constraints, we may run into scalability issues, stemming from the inherent computational capacity limitation of the solver. The load on the solver is even more severe with the use of optimization functions in the LP formulation, which further hurts the handling capacity. Hence, to sidestep this challenge, in this chapter, we propose a *workload-division* strategy to produce a few sub-workloads from $W_{Extended}$, the extended workload containing large number of cardinality constraints, and eventually construct an individual database summary for each of them using the same procedure discussed in the previous chapter.

When a testing query is fired, we then deterministically forward it to be executed on the database generated from one of these summaries, which is expected to produce the maximum volumetric similarity.

We start the discussion with an overview of our workload division strategy. Next, we talk about how attribute intervals for training domain space are collected. Subsequently, we talk about our discovery process to figure out the number of sub-workloads that would be required to bypass the solver's limitation. Finally, we look into how the training queries are mapped into individual sub-workloads, and how a given test query is deterministically forwarded.

6.1 Workload Division Strategy

The goal of our workload-division strategy is to create sub-workloads from $W_{Extended}$, such that each individual sub-workload is within the handling capacity of the solver. We do this by dividing the domain space that is formed across all attributes from the training workload into sub-domains. Each sub-domain of an attribute is then assigned to a sub-workload, and it is ensured that all sub-workloads *collectively* cover the training workload domain space.

Let us consider that we divide the domain space across all attributes into four sub-domains giving rise to four sub-workloads. For two attributes A and B, a pictorial representation of our proposed workload-division strategy is given in Figure 6.1. The first sub-workload will focus on the space that comprises the first sub-domain across all attributes as shown in Figure 6.1a, the second sub-workload will focus on the space that comprises the second sub-domain across all attributes as shown in Figure 6.1b, and likewise, up till m number of sub-workloads.

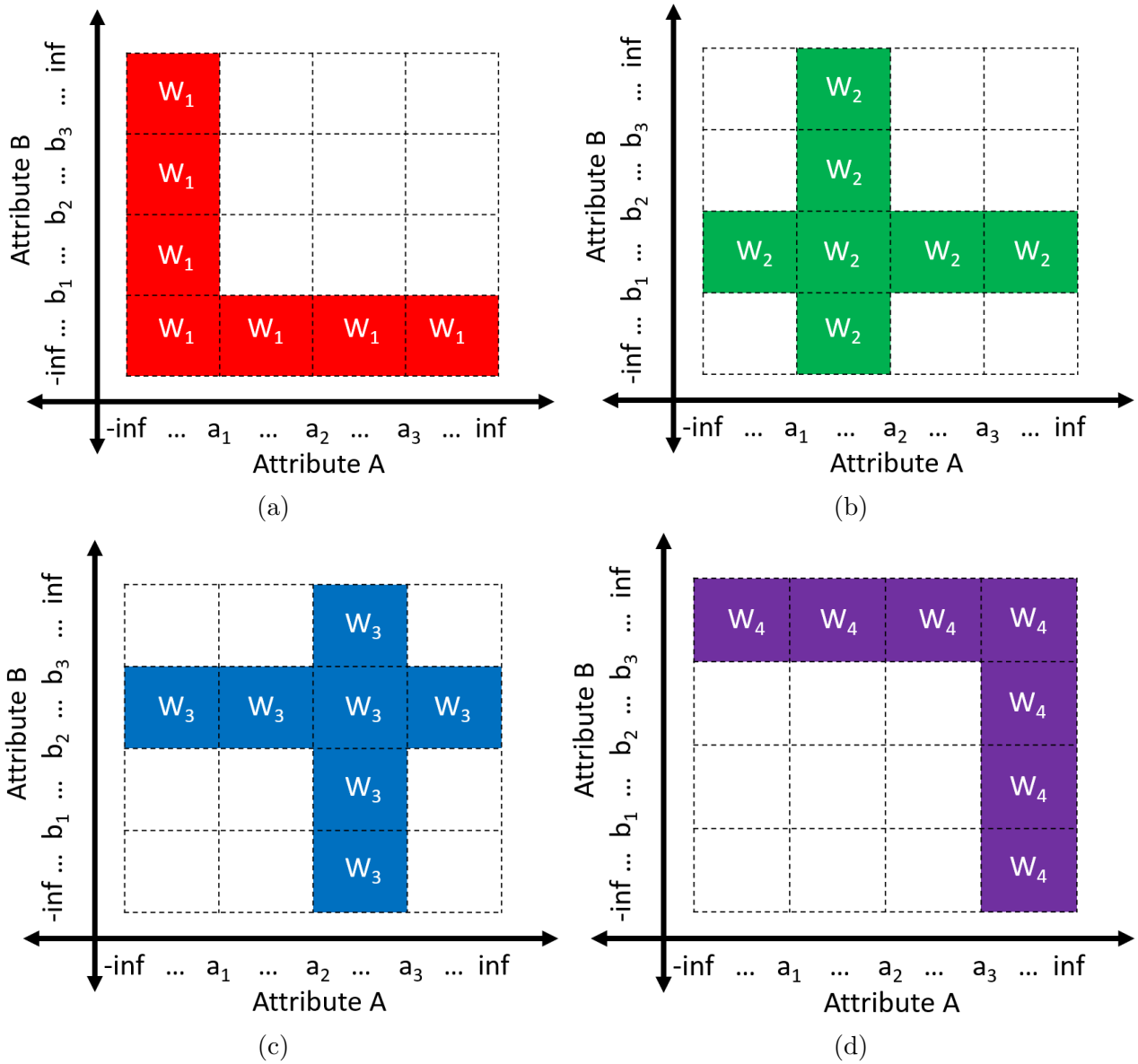


Figure 6.1: Overview of Workload-Division Strategy

A consolidated view of the domain space post the division is shown in Figure 6.2. It is to be noted any other combination in the ordering of assignments among the sub-domains and the sub-workloads is acceptable as well. But care needs to be taken that such an assignment ends up in all the individual sub-workloads collectively covering the entire domain space.

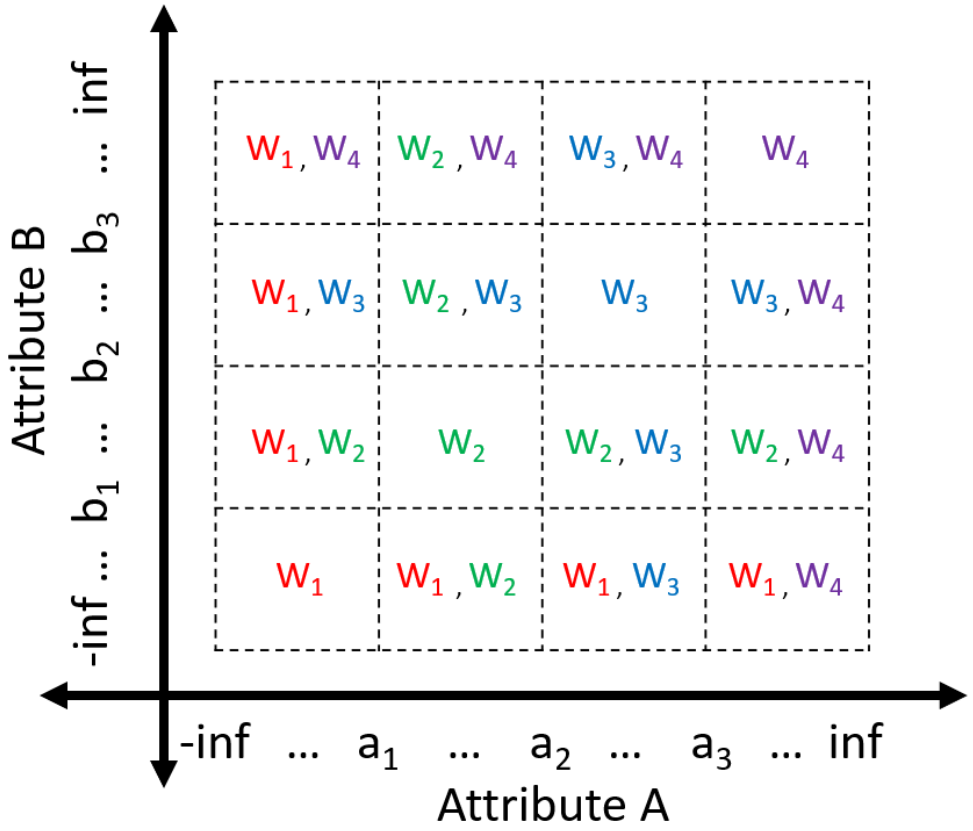


Figure 6.2: Consolidated view of sub-workloads covering the entire domain space

6.2 Collecting Domain of Attributes

To facilitate the workload-division strategy to create sub-workloads, we first collect the domain of each attribute that participate in $W_{Extended}$ using the following procedure:

1. All CCs from AQPs, and all CCs from metadata statistics that feature in $W_{Extended}$ are extracted, and for each CC, we collect all the participating filter predicates. Each filter predicate constitutes of – (a) the attribute on which the filter is applied, (b) the filter operator used and, (c) the filter predicate constant.

2. Among the three, we collect only the attribute name and the filter predicate constant,

and thereby, at the end of this step, we create a list of all attributes along with all the predicate constants that have featured for that attribute in $W_{Extended}$.

3. For each attribute, the accumulated predicates are sorted. This sorted list represents the entire domain interval of each attribute that has been featured in $W_{Extended}$. The same strategy is adopted for attributes of all data types.

The complete collection of all these individual domain intervals constitute the whole domain space of our $W_{Extended}$.

6.3 Determining the Number of Sub-Workloads

The number of sub-workloads, m, is determined by a deterministic discovery process. Recall that the need for dividing $W_{Extended}$ into sub-workloads stems from the inherent scalability limitation of the solver which prevents us from using all the constraints at once in our summary construction process. The solver breaks down when the LP formulation ends up with a complex system of equations with too many variables. Both the nature of the equations, and the number of variables that are formulated affect the handling capacity of the solver, However, in our proposal, we make use of the *heuristic* of the number of variables generated from the constraints in a sub-workload alone to tell whether it is within the handling capacity of the solver. The detailed procedure is as follows:

1. The process starts with producing two sub-workloads from $W_{Extended}$. The domain interval constructed for all attributes as discussed in the previous subsection, is broken into two, and each sub-domain is associated with each sub-workload.

2. Once the sub-domain intervals are finalized for each sub-workload, we iterate over $W_{Extended}$, and for each CC therein, we assign it to an appropriate sub-workload using the *mapping procedure*, which is discussed in the upcoming subsection 6.4.

3. Once the mapping process is complete, each sub-workload is then checked for its *computational viability* in terms of the number of variables in the LP formulation process. The numerical limit for this heuristic is set by the vendor. For Z3 solver that is used in HF-Hydra, we have frequently observed the solver taking enormous amount of time and resources to compute with equations spanning variables upwards of 40,000. It is to be noted that the solver is invoked independently for each relation (view, to be precise in HF-Hydra) to assign cardinalities for regions that are formed post the refined region partitioning for that relation. Hence, this number corresponds to the variables formulated for a single relation, and not across all relations at once.

4. For those sub-workloads that fail the check, the corresponding sub-domain for all attributes is again split into two, creating two sub-sub-domains, thus giving rise to two new sub-sub-workloads. Then the CCs that are featured in the sub-workload are re-assigned by the same mapping procedure into the new sub-sub-workloads.

5. This checking and eventual splitting is repeated recursively for each sub-workload until we reach the stage where all individual sub-workloads are within the computationally viable zone, i.e. all sub-workloads have the number of variables from their LP formulation that is easily manageable by the solver.

An extreme worst case possibility from the above procedure is when the sub-workloads keep getting split into sub-sub-workloads. Here, this procedure will end up in yeilding a single query per sub-workload scenario. At that point, we can take over with the techniques to create relation summaries as discussed in chapter 4. But as long as a sub-workload contains more than one constraint, we resort to creating summary using the general framework from chapter 5.

We talked about dividing the domain of the attribute that features in the training workload in Step 1 of the above procedure. During each bifurcation, we take care to make a *continuous* and *almost equal cardinality assignment* to each sub workload. Specifically, in the sorted list of predicates that we collect for all attributes, we maintain the count for unique predicates, and during the actual bifurcation process, we split the domain into two continuous sub-domains such that the cardinality of filter predicates in each sub-domain is tried to be equal. The intuition behind doing this can be better understood when we discuss it in subsection 6.4.1.

Another possible corner case is that if there are fewer predicate points that feature in the workload than m, the number of sub-workloads. This is especially possible in the case of categorical domain attributes. In this case, for that attribute alone, we assign the same sub-domain for more than one sub-workload. As long as the entire space is covered, any assignment of sub-domains to sub-workloads is acceptable.

Before discussing the mapping procedure for the assignment of training CCs into appropriate sub-workloads, let us take an example to understand the division of domain of attributes that feature the training workload.

6.3.1 Example for Intervalisation of Attribute Domain

Let us take an example training workload of CCs that are applied on *Customer* relation of the TPC-DS benchmark, on attributes *c_birth_day* and *c_birth_month*.

$$CC_1\text{: } |\sigma_{c\_birth\_day \geq 25 \,\wedge\, c\_birth\_month = 3} \text{ (Customer)}| = 150$$
$$CC_2\text{: } |\sigma_{c\_birth\_month \leq 2} \text{ (Customer)}| = 200$$

$$CC_3\text{: } |\sigma_{c\_birth\_day=28 \wedge c\_birth\_month=1} \text{ (Customer)}| = 100$$
$$CC_4\text{: } |\sigma_{(c\_birth\_day\geq 22 \wedge c\_birth\_day\leq 25) \wedge c\_birth\_month=3} \text{ (Customer)}| = 250$$

The first step in our workload-division strategy is to create a sorted list with all predicate values that represent the domain of the attribute featured in $W_{Extended}$. These predicate values may have appeared with any operator, i.e., with $=, <, >, \leq, \geq$, or LIKE. We can see that *c_birth_day* has predicate values 22, 25, and 28, whereas *c_birth_month* has 1, 2, and 3. Likewise, let us consider that with similar additional CCs from $W_{Extended}$, we continued with the same procedure to collect all the predicates, and the final list of attributes with their sorted domain of predicates looks as illustrated in Table 6.1.

Attribute	Predicates
c_birth_day	20, 22, 22, 22, 22, 25, 25, 25, 25, 28, 28
c_birth_month	1, 2, 2, 3, 3, 4

Table 6.1: Sorted Domain for Attributes from $W_{Extended}$

The two individual lists represent the sorted domain of attributes *c_birth_day* and *c_birth_month* featured in the training workload. In addition to the list, we also maintain the count of each unique predicate that features in the list, as shown in Table 6.2.

c_birth_day		c_birth_month	
Predicate	Count	Predicate	Count
20	1	1	1
22	4	2	2
25	4	3	2
28	2	4	1

Table 6.2: Count of Unique Predicates

Using this information, we can begin the intervalisation of domains of *c_birth_day* and *c_birth_month* into two sub-domains. From the predicate points in Table 6.1 and their counts in Table 6.2, we can see that by doing a continuous-almost-equal-partitioning, we end up with two sub-domain intervals (for the case where $m = 2$) as shown in Table 6.3. Further, by the

same logic, we also illustrate the situation where each of these two sub-domains is further recursively split into two finer sub-sub-domains (for the case when $m = 4$) in Table 6.4.

Attribute	Sub-Domain Intervals	
	W_1	W_2
c_birth_day	(-∞, 25)	[25, ∞)
c_birth_month	(-∞, 3)	[3, ∞)

Table 6.3: Intervalisation of Attribute Domains for $m = 2$

Attribute	Sub-Domain Intervals			
	W_1	W_2	W_3	W_4
c_birth_day	(-∞, 22)	[22, 25)	[25, 28)	[28, ∞)
c_birth_month	(-∞, 2)	[2,3)	[3,4)	[4, ∞)

Table 6.4: Intervalisation of Attribute Domains for $m = 4$

The sub-domain interval information constructed here is stored and is used for both the procedures of mapping of training CCs into appropriate sub-workloads and also during the forwarding of test query to be executed on the database from the summary where it is expected to achieve maximum volumetric similarity. We now look into each of these procedures in detail in the upcoming subsections.

6.4 Mapping of Training CCs

1. For a given CC from $W_{Extended}$, we take into consideration all its participating attributes and extract the predicates.

2. From each participating attribute's sub-domain intervalisation information, we figure out the sub-domain(s) with which the predicate has an intersection. We build an *Allocation table* to store this information. Once an intersection is found to exist, the CC's presence in that sub-domain is marked by denoting a one against the CC in the corresponding sub-workload in the Allocation table.

3. The above step is repeated for all CCs to populate the Allocation table. By the end of this process, a CC that has had an intersection in at least one of its participating attributes

with a particular sub-domain will feature in that corresponding sub-workload. There is a possibility that a particular CC has two or more predicates with intersections in sub-domains corresponding to different sub-workloads. In such a case, that CC becomes part of all the sub-workloads which has its presence. Hence, with this mapping process, we do not do a strict disjoint division of CCs into sub-workloads.

4. After the mapping procedure is complete, each sub-workload of constraints is individually processed as discussed in the previous chapter to do refined region partitioning, and an LP is formulated to note the number of variables that are formed. Then the decision as to whether the splitting can be stopped for this sub-workload is taken accordingly.

6.4.1 Best Effort to Reduce Number of Sub-Workloads

Note that in our attribute intervalisation procedure, we give weightage to the count of the unique predicates so as to do an almost equal partitioning of CCs in the sub-workloads. Instead, if we just make the sub-domain division based on the range of the predicates that are collected, then there are chances that some sub-workload(s) do not get any CCs mapped to it, whereas some other sub-workload(s) which is associated with a heavily featured predicate(s) will get most of the CCs mapped to it. Such sub-workload(s) will be then needed to be recursively split again and again, and so resulting in an unnecessary increase in the total number of sub-workloads. With best effort in our almost equal partitioning approach, we try to load balance the number of CCs that go into every sub-workload, so as to keep the total number of sub-workloads minimal. However, we do not guarantee any minimality with our technique.

6.4.2 Example for Mapping Procedure

To demonstrate the above-mentioned procedure for mapping training CCs into sub-workloads, let us consider the filter predicates of CC_1 till CC_4 from our earlier example:

$$\text{Filter predicate from } CC_1\text{: } \sigma_{c\_birth\_day \geq 25 \,\wedge\, c\_birth\_month=3}$$
$$\text{Filter predicate from } CC_2\text{: } \sigma_{c\_birth\_month \leq 2}$$
$$\text{Filter predicate from } CC_3\text{: } \sigma_{c\_birth\_day=28 \,\wedge\, c\_birth\_month=1}$$
$$\text{Filter predicate from } CC_4\text{: } \sigma_{(c\_birth\_day \geq 22 \,\wedge\, c\_birth\_day \leq 25) \,\wedge\, c\_birth\_month=3}$$

Let us start with splitting this workload of four CCs into two sub-workloads. For CC_1, the participating attributes are *c_birth_day* and *c_birth_month*. For *c_birth_day*, the predicate is $\geq$25, and from our sub-domain interval information for *c_birth_day* from Table 6.3, we can see that it has an intersection with W_2 alone. We mark this presence by putting a one against CC_1

for W_2. For *c_birth_month*, the predicate is =3, and from our sub-domain interval information for *c_birth_month* from Table 6.3, we can see that the intersection is with W_2 alone again. We mark this presence by putting an additional one against CC_1 for W_2.

In CC_2, the participating attribute is *c_birth_month* only. For *c_birth_month*, the predicate is < 2, and from our sub-domain interval information for *c_birth_month* from Table 6.3, we can see that the intersection is with W_1 alone. We mark this presence by putting a one against CC_2 for W_1. Similarly. we fill the allocation table by marking the presence of each filter predicate from CC_3 and CC_4. The Allocation table at the end of this process for the workload is shown in Table 6.5 as follows:

Query	Sub-Workload	
	W_1	W_2
CC_1		1,1
CC_2	1	
CC_3	1	1
CC_4	1	1,1

Table 6.5: Allocation Table for $m = 2$

From the data in Allocation Table 6.5, we can then map the constraints into respective sub-workloads as shown in Table 6.6. Note that a CC is mapped to all the sub-workloads where its presence is marked.

Workload	W_1	W_2
Constraints	CC_2, CC_3, CC_4	CC_1, CC_3, CC_4

Table 6.6: Mapping of CCs for $m = 2$

From the above Table 6.6, we can see that both W_1 and W_2 are allocated with three out of the four CCs, and so when such a scenario happens for an extended workload of queries, both sub-workload may still fail our heuristic check, and so further refinement might be needed. Let us now look into the case of how each of these two sub-workloads is recursively split into two further sub-sub-workloads.

Using the domain interval information from Table 6.4, which corresponds to four sub-workloads, we proceed with the re-mapping. The same procedure is used to re-map the con-

straints from W_1 and W_2 of Table 6.5 into W_1, W_2, and W_3, W_4 of Table 6.7, respectively.

Query	Sub-Workload			
	W_1	W_2	W_3	W_4
CC_1			1,1	1
CC_2	1	1		
CC_3	1			1
CC_4		1	1,1	

Table 6.7: Allocation Table for $m = 4$

From the data in Allocation Table 6.7, we can then map the constraints into respective sub-workloads as shown in Table 6.8.

Workload	W_1	W_2	W_3	W_4
Constraints	CC_2, CC_3	CC_2, CC_4	CC_1, CC_4	CC_1, CC_3

Table 6.8: Mapping of CCs for $m = 4$

6.5 Forwarding Test Query

When a test query is fired, we now have a choice of sub-workloads to which we can forward it to be executed. But the challenge here is to forward it to the database generated from that summary, where it is most likely to achieve *maximum volumetric similarity*. The procedure to forward the test query to one of the m sub-workloads is as follows:

1. The test query is broken down to extract all its participating filter predicates, and using the sub-domain interval information created during the intervalisation exercise from section 6.2, we follow the same procedure as discussed in section 6.4, which was used for training CC mapping, and create a *Match table*, similar to that of the Allocation Table.

2. The key difference in the Match table is that, instead of marking the presence of a filter predicate in a particular sub-workload, we maintain the *actual count* of filter predicates that get mapped into each sub-workload.

3. The sub-workload with the *maximum count* at the end of this exercise is likely to have the maximum intersection with this test query, and so we forward the test query to be executed against the database generated from the summary corresponding to that sub-workload.

4. In case multiple candidate sub-workloads have the same count as the maximum count across all sub-workloads, then we can forward the test query to any of those candidate databases.

An important point to note here is that, while mapping training CCs for individual sub-workloads in section 6.4, we worked at the granularity of CCs. So, it is possible that a set of individual CCs, which were derived from the same AQP of a training query can be mapped to different sub-workloads. While forwarding test query, we use the participating filter predicates to determine the sub-workload with which this test query has maximum intersection, but the forwarding is done at the granularity of the whole of the test query and not the individual components of the test query to different sub-workloads.

6.5.1 Example for Test Query Forwarding

To demonstrate the test query forwarding procedure discussed previously, let us consider that the following queries based on *Customer* relation from the TPC-DS benchmark make up our testing workload.

Q_1: Select * from Customer where c_birth_day $\leq$ 26 and c_birth_month = 2;
Q_2: Select * from Customer where c_birth_day = 24 or c_birth_month = 3;
Q_3: Select * from Customer where c_birth_day $\geq$ 29;
Q_4: Select * from Customer where (c_birth_day $\geq$ 20 and c_birth_day $\leq$ 26) and (c_birth_month = 1 or c_birth_month = 2);

The final Match Table at the end of this forwarding process is shown in Table 6.9. For Q_1, the participating attributes in the filter predicates are *c_birth_day* and *c_birth_month*. For *c_birth_day*, the predicate is $\leq$26, and from our sub-domain interval information for *c_birth_day* from Table 6.4, we can see that it has an intersection with W_1, W_2, W_3. We make the count for Q_1 against W_1, W_2, W_3 as 1. For *c_birth_month*, the predicate is =2, and from our sub-domain interval information for *c_birth_month* from Table 6.4, we can see that the intersection is with W_2 alone. We now increase the count for Q_1 against W_2 as 2. As we can see among the four sub-workloads, test query Q_1 has a maximum count with W_2, and hence Q_1 will be forwarded to be executed against the database generated from the summary produced from W_2.

For Q_2, the participating attributes in the filter predicates are *c_birth_day* and *c_birth_month*. For *c_birth_day*, the predicate is =24, and from our sub-domain interval information for *c_birth_day* from Table 6.4, we can see that it has an intersection with W_2 alone. We make the count for Q_2 against W_2 as 1. For *c_birth_month*, the predicate is =3, and from our sub-domain interval information for *c_birth_month* from Table 6.4, we can see that the intersection is with W_3 alone. We now make the count for Q_2 against W_3 as 1. As we can see among the four sub-workloads, both W_2 and W_3 have equal count, which is the same as maximum, and hence Q_2 can be forwarded to be executed against the database generated from the summaries of either W_2 or W_3.

Likewise, Q_3 is forwarded to the database generated from the summary produced from W_4, and Q_4 can be forwarded to the database generated from the summaries of either W_1 or W_2.

Query	Sub-Workload			
	W_1	W_2	W_3	W_4
Q_1	1	2	1	
Q_2		1	1	
Q_3				1
Q_4	2	2	1	

Table 6.9: Match Table for Test Queries

Chapter 7

Experimental Evaluation

In this chapter, we empirically evaluate the performance of HF-Hydra's suite of solutions as compared to Hydra. We implemented the modules of HF-Hydra entirely in Java, on top of the Hydra codebase, which was sourced from [2].

The performance is evaluated using a 100 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 with Intel Xeon 3.2 GHz 8 core processor, 32 GB memory, running Ubuntu Linux 18.04. Our setup is similar to the one used in [25]. We present the results from different scenarios discussed in chapters 4, 5, 6 in each section.

7.1 Query Workload

7.1.1 Specialized Workload Scenario

We constructed a workload, $W_{Projection}$ consisting of 150 projection-inclusive representative queries based on the TPC-DS benchmark query suite. Although in the specialized workload scenario, we execute one query at a time, to provide an essence of the diversity of the queries chosen for our experiments, we present the following statistics. The distribution of the cardinalities (with the cardinalities measured on a log_{10} scale) in the CCs from all the queries in $W_{Projection}$ is shown in Figure 7.1. The distribution of tuples for the CCs are presented with a separation into three groups – CCs that are pure selection filters, and CCs that involve joins applied on top of filters, and lastly, CCs that involve projection. We can clearly see from the figure that the distribution of the cardinalities in these AQPs covered a wide range, from a few tuples to several millions. Also, the distribution of number of projected attributes from all the queries in $W_{Projection}$ is shown in Figure 7.2. From the figure, we can see that we have good diversity in the number of projected attributes as well.

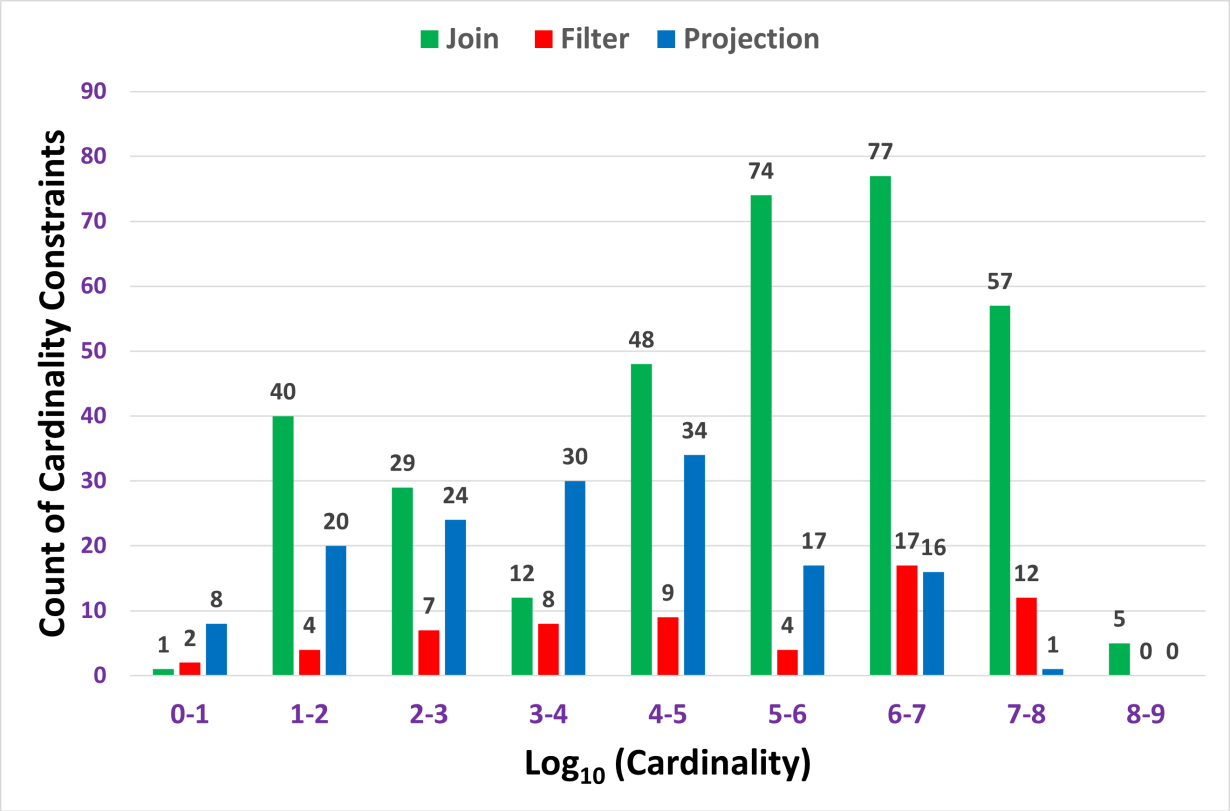


Figure 7.1: Distribution of Cardinality in CCs from AQPs of $W_{Projection}$

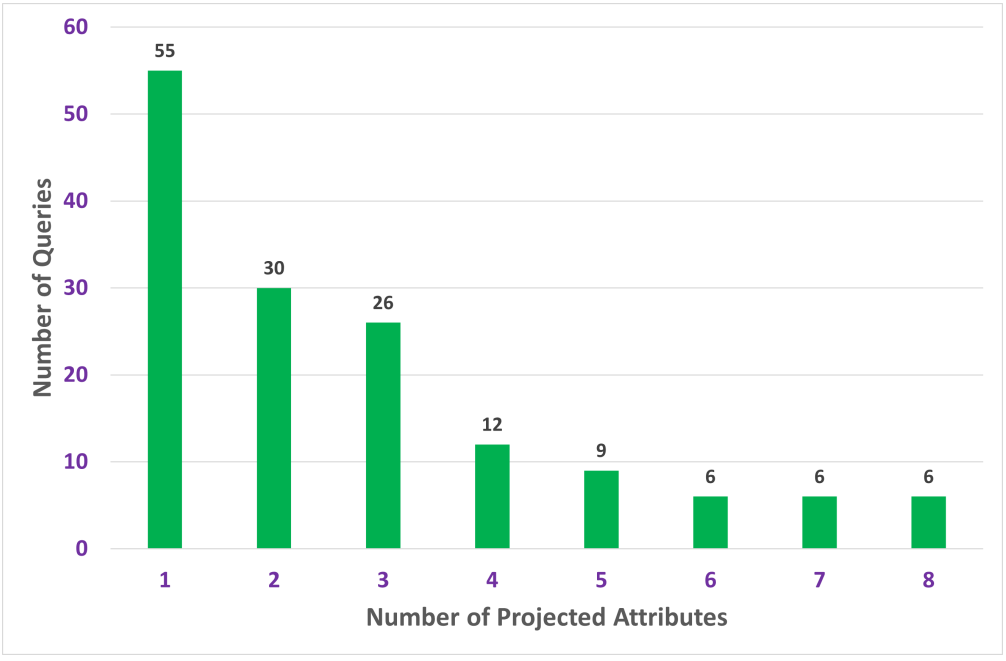


Figure 7.2: Distribution of Number of Projected Attributes in Queries from $W_{Projection}$

7.1.2 Unified Workload Scenario

For the unified workload scenario, where HF-Hydra produces a single unified summary for the entire workload, we constructed two different workloads, W_{Train} and W_{Test} consisting of 176 and 55 representative queries, respectively, based on the TPC-DS benchmark query suite. These were used as our input *training* and unseen *testing* workloads, respectively. The corresponding AQPs from W_{Train} resulted in 419 CCs, while there were 207 such CCs from W_{Test}. The distribution of tuples for the CCs from W_{Train} and W_{Test} are presented with a separation into two groups – pure selection filters on base relations and CCs that involve joins applied on top of filters. The distribution of the cardinalities (with the cardinalities measured on a log_{10} scale) in the CCs from W_{Train} is shown in Figure 7.3, and that of W_{Test} is shown in Figure 7.4.

In addition to that, 1245 CCs were derived from Metadata statistics and made part of our W_{Train} totaling the total number of CCs used for training as 1664. From figures 7.3 and 7.4 we can see again that the distribution of the cardinalities in the AQPs from both W_{Train} and W_{Test}, cover a wide range, from a few tuples to several millions.

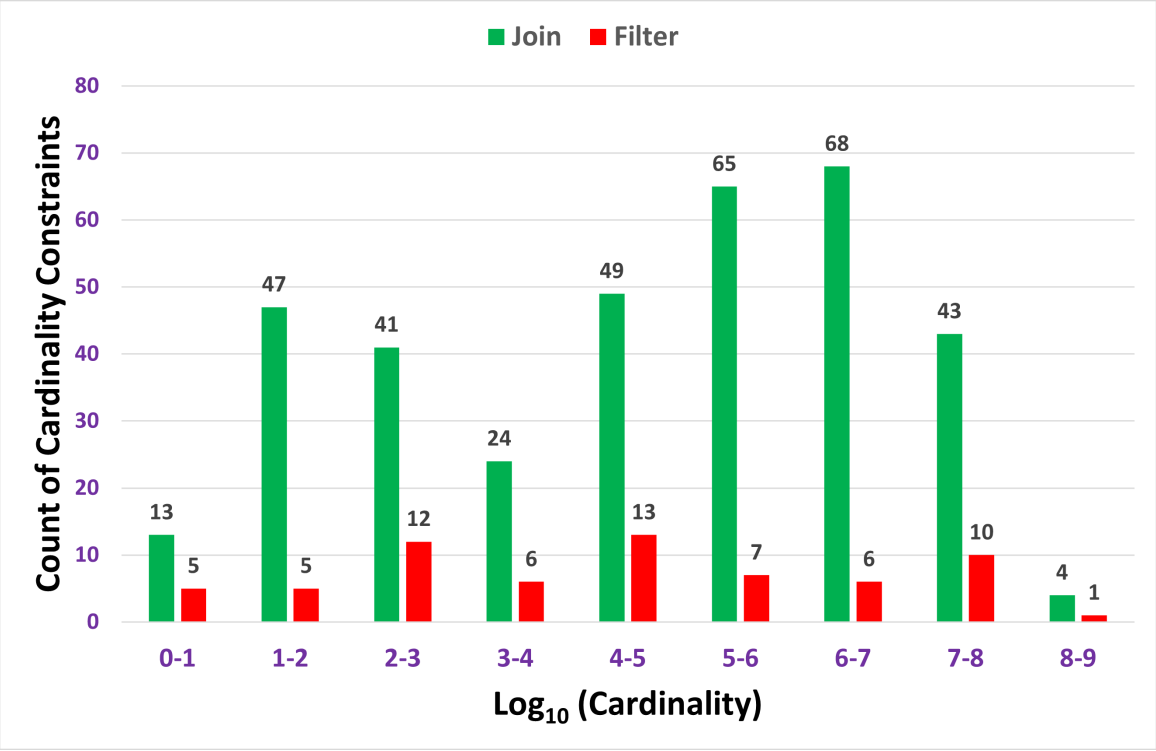


Figure 7.3: Distribution of Cardinality in CCs from AQPs of W_{Train}

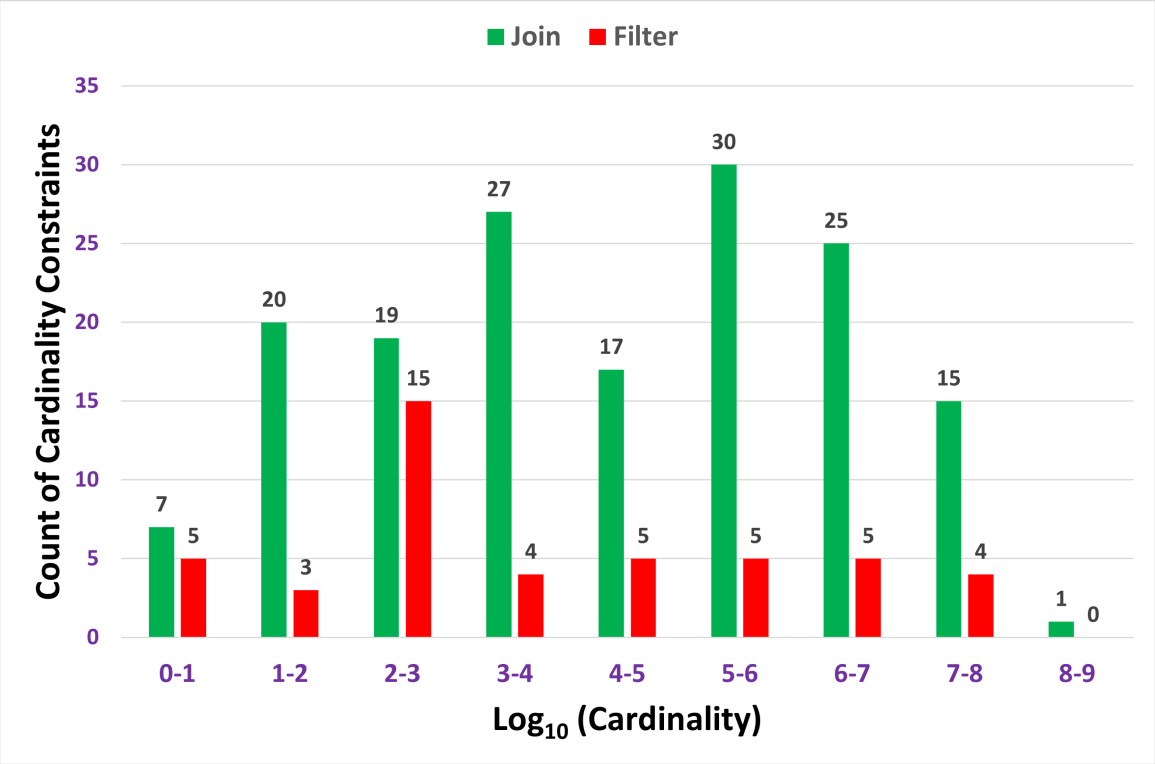


Figure 7.4: Distribution of Cardinality in CCs from AQPs of W_{Test}

7.1.3 Split Workload Scenario

For the split workload scenario, where HF-Hydra produces a different summary for each sub-workload, we extended the above constructed W_{Train} workload with an additional 24 queries to create $W_{Extended}$ of 200 queries in total, and this resulted in a total of 502 CCs. The distribution of the cardinalities (with the cardinalities measured on a log_{10} scale) in the CCs from $W_{Extended}$ is shown in Figure 7.5. In addition to the queries, we derived 4028 CCs from Metadata statistics and was made part of $W_{Extended}$. With this, the total count of CCs that formed our *extended* training workload went up to 4530. This $W_{Extended}$ was found to be not manageable by HF-Hydra to create a unified summary, owing to the inherent limitation of the solver. Thus we invoked our *workload-division* strategy, and four sub-workloads were created. The division of CCs from $W_{Extended}$ into the individual sub-workloads is quantified in Table 7.1. It is to be noted that our workload-division strategy does not perform a disjoint mapping of CCs into sub-workloads, and hence some CCs can be part of multiple sub-workloads as well. We used the same W_{Test} constructed in the previous subsection as our testing workload for this scenario as well.

	Number of Constraints			
	W_1	W_2	W_3	W_4
Workload (AQP) Cardinality constraints	308	355	355	318
Metadata Cardinality constraints	1248	1095	1008	952
Total Cardinality constraints	1556	1450	1363	1270

Table 7.1: Division of $W_{Extended}$'s Cardinality constraints for individual sub-workloads

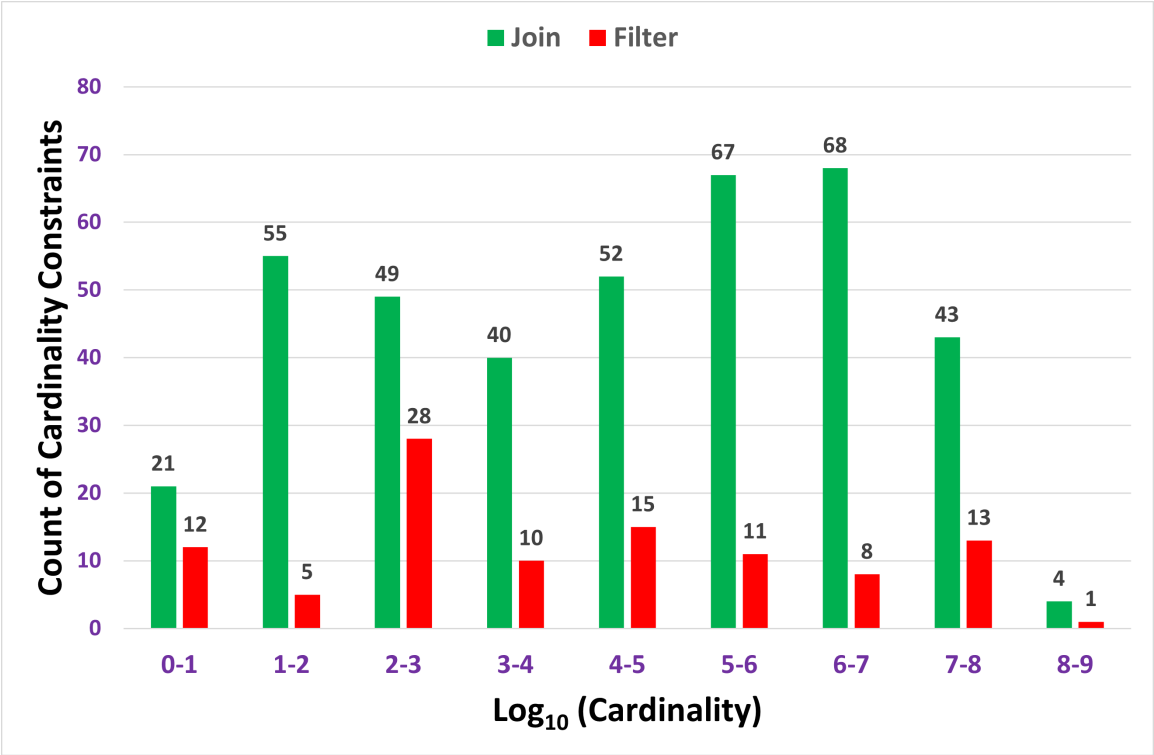


Figure 7.5: Distribution of Cardinality in CCs from AQPs of $W_{Extended}$

7.2 Count of Variables

In this section, we present the number of variables that were created during the LP formulation phase in HF-Hydra in each scenario. This is particularly important, as we create sub-workloads in the split workload scenario based on the heuristic of the number of variables formed in the workload. But for specialized workload scenario, when we work with the level of individual queries, the number of variables is not a pressing concern since we can solve the equations with mere substitution as seen in section 4.6. Hence, here we present the numbers for the other two scenarios. In each case, in addition to presenting the total number, we drill down specifically

to show six relations that had the most number of variables.

7.2.1 Unified Workload Scenario

For our $W_{Unified}$, the number of variables that got formulated is highlighted in Table 7.2.

Relation Name	Number of Variables	
	Hydra	HF-Hydra
Catalog_sales	1,069	1,472
Date_dim	158	447
Inventory	16	898
Item	230	1,458
Store_sales	1,876	4,944
Web_sales	412	437
Total (across all relations)	3,944	10,493

Table 7.2: Number of variables from $W_{Unified}$

7.2.2 Split Workload Scenario

For our $W_{Extended}$, the number of variables that got formulated is highlighted in Table 7.3.

Relation Name	Number of Variables			
	W_1	W_2	W_3	W_4
Catalog_sales	378	466	13,507	6,124
Date_dim	479	302	313	404
Inventory	287	385	140	128
Item	356	292	268	294
Store_sales	1,738	3,935	3,756	3,351
Web_sales	398	392	17,338	9,662
Total (across all relations)	4,254	6,372	35,869	21,225

Table 7.3: Number of variables from $W_{Extended}$

A point here to note is that we used all the CCs from W_{Train} for HF-Hydra, but for the case of Hydra, we used only the 176 queries within W_{Train}, as Hydra cannot handle the inconsistency among CCs of AQP and Metadata stats. For the case of HF-Hydra, we explicitly addressed this in our LP formulation discussed in section 5.4.1. This note applies to all the experiment results presented comparing Hydra and HF-Hydra in the upcoming subsections as well.

7.3 Database Summary Size

7.3.1 Specialized Workload Scenario

In the specialized workload scenario of HF-Hydra, the summary generated is similar to that of Hydra. However, since we deal at the level of each individual query, the summary is much simpler than the one produced in Hydra. Specifically, for each of the 150 queries from $W_{Projection}$, the range of the generated summary sizes was just in the order of *20-40 KBs*, which is minuscule.

7.3.2 Unified Workload Scenario

Coming to our unified workload scenario, a key difference between HF-Hydra and Hydra is with regard to the structure of the database summary. Firstly, there are many more regions in the HF-Hydra summary. Secondly, instead of picking a single point per region, HF-Hydra's summary retains the entire region and generates a diverse spread of tuples. The summary sizes produced from both Hydra and HF-Hydra for W_{Train} is illustrated in Table 7.4. We can see 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.

	Hydra	HF-Hydra
Database Summary Size	40 KB	21.8 MB

Table 7.4: Database Summary size for $W_{Unified}$

7.3.3 Split Workload Scenario

In our split workload scenario of HF-Hydra, the individual summary sizes for each of the four sub-workloads created from our workload-division strategy for $W_{Extended}$ is shown in Table 7.5. We can note here that the summary sizes are comparatively higher for W_3 and W_4. This can be associated with the large number of variables that got formulated for these two sub-workloads as can be seen from Table 7.3. Thus, the summary for W_3 and W_4 constitutes a richer summary

with a much more number of regions than that of W_1 and W_2.

	W_1	W_2	W_3	W_4
Database Summary Size	5.1 MB	5.5 MB	87.1 MB	50 MB

Table 7.5: Database Summary size for $W_{Extended}$

7.4 Database Summary Construction Time

7.4.1 Specialized Workload Scenario

For each of the 150 queries from $W_{Projection}$, the database summary construction time were observed to be very practical. Specifically, the individual summary construction process for each query took *less a second*.

7.4.2 Unified Workload Scenario

Coming to our unified workload scenario of HF-Hydra, the summary construction time for $W_{Unified}$ is shown in Table 7.6. We can note here that the construction times are comparatively higher for HF-Hydra, and this can be associated with the use of optimization function to get a better inter-region distribution. To throw more light, the time taken by the solver to process our LP formulation is mentioned explicitly – however, in *absolute* terms, the summary construction time is still small enough to be easily viable on contemporary computing platforms.

	Hydra	HF-Hydra
Total Summary Construction Time	1 min 26 sec	5 min 56 sec
Time taken by Solver	6 sec	2 min 55 sec
Summary Construction Time (Excluding Solver)	1 min 20 sec	3 min 1 sec

Table 7.6: Database Summary Construction Time for $W_{Unified}$

7.4.3 Split Workload Scenario

In our split workload scenario of HF-Hydra, the individual summary construction times for each of the four sub-workloads created from our workload-division strategy for $W_{Extended}$ is shown in Table 7.7. We can note here that the construction times are comparatively higher for W_3 and W_4. This can be again associated with the large number of variables that got

formulated for these two sub-workloads, as can be seen from Table 7.3. Also, to stress the overheads of the solver in our summary construction process, we exclusively present the time taken by the solver. We can clearly see that with an increase in variables, the solver takes a larger time. Thus, with extended workloads, the load on the solver becomes unmanageable, and eventually ends up crashing. This motivates us to use our workload-division strategy to process and provide robustness to extended workloads.

	W_1	W_2	W_3	W_4
Total Summary Construction Time	1 min 42 sec	3 min 32 sec	48 min 52 sec	19 min 40 sec
Time taken by Solver	38 sec	1 min 59 sec	39 min 29 sec	9 min 59 sec
Summary Construction Time (Excluding Solver)	1 min 4 sec	1 min 33 sec	9 min 23 sec	9 min 41 sec

Table 7.7: Database Summary Construction Time for $W_{Extended}$

7.5 Dynamic Generation Time

In HF-Hydra, we have the ability to generate the entire database *on-the-fly* from the database summary. The database summary can be used to create materialized database as well. To verify whether dynamic generation can indeed produce data at rates that are practical for supporting query execution, for both the specialized workload scenario and unified workload scenario, we constructed materialized databases and made a comparison. The dynamic generation time of HF-Hydra, as compared to Hydra and the standard sequential scan from the disk are presented in Table 7.8. The split workload scenario uses the same framework of dynamic generation as that of the unified workload scenario, and so we exclude it from presenting.

Disk Scan	Dynamic Generation			
	Hydra	HF-Hydra		
		Specialized Summary Scenario	Unified Summary Scenario	Split Summary Scenario
8 min 1 sec	5 min 17 sec	4 min 11 sec	6 min 44 sec	6 min 40 sec

Table 7.8: Dynamic Generation time taken to produce and supply 100 GB of data

We see here that dynamic generation is not only competitive with a materialized solution but is in fact, typically faster. Therefore, using dynamic generation can prove to be a good option since it can help to eliminate the large time and space overheads incurred in: (1) dumping generated data on the disk, and (2) loading the data on the engine under test.

7.6 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 100 GB and 1 TB, and generating summary for unified workload scenario. The results are enumerated in Table 7.9. While going from 1× to 10× with the database size, the summary sizes and the generation time do not vary much.

Data Scale (TPC-DS)	100 GB	1 TB
Database Summary Size	21.8 MB	35.6 MB
Summary Generation Time	5 min 56 sec	8 min 34 sec

Table 7.9: Data Scale Experiment Analysis

7.7 Heuristic Validation

Recall that we mentioned in section 6.3 that, we make use of the *heuristic* of the number of variables generated from the constraints in a workload alone to tell whether it is within the handling capacity of the solver. To portray the impact of the number of variables on the time taken to obtain a solution from the LP solver in HF-Hydra, we consider three workloads, and present the number of variables and the time taken by the solver in each case in Table 7.10. The workloads chosen are as follows: (a) the W_{Train} workload used for our unified workload scenario, (b) the workload used for experimental evaluation in [27], (let's call it W_A), which contained 225 cardinality constraints sourced from AQPs and 2622 cardinality constraints sourced from metadata constraints, and (c) the same $W_{Extended}$, but with 3077 metadata constraints (let's call it W_B). We can clearly observe that with increase in the number of variables, the solving time significantly increases and with further increase in variables, the solver ends up crashing. Hence, the heuristic used for splitting the workloads in split workload scenario is reasonable.

	W_{Train}	W_A	W_B
Variable Count (Top 3 relations)	4,944 & 1,472 & 1,458	64,427 & 60,792 & 4,027	14,87,868 & 1,55,084 & 61,314
Total Variable Count (across all relations)	10,493	1,33,940	17,45,397
Total Time Taken by Solver	2 min 55 sec	22 hr 36 min 27 sec	Crash

Table 7.10: Impact of Variable Count on Solver Time

7.8 Data Diversity and Realism

To showcase the diversity in our synthetic data generation, we drill down into the *look* of the data produced by HF-Hydra. For this purpose, a sample fragment of Item relation from the TPC-DS produced by Hydra is shown in Figure 7.6a, and the corresponding fragment generated by HF-Hydra is shown in Figure 7.6b. It is evident from these snapshots that unlike Hydra, which has heavily repeated attribute values – for instance, all the REC_START_DATE values are the same, HF-Hydra delivers more diversified realistic databases with significant variations in the attribute values.

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

(a) Hydra Database (Item)

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

(b) HF-Hydra Database (Item)

Figure 7.6: Showcasing Data Diversity in HF-Hydra

7.9 Volumetric Similarity - Training Queries

Volumetric similarity on training queries is affected by the addition of spurious tuple while ensuring referential integrity. Here we throw light on the number of such tuples that were added.

7.9.1 Specialized Workload Scenario

For each of the 150 queries of $W_{Projection}$, we achieved *100 percent* volumetric similarity on all types of CCs, specifically, base relations, filters over base relations, join nodes, and projection nodes, from the AQP of individual queries. This is because, while working on the level of each individual query, the summary is directly constructed without any violation of referential integrity, and hence there is no addition of spurious extra tuples.

7.9.2 Unified Workload Scenario

Coming to our unified workload scenario, we present the share of the spurious tuples as compared to the relation's actual cardinality for some relations of the TPC-DS benchmark in Figure 7.7. We can clearly see that the number of extra tuples added is negligibly small. Also, the number of extra tuples that get added are *independent of scale* of the actual database, which is a property we retain from Hydra. We exclude presenting the figures for split workload scenario, as it shares the same summary construction mechanism as that of the unified workload scenario, and so the results are very similar.

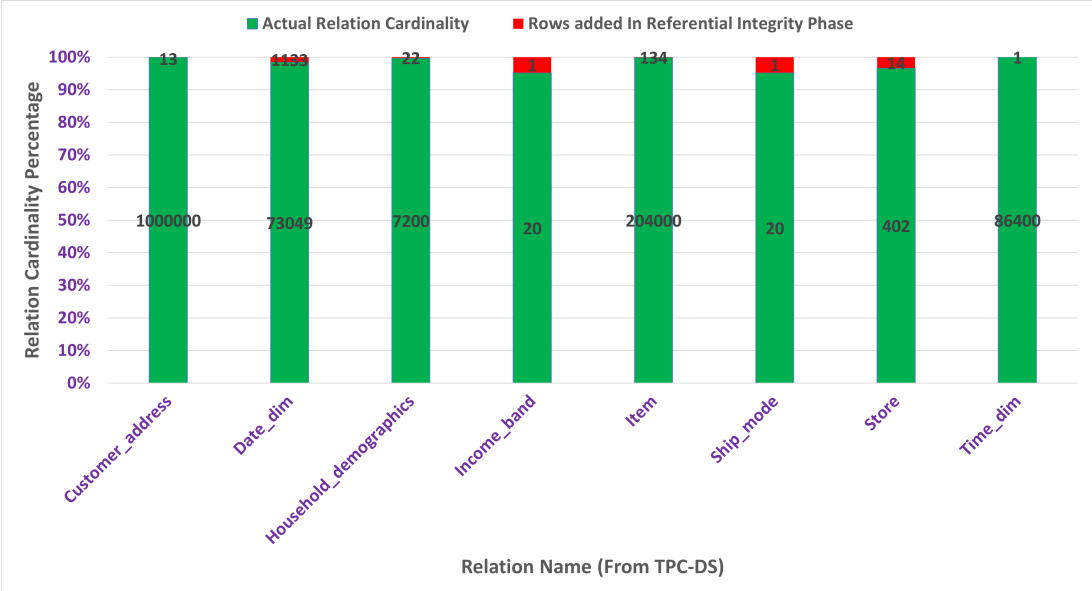


Figure 7.7: Volumetric similarity for Training Queries

7.10 Volumetric Similarity - Testing Queries

For evaluating the volumetric accuracy on the constraints derived from the testing queries, we use the **UMBRAE** (Unscaled Mean Bounded Relative Absolute Error) model-comparison metric [14], with Hydra serving as the reference model. Apart from its core statistical soundness, UMBRAE's basis in the absolute error is compatible with the L1 norm used in the HF-Hydra optimization functions. UMBRAE values range over the positive numbers, and a value U has the following physical interpretation:

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

The value U is computed using the formula:

$$U = \frac{MBRAE}{1 - MBRAE} \ ,$$

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

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

We chose the UMBRAE metric to present our results owing to its following salient features: (a) Informative: it can provide an informative result without the need to trim errors (b) Resistant to outliers: it can hardly be dominated by a single forecasting outlier (c) Symmetric: over-estimates and under-estimates are treated fairly (d) Scale-independent: it can be applied to data sets on different scales; (e) Interpretability: it is easy to understand and can provide intuitive results.

7.10.1 Unified Workload Scenario

The UMBRAE values obtained by HF-Hydra over the 55 queries from W_{Test} are presented here, with Hydra serving as the reference baseline ($U = 1$). For a clearer understanding, the results for base filters and join nodes are shown separately in Figures 7.8 and 7.9, respectively. For join nodes, we provide the breakup showing individual accuracies for CCs involving 1-4 joins. We see that HF-Hydra delivers **44 percent** better performance on filters while also achieving an improvement of **27 percent** with regard to joins. The reason for the greater improvement in filters over join nodes is that metadata statistics is typically maintained on a per-column basis,

making it harder to capture joins that combine information across columns. On drilling down in the join nodes, we see a performance improvement of *32 percent* for CCs with one joins, *27 percent* for CCs with two joins, *19 percent* for CCs with three joins, and a marginal *3 percent* for CCs with four joins. We can note that as the number of joins increases, the improvement keeps reducing, as should be expected due to the progressively reduced quality of the input statistics.

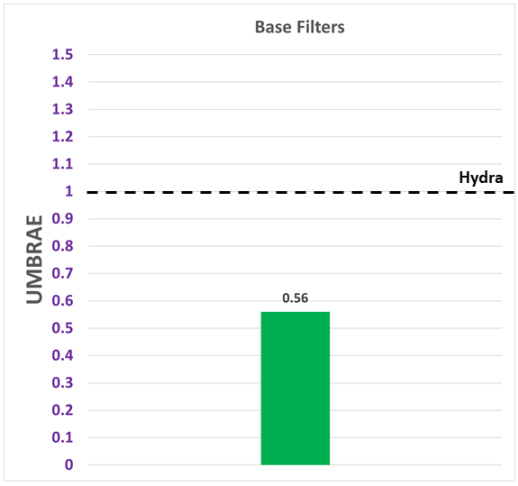


Figure 7.8: Base Filter accuracy for Unified Workload Scenario

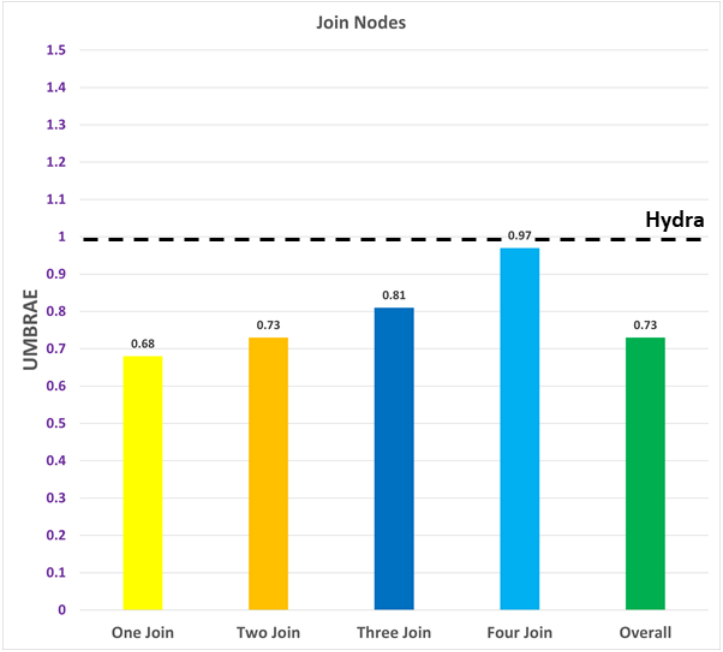


Figure 7.9: Join Nodes accuracy for Unified Workload Scenario

7.10.2 Split Workload Scenario

The UMBRAE values obtained by HF-Hydra over the 55 queries from W_{Test} is presented here for the four sub-workloads created by our workload-division strategy run over $W_{Extended}$. Unlike the unified workload scenario, a unified database from Hydra to serve as the reference baseline is not possible for $W_{Extended}$. Also, from Table 7.1, we can see that about a quarter of constraints from $W_{Extended}$ has gone into each sub-workload. Hence, to lay grounds for a fair comparison, we picked up a subset of half, i.e. 100 AQPs from $W_{Extended}$ each chosen at random and used that to get a database from Hydra to serve our baseline model. As earlier, for clearer understanding, the results for base filters and join nodes are shown separately, and the join nodes are broken up to show accuracy for CCs involving 1-4 joins.

Using the procedure discussed in section 6.5 for test query forwarding, we assigned all the queries from W_{Test} to the sub-workloads where it is likely to achieve maximum volumetric similarity. The number of queries that each sub-workload was assigned is quantified in Table 7.11.

W_1	W_2	W_3	W_4
14	10	17	14

Table 7.11: Best Assignment for W_{Test}

Also, to showcase the effectiveness of our test query forwarding logic, we also depict the effect of the reverse, i.e., each test query mapped to the sub-workload where it is likely to achieve the worst volumetric similarity. This is done by tweaking the same logic from section 6.5, but this time choosing the sub-workload with minimum points instead of maximum. For this case, the number of queries that each sub-workload was assigned is quantified in Table 7.12.

W_1	W_2	W_3	W_4
15	13	13	14

Table 7.12: Worst Assignment for W_{Test}

The performance of each sub-workload with respect to filter nodes, both in the best and worst case assignments is shown in Figure 7.10. Also, the performance of each sub-workload with respect to join nodes, both in the best and worst case assignments are shown in Figures 7.11, 7.12, 7.13, 7.14.

A point here to note that we intend to show the net effect in a sub-workload for both best and worst case assignments. The actual subset of queries from W_{Test} assigned to a specific

workload in both the best and worst is very likely to be different, and so the number of filter and join edges that feature in each sub-workload will also be different. This can be especially true for higher order joins, as not all queries from our W_{Test} feature four joins. This can be evidently seen from Figures 7.11 and 7.12 that there were no four join nodes in the worst case assignment scenario for W_1 as well as best case assignment for W_2. This is marked by NA in the figure.

From figure 7.10, we can see that for each sub-workload, the overall accuracy improvement over the baseline for Filter nodes is close to **30 percent** on average. We can also clearly see the severe accuracy degradation in the worst case assignment in each case. Specifically, for W_1 and w_3, the performance is worse than the baseline itself, and for W_4, the performance is almost the same as the baseline.

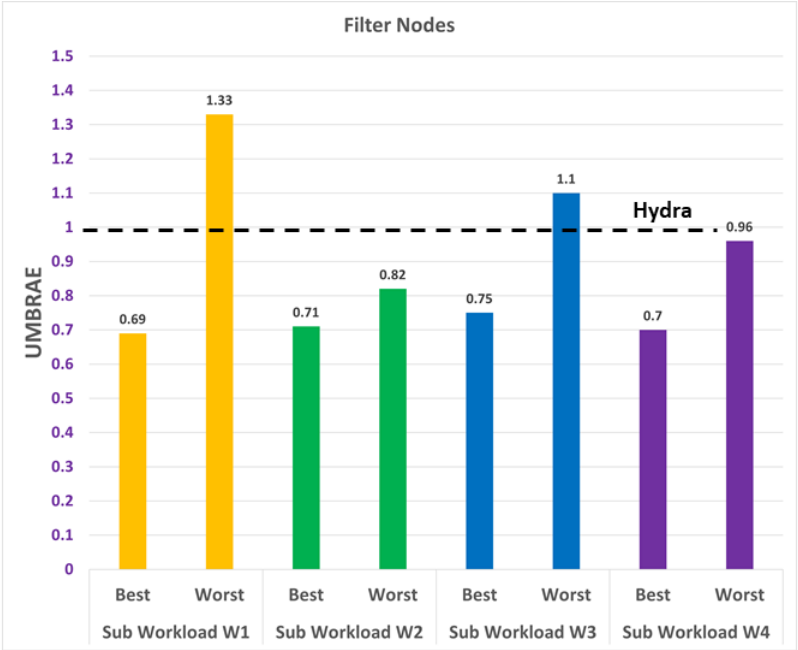


Figure 7.10: Base Filter accuracy for Split Workload Scenario

Coming to performance in the join nodes, from Figures 7.11, 7.12, 7.13, 7.14, we can observe that the overall accuracy improvement over the baseline is close to **20 percent** in average. We can observe a lot of cases wherein the best case and worst case performance of HF-Hydra in the sub-workloads, especially in the higher order joins, to be identical to that of the baseline. This is because we observed that for many higher order joins, the cardinality of both the cases of Hydra and HF-Hydra turned out to be zero. So numerically, it denotes identical performance as that of the baseline. This is again due to the progressively reduced quality of the input statistics, as seen in the results of unified workload scenario itself.

We can also in general, observe that for both filters and joins, the accuracy improvement is not as great as it was for the unified summary case in the previous subsection. One obvious reason is that with a reduced number of constraints in a sub-workload, the summary would not be rich as it otherwise would be with information from all the training queries. Secondly, while forwarding the test query, we chose only one of the sub-workload likely to give maximum volumetric similarity, and so, we inevitably lose out on the performance of some edges of the query which may have had better intersections with sub-domains of the other sub-workloads.

Figure 7.11: Join Nodes accuracy for Sub Workload W_1

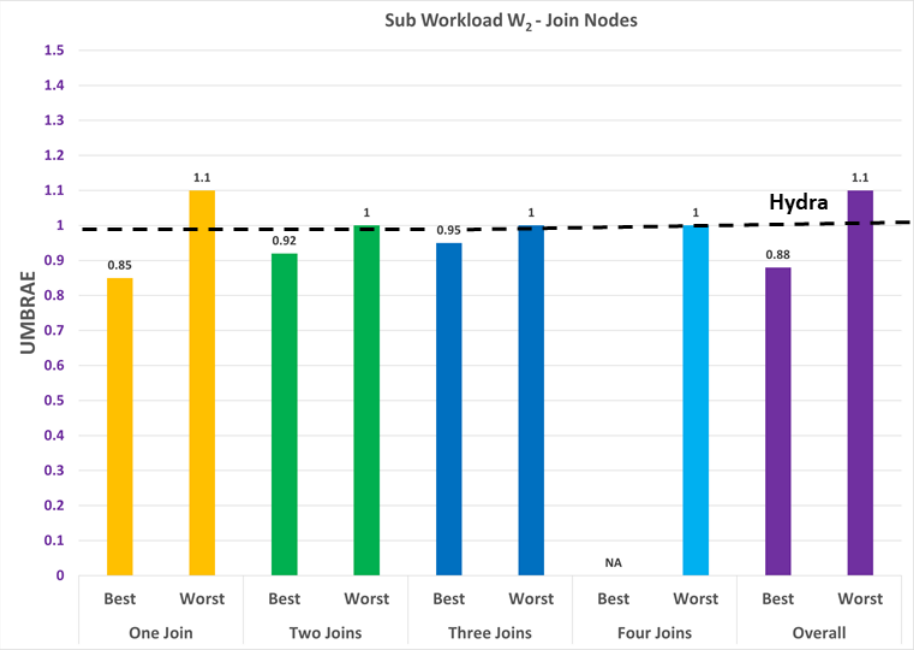


Figure 7.12: Join Nodes accuracy for Sub Workload W_2

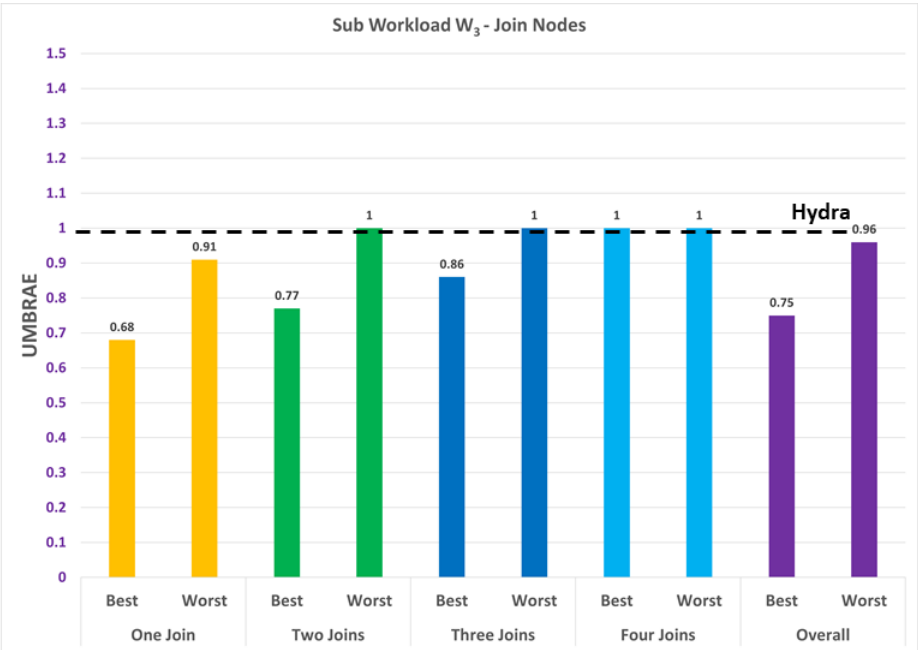


Figure 7.13: Join Nodes accuracy for Sub Workload W_3

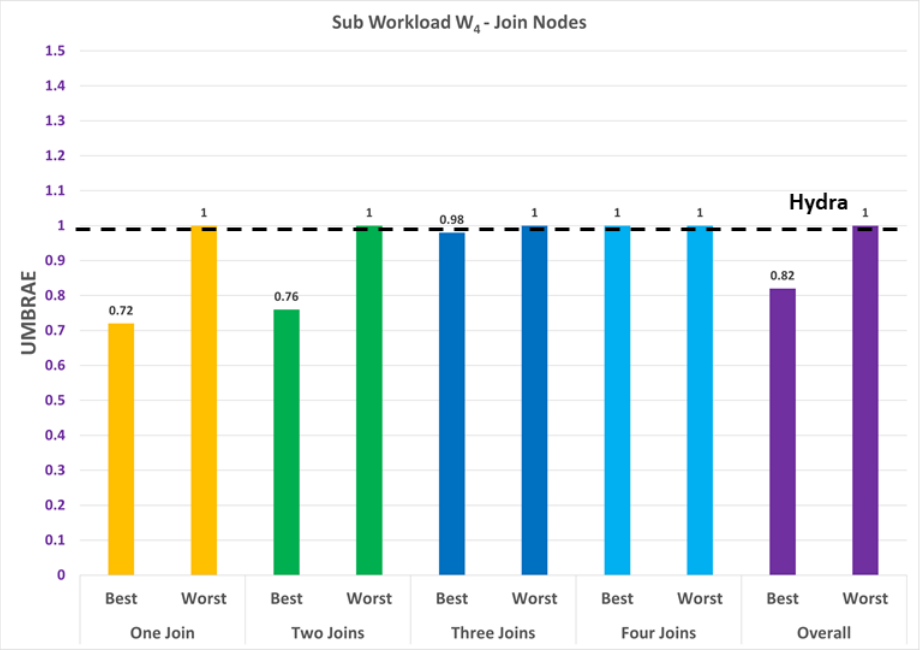


Figure 7.14: Join Nodes accuracy for Sub Workload W_4

7.11 Optimal Number of Sub-Workloads

Recall that we mentioned in sub-section 6.4.1 that, with our attribute intervalisation procedure, and eventual sub-workload splitting, we do not guarantee any minimality. However, to investigate how far we were from the optimal number of sub-workloads, we tried workload division by brute force, and figured out that for our $W_{Extended}$, 3 sub-workloads were sufficient to have each workload feasible. But interestingly, the solving time for two of them was very high. We present these figures in Table 7.13.

	W_1	W_2	W_3
Total Variable Count	7,030	69,598	40,810
Total Time taken by Solver	1 min 36 sec	2 hr 41 min 26 sec	52 min 23 sec

Table 7.13: Optimal Number of Sub-Workloads for $W_{Extended}$

This clearly shows that, not only the number of variables, but also a careful consideration of the nature of the equations, and other aspects like including the underlying data statistics of the attributes comprising the workload are essential to be included in our workload-splitting algorithm, and so we consider them as interesting future work.

Chapter 8

Conclusion and Future Work

Testing database engines efficiently is a critical issue in the industry, and the ability to accurately mimic client databases forms a key challenge in this effort. A rich body of literature exists on data regeneration, beginning with *workload-independent* followed by *workload-dependent* techniques. Specifically, workload-dependent techniques aim at producing volumetric similarity to seen (training) query workloads. A common limitation in these contemporary techniques is that they run into issues of *scale* or *efficiency* at one stage or the other in the regeneration pipeline. This is partly due to their focus on *materialized* static solutions, making them impractical at large volumes. Also, in testing applications, generalization to new queries can be necessary requirement at the vendor site. Hence, reasonable volumetric similarity on any unseen (testing) query workload is critical as well. This ability to provide *robustness* to test queries has not been explicitly addressed in generators proposed in the literature.

Hydra is a recently proposed workload-aware data regenerator, which ensures *volumetric similarity* on training queries. Hydra consciously addresses data scale and efficiency issues through the entire regeneration pipeline. Hydra constructs a database summary, which is then used to *dynamically regenerate* the database, on-the-fly during query execution. Also, the summary construction is *independent of the data scale.* Hence, Hydra can practically cater to client databases with large volumes, unlike many of its competitors.

Notwithstanding these desirable characteristics, Hydra suffers from critical limitations: (a) limited scope of SQL operators in the training query workload, (b) lack of scalability with respect to the size of the training query workload, and (c) poor volumetric similarity on test queries. The inherent scalability limitation in the size of the training query workload stems from the computational limitation of the theorem prover. The consequences are restrictions on the training workload size, as well as inaccurate similarities for test queries. Robustness on test queries is further adversely affected by design deficiencies such as a lack of preference among

candidate synthetic databases, and restricted domain value representations in the generated data.

8.1 Conclusion

In this thesis, we presented **High-Fidelity Hydra (HF-Hydra)**, an enhancement over Hydra, to address the critical limitations of Hydra. In the first segment of the thesis, we discussed extending the ambit of Hydra's operator coverage which is restricted to only equality and range predicates in filter constraints, to also include the LIKE operator. We proposed a solution to discover and handle intersections. We then transformed the LIKE predicates into equivalent equality filter predicates, which can be eventually processed by the existing setup.

In the next segment of the thesis, we looked into the specialized case that focuses only on handling training queries. Here, HF-Hydra provided an extreme solution wherein for each query in the workload, a different database summary was produced. The idea here was that with dynamic generation in action, when a training query is fired, the summary for that query is picked to generate tuples on the fly. With this approach, we bypassed the inherent scalability limitation and managed to handle any number of training queries. We also extended support for projection-based SQL operators in this scenario.

In the subsequent segment of the thesis, we discussed the framework of HF-Hydra that addressed the general case, where the focus was also to provide robustness for test queries. We modeled the metadata statistics as constraints and augmented them to obtain a *refined* region-partitioning of the data space. With optimization functions in our LP formulation, the optimizer was leveraged to make a focused choice among the candidate databases. Finally, a summary was constructed, and for each region therein with a non-zero cardinality assignment, we generated tuples adhering to the optimizer's distribution model, with greater diversity in the represented domain values.

In the last segment of the thesis, we proposed a *workload-division* strategy to produce a few sub-workloads, and a summary was produced for each sub-workload. We used the heuristic of whether the number of variables that are produced from the constraints in a sub-workload was under the limit that the solver could easily handle. When a test query was fired, we then deterministically directed the test query to be executed on the data generated from the sub-workload where the volumetric similarity was likely to be maximum.

Finally, we saw a detailed experimental evaluation discussing each scenario with a diverse set of workloads derived from the TPC-DS benchmark, and confirmed that HF-Hydra takes a substantive step forward with regard to creating expressive, robust and scalable data regeneration frameworks, with organic relevance to testing deployments.

8.2 Future Work

Following are some areas with future research scope regarding the design of HF-Hydra:

1. Extend the ambit of HF-Hydra to support projection-inclusive constraints in the framework of providing robustness to test queries as well. Also we would like to work on extending support to queries with other operators like Union, Group-By and also Having clause.

2. Include more input characteristics of the client database and model in the form of CCs to help in obtaining a better inter-region solution. While doing so, we need to make sure that we don't compromise on the data privacy of the client. Also, the additional information we demand should be practical to construct and ship to the vendor.

3. 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 HF-Hydra. We would like to try and incorporate the recently proposed selectivity estimation algorithms based on machine-learning techniques (e.g., NARU [31], MSCN [18], SDV [21]) into the HF-Hydra framework.

4. We currently use the solver as a black box to get a feasible solution for the LP. Instead, we would like to develop an invasive algorithm with guarantees, to guide the solver to prefer one solution over the other, so that would eventually help in better inter-region distribution, thus increasing the fidelity in the generated synthetic database with respect to the original client database.

5. Currently, our workload-division strategy uses the heuristic of number of variables to perform the splitting of constraints into sub-workloads. We would like to design an algorithm to include into consideration the nature of the system of equations that are formed, and also the underlying data statistics of the attributes comprising the workload.

6. We would like to enhance our workload-division strategy to explore other methods for dividing the domain space into associated sub-workloads, as well as provide provable guarantees of minimality with respect to the number of sub-workloads produced.

Bibliography

[1] CODD Metadata Processor. dsl.cds.iisc.ac.in/projects/CODD. 7, 60

[2] Hydra Database Regenerator. dsl.cds.iisc.ac.in/projects/HYDRA. 84

[3] *USE PLAN SQL Server.* https://technet.microsoft.com/en-us/library/ms186954(v=sql.105).aspx. 1

[4] TPC-H. http://www.tpc.org/tpch/. 2

[5] Z3 Solver. https://github.com/Z3Prover/z3. 5, 56

[6] Alexander Alexandrov, Kostas Tzoumas, and Volker Markl. Myriad: Scalable and Expressive Data Generation. *Proc. VLDB Endow.*, 5(12):1890–1893, 2012. doi: 10.14778/2367502.2367530. URL http://vldb.org/pvldb/vol5/p1890_alexanderalexandrov_vldb2012.pdf. 13

[7] Arvind Arasu, Raghav Kaushik, and Jian Li. Data Generation Using Declarative Constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12 2011 - June 16 2011*, pages 685–696. ACM, 2011. doi: 10.1145/1989323.1989395. URL https://doi.org/10.1145/1989323.1989395. 1, 2, 3, 12, 14, 19, 24

[8] Arvind Arasu, Raghav Kaushik, and Jian Li. DataSynth: Generating Synthetic Data using Declarative Constraints. *Proc. VLDB Endow.*, 4(12):1418–1421, 2011. doi: 10.14778/3402755.3402785. URL http://www.vldb.org/pvldb/vol4/p1418-arasu.pdf. 2, 23

[9] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14,*

2011 - July 20, 2011. Proceedings, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi: 10.1007/978-3-642-22110-1_14. URL https://doi.org/10.1007/978-3-642-22110-1_14. 5

[10] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse Query Processing. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15 2007 - April 20 2007*, pages 506–515. IEEE Computer Society, 2007. doi: 10.1109/ICDE.2007.367896. URL https://doi.org/10.1109/ICDE.2007.367896. 14

[11] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. QAGen: Generating Query-Aware Test Databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD 2007, Beijing, China, June 12 2007 - June 14 2007*, pages 341–352. ACM, 2007. doi: 10.1145/1247480.1247520. URL https://doi.org/10.1145/1247480.1247520. 1, 2, 3, 12, 14, 16, 18

[12] Nicolas Bruno and Surajit Chaudhuri. Flexible Database Generators. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, Trondheim, Norway, August 30 2005 - September 2 2005*, pages 1097–1107. ACM, 2005. doi: 10.5555/1083592.1083719. URL http://www.vldb.org/archives/website/2005/program/paper/wed/p1097-bruno.pdf. 1, 12

[13] Teodora Sandra Buda, Thomas Cerqueus, John Murphy, and Morten Kristiansen. ReX: Representative Extrapolating Relational Databases. In Sebastian Maneth, editor, *Proceedings of the 30th British International Conference on Databases, BICOD 2015, Edinburgh, UK, July 6 2015 - July 8 2015*, volume 9147 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2015. doi: 10.1007/978-3-319-20424-6_10. URL https://doi.org/10.1007/978-3-319-20424-6_10. 13

[14] Chao Chen, Jamie Twycross, and Jonathan Garibaldi. A new accuracy measure based on bounded relative error for time series forecasting. *PLOS ONE*, 12:1–23, 03 2017. doi: 10.1371/journal.pone.0174202. URL https://doi.org/10.1371/journal.pone.0174202. 96

[15] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*

2008, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://link.springer.com/chapter/10.1007/978-3-540-78800-3_24. 5, 23, 24, 56

[16] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, SIGMOD 1994, Minneapolis, Minnesota, USA, May 24 1994 - May 27 1994*, pages 243–252. ACM Press, 1994. doi: 10.1145/191839.191886. URL https://doi.org/10.1145/191839.191886. 1, 12

[17] Joseph E. Hoag and Craig W. Thompson. A Parallel General-Purpose Synthetic Data Generator. *SIGMOD Rec.*, 36(1):19–24, 2007. doi: 10.1145/1276301.1276305. URL https://doi.org/10.1145/1276301.1276305. 12

[18] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13 2019 - January 16 2019, Online Proceedings*. www.cidrdb.org, 2019. URL http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf. 105

[19] Yuming Li, Rong Zhang, Xiaoyan Yang, Zhenjie Zhang, and Aoying Zhou. Touchstone: Generating Enormous Query-Aware Test Databases. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11 2018 - July 13 2018*, pages 575–586. USENIX Association, 2018. doi: 10.5555/3277355.3277411. URL https://www.usenix.org/conference/atc18/presentation/li-yuming. 20

[20] Eric Lo, Nick Cheng, Wilfred W. K. Lin, Wing-Kai Hon, and Byron Choi. MyBenchmark: Generating Databases for Query Workloads. *Proc. VLDB Endow.*, 23(6):895–913, 2014. doi: 10.1007/s00778-014-0354-1. URL https://doi.org/10.1007/s00778-014-0354-1. 1, 14, 18, 21

[21] N. Patki, R. Wedge, and K. Veeramachaneni. The Synthetic Data Vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 399–410, October 2016. doi: 10.1109/DSAA.2016.49. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7796926. 105

[22] Tilmann Rabl, Michael Frank, Hatem Mousselly Sergieh, and Harald Kosch. A Data Generator for Cloud-Scale Benchmarking. In *Performance Evaluation, Measurement and*

Characterization of Complex Systems - Second TPC Technology Conference, TPCTC 2010, Singapore, September 13 2010 - September 17 2010, volume 6417 of *Lecture Notes in Computer Science*, pages 41–56. Springer, 2010. doi: 10.1007/978-3-642-18206-8_4. URL https://doi.org/10.1007/978-3-642-18206-8_4. 12, 13

[23] Tilmann Rabl, Manuel Danisch, Michael Frank, Sebastian Schindler, and Hans-Arno Jacobsen. Just can't get enough: Synthesizing Big Data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD 2015, Melbourne, Victoria, Australia, May 31 2015 - June 4 2015*, pages 1457–1462. ACM, 2015. doi: 10.1145/2723372.2735378. URL https://doi.org/10.1145/2723372.2735378. 12, 13

[24] Ashoke S. and Jayant R. Haritsa. CODD: A Dataless Approach to Big Data Testing. *Proc. VLDB Endow.*, 8(12):2008–2011, 2015. doi: 10.14778/2824032.2824123. URL http://www.vldb.org/pvldb/vol8/p2008-s.pdf. 1, 23

[25] Anupam Sanghi, Raghav Sood, Jayant R. Haritsa, and Srikanta Tirthapura. Scalable and Dynamic Regeneration of Big Data Volumes. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26 2018 - March 29 2018*, pages 301–312. OpenProceedings.org, 2018. doi: 10.5441/002/edbt.2018.27. URL https://doi.org/10.5441/002/edbt.2018.27. 2, 84

[26] Anupam Sanghi, Raghav Sood, Dharmendra Singh, Jayant R. Haritsa, and Srikanta Tirthapura. HYDRA: A Dynamic Big Data Regenerator. *Proc. VLDB Endow.*, 11(12): 1974–1977, 2018. doi: 10.14778/3229863.3236238. URL http://www.vldb.org/pvldb/vol11/p1974-sanghi.pdf. 2

[27] Anupam Sanghi, Rajkumar Santhanam, and Jayant R. Haritsa. Towards Generating Hifi Databases. In *Database Systems for Advanced Applications - 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11 2021 - April 14, 2021, Proceedings, Part I*, volume 12681 of *Lecture Notes in Computer Science*, pages 105–112. Springer, 2021. doi: 10.1007/978-3-030-73194-6_8. URL https://doi.org/10.1007/978-3-030-73194-6_8. 93

[28] Entong Shen and Lyublena Antova. Reversing Statistics for Scalable Test Databases Generation. In *Proceedings of the 6th International Workshop on Testing Database Systems, DBTest 2013, New York, NY, USA, June 24, 2013*, pages 7:1–7:6. ACM, 2013. doi: 10.1145/2479440.2479445. URL https://doi.org/10.1145/2479440.2479445. 12, 13

[29] John M. Stephens and Meikel Poess. MUDD: A Multi-Dimensional Data Generator. In *Proceedings of the 4th International Workshop on Software and Performance, WOSP 2004*, page 104–109, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581136730. doi: 10.1145/974044.974060. URL https://doi.org/10.1145/974044.974060. 12

[30] Y. C. Tay, Bing Tian Dai, Daniel T. Wang, Eldora Y. Sun, Yong Lin, and Yuting Lin. UpSizeR: Synthetically Scaling An Empirical Relational Database. *Inf. Syst.*, 38(8):1168–1183, 2013. doi: 10.1016/j.is.2013.07.004. URL https://doi.org/10.1016/j.is.2013.07.004. 13

[31] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.*, 13(3):279–292, 2019. doi: 10.14778/3368289.3368294. URL http://www.vldb.org/pvldb/vol13/p279-yang.pdf. 105

[32] J. W. Zhang and Y. C. Tay. Dscaler: Synthetically Scaling A Given Relational Database. *Proc. VLDB Endow.*, 9(14):1671–1682, 2016. doi: 10.14778/3007328.3007333. URL http://www.vldb.org/pvldb/vol9/p1671-zhang.pdf. 13