Synthetic Regeneration of Relational Data at Scale

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Engineering

IN

Computer Science and Engineering

BY

Raghav Sood

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

June, 2017

Declaration of Originality

I, **Raghav Sood**, with SR No. **04-04-00-10-41-15-1-12209** hereby declare that the material presented in the report titled

Synthetic Regeneration of Relational Data at Scale

represents original work carried out by me in the **Deparment of Computer Science and Automation** at **Indian Institute of Science** during the years **2015-2017**.
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 discusions.
- 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: 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 Nation

Acknowledgements

I would like to express my sincere gratitude to my project advisor, Prof. Jayant R. Haritsa for giving me an opportunity to work on this project. I am thankful to him for his valuable guidance and moral support. I feel lucky to be able to work under his supervision.

I also sincerely thank Mr. Anupam Sanghi, Huawei Technologies Bengaluru and Prof. Srikanta Tirthapura, Iowa State University. Without their collaboration this work would not have been possible.

I would also like to thank the Department of Computer Science and Automation for providing excellent study environment. The learning experience has been really wonderful here.

Finally I would like to thank all IISc staff, my family and friends for helping me at critical junctures and making this project possible.

Abstract

To locally reproduce and analyze a client's problem, database vendors often construct a *synthetic* version of the client's database and query workload at their development sites. Such a regenerative approach becomes imperative in the world of Big Data systems, where transferring and storing client data at vendor sites has impractical space and time overheads, apart from the standard privacy and liability risks.

A rich body of literature exists on synthetic database regeneration, but suffers critical limitations with regard to maintaining statistical fidelity to the client database and/or scaling to large volumes. In this report, we present PhoneyMocker, a database generator that leverages the declarative approach to data regeneration proposed in [1], and materially improves on it by adding scale, dynamism and functionality. Specifically, PhoneyMocker incorporates an optimized LP (linear programming) formulation that replaces the *grid-partitioning* approach of [1] by a *region-partitioning* approach, in the process delivering an algorithm that reduces the LP complexity by orders of magnitude. Secondly, PhoneyMocker introduces the potent concept of dynamic regeneration by using a minuscule database summary that can on-the-fly create databases of arbitrary size. Finally, PhoneyMocker extends the scope of the generation framework to a richer set of database schemas, query workloads and data operators.

A detailed experimental evaluation based on the TPC-DS benchmark demonstrates that PhoneyMocker can be successfully used to handle the requirements of contemporary database deployments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In industrial practice, a common requirement for database vendors is to be able to test their database engines with representative data and workloads that accurately mimic the data processing environments at client 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, with the advent of the so-called Big Data systems, transferring and storing the data at the vendor's site may prove to have impractical space and time overheads. Therefore, vendors need the ability to *regenerate* a *synthetic* version of the client's data processing environment at their development sites.

To address the data regeneration problem, a particularly potent approach, called *workload-dependent database generation*, was introduced in [4], and has served as the foundation for most of the practicable systems proposed over the last decade, such as DataSynth from Microsoft [5, 1, 2]. The basic principle of this approach is to generate synthetic data whose behavior is *volumetrically similar* to the original database on the prespecified query workload. That is, assuming a common choice of query execution plans at the client and vendor sites (which can be ensured either through plan forcing or metadata matching), the output row cardinalities of the individual operators in these plans are very similar in the original and synthetic databases.

A common limitation among the prior work is that they all run into the issues of *scale* and *efficiency*, at some stage or the other in the regeneration pipeline. So, for example, in DataSynth, the focus is on *materialized* static solutions wherein a complete database is created and then analyzed – this approach is not practical at large volumes, or when data is processed in streaming format. Similarly, the ability to scale the generation process

to large query workloads and data volumes has not been clearly established, with the validations being typically restricted to relatively simple and small benchmarks such as TPC-H [9]. Finally, there are restrictions about the schema types and query workloads that are amenable to this framework.

The above limitations become especially problematic from a futuristic Big Data perspective, where we have to cope with enormous data volumes and complex query workloads. To materially address this challenge, we present in this report the PhoneyMocker[1] (referred as PM here onwards) data generation tool, which is based on the DataSynth approach, but ensures that scale and efficiency aspects are addressed through the *entire regeneration pipeline*. As a concrete example, the data processing environment of a 100GB TPC-DS client database with 131 queries was regenerated in less than three minutes at the vendor site using PM!

1.1 Workload Dependent Generation[4]

The goal of the workload dependent generation technique [4] is to generate synthetic data that has the required output row cardinality for each operator in the query execution plan. For instance, consider the following client database schema:

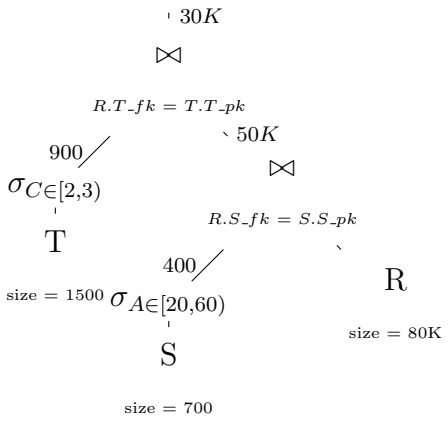
R (R_pk, S_fk, T_fk)
S (S_pk, A, B)
T (T_pk, C)

where pk and fk refer to primary-key and foreign-key attributes, respectively. A sample client query on this schema is shown in Figure 1.1a, with the corresponding query execution plan in Figure 1.1b. Note that this execution plan has the output edge of each operator annotated with the associated row cardinality (as evaluated during the client's execution) – for instance, there are 50000 rows resulting from the join of R and S. Such a plan is referred to as an "Annotated Query Plan" (AQP) in [4]. 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.

[1]Mocking, symbolizes that the generated database volumetrically mocks/mimics the original database and Phoney, symbolizes that the Tuple Generator (Section 5) superficially supplies gigabytes of data with not more than a few kilobytes of data actually existing on disk.

```
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
```

(a) Example Query



(b) Annotated Query Plan (AQP)

Figure 1.1: Example Database Scenario

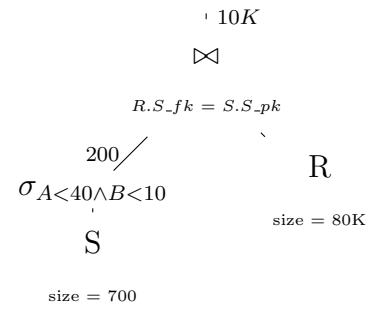
1.2 The DataSynth Generator[1, 2]

Here we give an overview of the related work DataSynth. Their approach leverages the workload-aware generation approach to provide a unified mechanism for handling all data characteristics via declaratively specified *cardinality constraints* (CCs). For instance, the CCs expressing the AQP of Figure 1.1b are shown in Figure 1.2. The data generation technique takes the schematic information and the set of CCs from the client site and produces synthetic data that closely meet these CCs.

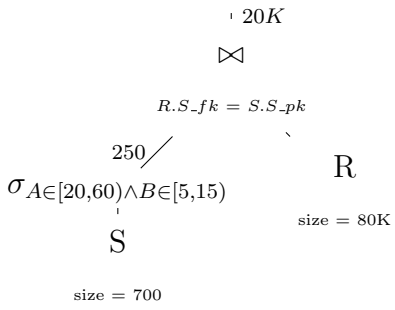
$|R| = 80K$
$|S| = 700$
$|T| = 1500$
$|\sigma_{S.A \in [20,60)}(S)| = 400$
$|\sigma_{T.C \in [2,3)}(T)| = 900$
$|\sigma_{S.A \in [20,60)}(R \bowtie S)| = 50K$
$|\sigma_{S.A \in [20,60) \wedge T.C \in [2,3)}(R \bowtie S \bowtie T)| = 30K$

Figure 1.2: Cardinality Constraints

We describe the generation algorithm of DataSynth with the help of the example discussed in Figure 1.1, and the inclusion of two more queries in the workload, whose AQPs are shown in Figure 1.3.



(a) AQP 2



(b) AQP 3

Figure 1.3: Additional AQPs for Query Workload

The generation algorithm starts by creating a *view* for every intermediate relation using the schema information, resulting in the join expressions in CCs being replaced by single views. More precisely, a view $\mathcal{V}_i$ is a set of non-key attributes that are present in either its corresponding

relation $\mathcal{R}_i$, or in any other relation on which $\mathcal{R}_i$ depends through referential constraints (both directly or transitively). For our running example, the views generated are:

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

All the CCs resulting from the query workload can be rewritten in terms of these views. As a case in point, Figure 1.4 shows the CCs applicable on the R_view.

$$|R\_view| = 80K$$
$$|\sigma_{A\in[20,60)}(R\_view)| = 50K$$
$$|\sigma_{A\in[20,60)\wedge C\in[2,3)}(R\_view)| = 30K$$
$$|\sigma_{A<40\wedge B<10}(R\_view)| = 10K$$
$$|\sigma_{A\in[20,60)\wedge B\in[5,15)}(R\_view)| = 20K$$

Figure 1.4: View-based Cardinality Constraints

Next, for each view, a *linear program* (LP) is formulated where the CCs are expressed as LP constraints. This is augmented by a dimensionality reduction technique that helps to reduce the complexity of the LP, wherein each view is decomposed into a set of sub-views based on co-appearance of attributes in the CCs. For example, the R_view is decomposed into two sub-views: $R_1(A, B)$ and $R_2(A, C)$ since A and B co-appear in Queries 2 and 3, while A and C co-appear in Query 1. This results in the original 3D attribute space (A, B, C) reducing into a pair of 2D problems: (A, B) and (A, C).

Now, to formulate the LP for a view, DataSynth adopts a *grid-partitioning* approach where each sub-view in the view is partitioned such that all its dimensions are split using the constants appearing in the CCs – this partitioning is referred to as *intervalization* in [1]. For example, the grid-partitioning of R_1 and R_2 is shown in Figure 1.5, where each colored box corresponds to a different CC.

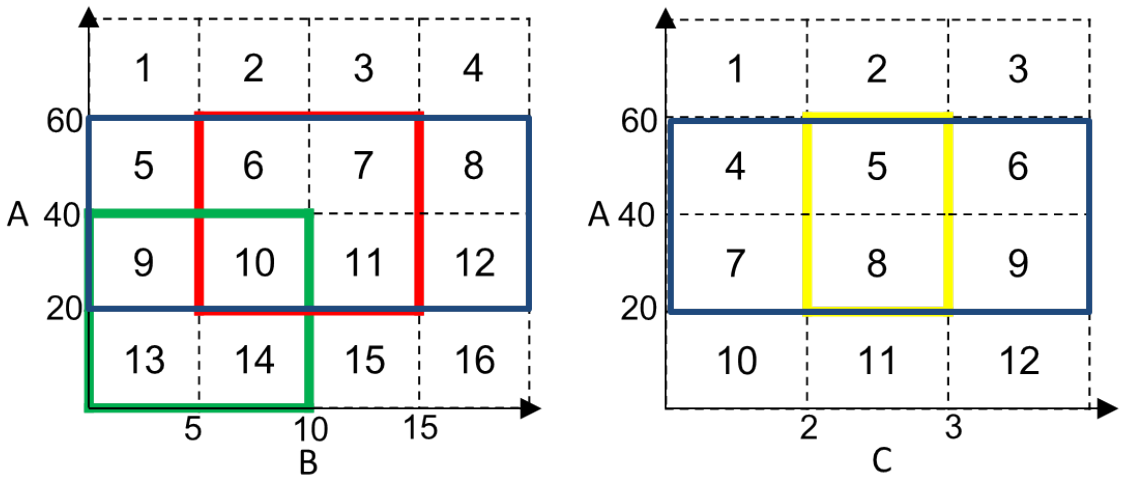


Figure 1.5: Grid-Partitioning in DataSynth

For each cell in the grid, a variable is created which represents the number of data rows present in that cell. So, a total of 28 variables are created, 16 corresponding to the R_1 sub-view: $x_1, x_2, ..., x_{16}$, and the remaining 12 for the R_2 sub-view: $y_1, y_2, ..., y_{12}$. The CCs shown in Figure 1.4 for *R_view* can now be expressed in terms of LP constraints as follows:

$$
\begin{aligned}
x_9 + x_{10} + x_{13} + x_{14} &= 10K \\
x_6 + x_7 + x_{10} + x_{11} &= 20K \\
x_5 + x_6 + ... + x_{12} &= 50K \\
x_1 + x_2 + ... + x_{16} &= 80K \\
y_5 + y_8 &= 30K \\
y_4 + y_5 + ... + y_9 &= 50K \\
y_1 + y_2 + ... + y_{12} &= 80K
\end{aligned}
$$

Sub-views R_1 and R_2 share A as the common dimension. To ensure same data distribution along dimension A, following four *consistency constraints* are added corresponding to the four splits on A:

$$
\begin{aligned}
x_1 + x_2 + x_3 + x_4 &= y_1 + y_2 + y_3 \\
x_5 + x_6 + x_7 + x_8 &= y_4 + y_5 + y_6 \\
x_9 + x_{10} + x_{11} + x_{12} &= y_7 + y_8 + y_9 \\
x_{13} + x_{14} + x_{15} + x_{16} &= y_{10} + y_{11} + y_{12}
\end{aligned}
$$

Additionally, since a variable denotes the number of tuples in a cell, non-negativity constraints are also added for all the variables.

A LP solver is now used to obtain a feasible solution to the enumerated constraints. Since the modeling was carried out at a sub-view level, we need to convert the LP solution to equivalents in the original 3D space. For this, DataSynth takes recourse to a *sampling* algorithm – in the running example, this algorithm will compute $Prob(A, B)$ and $Prob(C|B)$. Then, 80K tuples (size of R_view) will be generated by first sampling a point from the former distribution, and then sampling a point from the latter conditioned on this outcome. Finally, there is a last reconciliation step where a few additional rows are added to some of the views in order to satisfy referential integrity constraints.

1.3 Limitations of DataSynth

Clearly, DataSynth features a variety of novel ideas and problem formulations that go a long way in addressing the data regeneration challenge. However, as enumerated below, there still exist serious drawbacks that adversely impact its ability to handle current database scenarios.

LP Complexity: In spite of the various optimizations to reduce the LP complexity, the number of variables may still be enormous due to the underlying grid-partitioning strategy. This makes the solution process very slow for complex query workloads, and often even infeasible to solve in reasonable time. For instance, with the TPC-DS benchmark, there are cases, highlighted in our experiments, where the number of variables is *in the billions*, and the LP solver itself crashes at this scale.

Data Scale Dependency: The sampling algorithm is repeatedly invoked to generate the materialized views, whose sizes are commensurate with the scale of the database. This leads to impractical computational time and storage space overheads for enterprise data volumes.

Inaccuracy in satisfying CCs: The sampling algorithm, due to its probabilistic core, introduces errors into the generation process with respect to constraint satisfaction – this effect is especially prominent for CCs with small output cardinalities.

Limited coverage: The acceptable CCs are restricted to simple range filters of the type

$[low, high)$. Also, only schemas with a tree-structured dependency graph[1] can be handled in the framework.

1.4 Proposed PM Generator

The goal of the PM generator is to address the main limitations of DataSynth, and to create a tool that can take a substantive step forward towards handling the requirements of contemporary database deployments, where scale and efficiency need to be present on an end-to-end basis.

The key contributions of PM are the following:

Region-Partitioning: DataSynth's *grid-partitioning* approach is replaced by a potent *region-partitioning* strategy. A sample region-partitioning for the running example is shown in Figure 1.6, where the numbers now refer to region identifiers. A variable is created for each region, resulting in a total of 15 variables, 9 for R_1 and 6 for R_2, substantially less than the 28 created by DataSynth in Figure 1.5. We describe the algorithm in detail in Section 3 and prove that it is the *optimal* partioning strategy with regard to minimizing the number of variables. Further, as shown in our experiments, the shift from grid-based to region-based partitioning results in *orders-of-magnitude* reduction in the LP complexity. Specifically, referring back to the billions-of-variables situations mentioned above for DataSynth, the corresponding numbers are only a few thousand with PM and the solutions are typically obtained in less than a minute.

[1] A graph where every relation is a node and two nodes are connected with a directed edge if the source nodes holds a foreign-key to the target node.

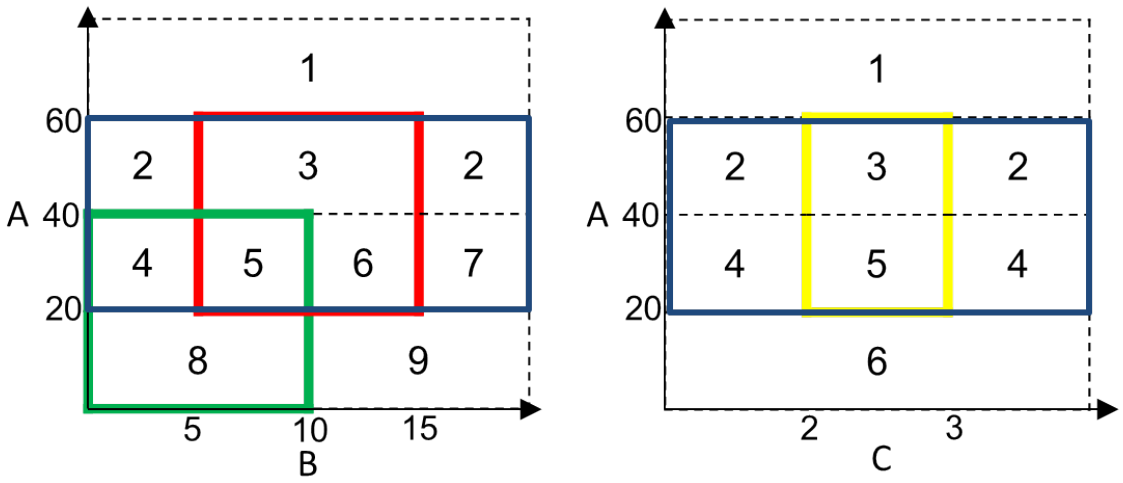


Figure 1.6: Region-Partitioning of PM

Database Summary and Dynamic Generation: A novelty of our data generation algorithm is that it delivers a *database summary* as the output, rather than the static data itself. This summary depends only on the query workload and *not* on the database scale, and is of negligible size. For instance, the database summary corresponding to the running example is shown in Figure 1.7, where entries of the type a - b in the primary key columns, mean that the relation has $b - a + 1$ tuples with values (a, $a + 1$, $a + 2$,...,b) for that column, keeping the other columns unchanged.

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

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

T	
T_pk	C
1–600	0
601–1500	2

Figure 1.7: Example Database Summary

The advantages of a summary-based approach are that in conjunction with the tuple

generator component (Section 5), the database can either be *dynamically* generated in streaming fashion, as might be expected in Big Data applications, or optionally materialized into static relations.

Deterministic View Generation: DataSynth's *sampling-based* approach to data generation is replaced by a *deterministic alignment* strategy. The alignment works at the level of database summaries, and is therefore extremely efficient. Further, it does not suffer the probabilistic errors that affect the sampling approach, and therefore delivers *more accurate* volumetric similarity.

Enhanced Coverage and Evaluation: In PM we have increased the scope of the query workload to include schema that have DAG-structured dependency graphs. Further, filter predicates in *disjunctive normal form (DNF)* are accepted as opposed to only simple range queries.

With regard to empirical analysis, DataSynth had been evaluated on a restricted set of 8 query templates from a 1 GB TPC-H benchmark in [1]. Whereas our experiments here feature as many as 131 queries from the much richer and complex TPC-DS benchmark, operated at 100 GB scale.

Integration with CODD [8, 3]: CODD is a graphical tool through which database environments with desired *meta-data characteristics* can be efficiently simulated without persistently generating and/or storing their contents. The CODD tool supports the transfer of meta-data from the client to the vendor site, and can therefore be used to ensure that the optimizer's plan choices at the client site are replicated at the vendor site. We have integrated PM with CODD, thereby providing an end-to-end system that fully replicates the client data processing environment at the vendor's site.

1.5 Organization

The remainder of this report is organized as follows: In Section 2, the complete architecture of the PM generator is presented and its components enumerated. The theoretical characterization of our region-based LP formulation is presented in Section 3. The database summary generator and the tuple generator are discussed in Sections 4 and 5, respectively. The experimental framework and performance results are reported in Section 6. Our conclusions are summarized in Section 7.

Chapter 2

PM Architecture

This section gives an overview of PM's architecture, with a brief description of its various components and their interactions with the database engine. The entire flow from the client site to the vendor site is shown in Figure 2.1. In this picture, the components coloured in green indicate the new components designed for PM, whereas the orange-colored modules are borrowed from DataSynth and the yellow-colored modules indicate the integration with CODD.

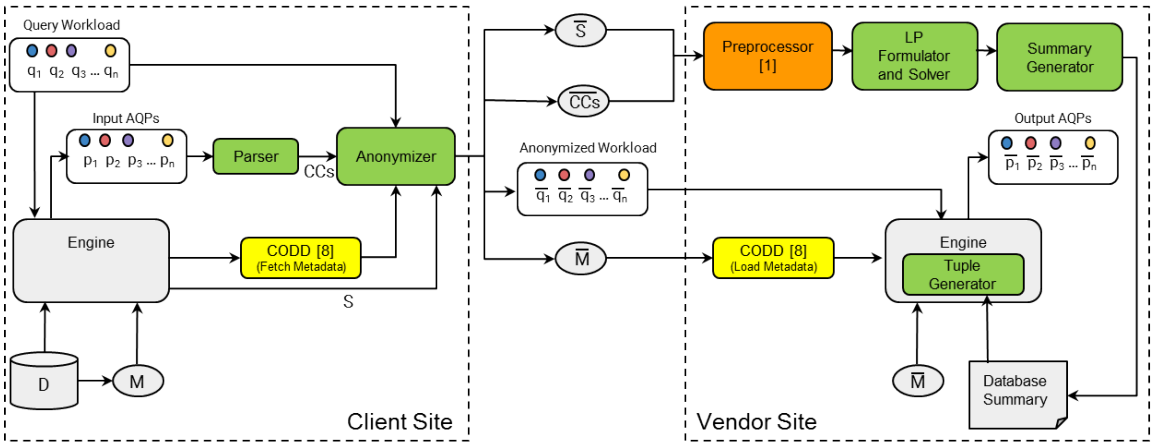


Figure 2.1: PM Architecture

2.1 Client Site

At the client site, PM fetches the schema information and the query workload with the corresponding AQPs from the engine. The metadata is also fetched with the help of CODD[3]. The

AQPs are converted to the set of CCs using a *Parser*. All this information (schema, metadata, queries and CCs) is passed to the *anonymizer* for masking.

In Figure 2.1 we can see that each of the queries $q_1, q_2, q_3, ..., q_n$ are fired on the engine with data D and we get $p_1, p_2, p_3, ..., p_n$ AQPs respectively. The schema information and metadata are represented as S and M respectively. The Parser takes AQPs as the input and returns CCs as the output. The anonymizer masks all the information and the resultant schema information (shown as $\overline{S}$), metadata (shown as $\overline{M}$), the set of queries (shown as $\overline{q_1}, \overline{q_2}, \overline{q_3}, ..., \overline{q_n}$) and masked cardinality constraints (shown as $\overline{CCs}$) are generated and shipped to the vendor's site.

2.2 Vendor Site

The vendor site contains four major modules:

Preprocessor: Here, the schema information and CCs are processed to create the input for the LP Formulator. This component has been borrowed from DataSynth. We briefly explain it here for the sake of completeness. The readers can refer to [1] for more details. The Preprocessor comprises of two sub modules:

- **View Construction[1].** This component takes the database schema as input and creates a *view* $\mathcal{V}_i$ corresponding to every relation $\mathcal{R}_i$. As discussed earlier, creation of views help us to replace the join expressions in the CCs, i.e., once the views are created, each constraint can be re-written as a filter predicate over a single view only. A view $\mathcal{V}_i$ can be considered as a set of non-key attributes that are present in either $\mathcal{R}_i$ or in any other relation on which $\mathcal{R}_i$ depends through referential constraints (both directly or transitively). Dependencies between relations can be computed using the dependency graph. In a dependency graph, a node for every relation is created. A directed edge from a node $\mathcal{R}_i$ to $\mathcal{R}_j$ is added, if $\mathcal{R}_i$ contains a *foreign-key* referencing $\mathcal{R}_j$. Now, a relation $\mathcal{R}_i$ is said to be dependent on relation $\mathcal{R}_j$ if there exists a path from $\mathcal{R}_i$ to $\mathcal{R}_j$ in the dependency graph.

  For example, consider a schema having four relations with the dependency graph as shown in Figure 2.2. A view is created corresponding to every relation. Here, $Catalog\_sales$ view will have non-key attributes of all the relations. $Customer$ view will have non key attributes of $Customer$ and $Customer\_address$ relations. The remaining two views will have the non-key attributes of the corresponding relations.

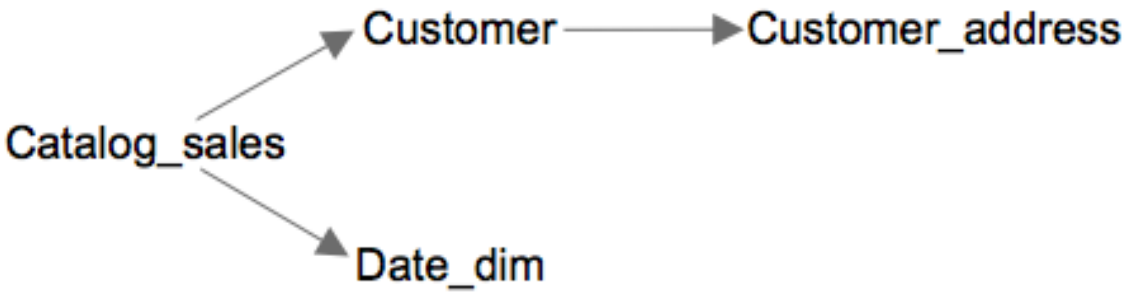


Figure 2.2: Dependency Graph

- **View Decomposition**[1]**.** In general, the number of attributes in a view are many. These lead to a high-dimensional (every attribute is considered a dimension) space problem to be solved. Therefore, the goal of this component is to divide a high-dimensional problem into several low-dimension problems by splitting the view into sub-views. In order to do that, first a *Markov network* is created. This is an undirected graph where a node for each attribute in the view is created. An edge between two nodes are added if the corresponding attributes co-appear in some CC. In order to decompose the view, we need to modify the graph such that it becomes chordal[1] (if not already). To do this, we have implemented the algorithm presented in [7]. Finally, *maximal cliques* are extracted from this graph. Each maximal clique is a sub-view.

  For example, let there be a view whose Markov network for a set of constraints is as shown in Figure 2.3a. This network is not chordal, therefore we add an edge to make it chordal. Resultant graph is as shown in Figure 2.3b. Finally, maximal cliques are extracted from this. We therefore obtain two sub-views as shown in Figure 2.3c.

[1] A graph is chordal if each cycle of length 4 or more has a chord; a chord is an edge joining two non-adjacent nodes of a cycle.

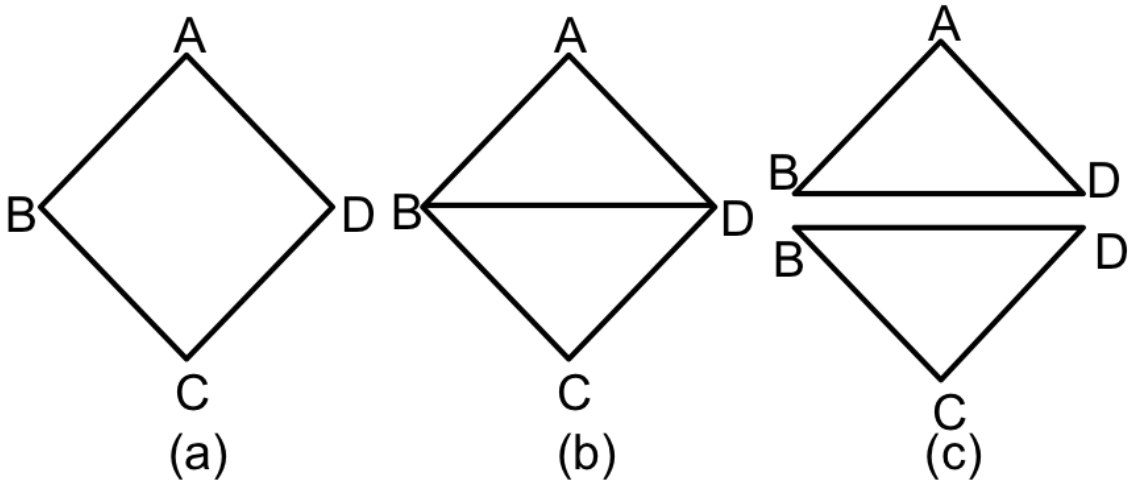


Figure 2.3: View Decomposition

LP Formulator and Solver: For each view that we obtain from the Preprocessor, the LP Formulator constructs a LP by taking as input the corresponding set of sub-views and applicable CCs. The formulation algorithm is explained in detail in Section 3.

Once this is done, the LP is passed on to the Solver. We have used Z3[6], a contemporary SMT solver developed by Microsoft to solve the LP. The solver takes the LP constraints as the input and gives one of the feasible solutions as the output.

Summary Generator: This component takes the LP solution for each view as the input, and generates the database summary from it. This component is also responsible for ensuring that the generated summary obeys referential integrity. We explain this component in detail in Section 4.

Tuple Generator: Tuple Generator resides inside the database engine. It ensures that whenever a query is fired, the executor does not fetch the data from the disk, rather the Tuple Generator supplies the data *on-the-fly* using the database summaries generated earlier. The details of this component are given in Section 5.

Note that we use CODD[3] to load the masked metadata ($\overline{M}$) on to the vendor's engine to ensure that when the masked queries are fired, the optimizer chooses the plans which are structurally the same as the ones that were present on the client's site.

Chapter 3

LP Formulation

The preprocessing step before LP formulation, which consists of constructing views and decomposing them into sub-views, has been borrowed from DataSynth. During view decomposition, a view is divided into sub-views using *view graphs*. A view graph is constructed by creating a node for each attribute and an edge between two attributes is added if the corresponding attributes co-appear in some CC. The sub-views are the maximal cliques extracted from this graph after making it chordal.

The LP formulation module treats each view as a set of its sub-views. It takes the CCs applicable on each of the sub-views and generates solution for each sub-view which is merged to give a view solution in the next step. Here CCs are written as LP constraints. Since the sub-views can be overlapping, additional LP constraints are added to keep the sub-views consistent with each other. We start by presenting the mathematical basis for our LP formulation, followed by a description of the algorithm for generating an LP of a small size.

3.1 Mathematical Basis for Small Size LP

Let n denote the number of attributes in the given relation. For $1 \leq i \leq n$, let $\mathcal{D}_i$ denote the domain of the ith attribute. For simplicity, we assume that attribute i is numeric, so that $\mathcal{D}_i$ is a subset of real numbers. [1] Let $\mathcal{D}$ denote the universe $\mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \mathcal{D}_n$.

We are given a set of m cardinality constraints that the relation satisfies. For $1 \leq j \leq m$, each constraint C_j is a pair $\langle \sigma_j, k_j \rangle$ where σ_j is a selection predicate, k_j is a non-negative integer, and which means that the number of tuples satisfying predicate σ_j is equal to k_j. We assume that each predicate σ_j is in disjunctive normal form (DNF).

[1] Our technique based on equivalence classes applies to more general attribute types.

3.1.1 Simple LP Formulation

Let us first consider a simple way of formulating an LP that satisfies all CCs. For each tuple $t \in \mathcal{D}$, we have a variable x_t that denotes the number of copies of this tuple in the relation. The LP shown in Figure 3.1 ensures that the resulting relation satisfies all CCs, including a constraint on the total size of the relation R. The problem with this formulation is that the number of variables in the resulting LP is as large as the size of the universe $\mathcal{D}$. Hence, it is infeasible to work directly with this formulation.

$$\text{For each } t \in \mathcal{D}, x_t \geq 0$$

$$\left[\sum_{t \in \mathcal{D}} x_t\right] = k$$

$$\text{For each } j, 1 \leq j \leq m, \left[\sum_{t:\sigma_j(t)=true} x_t\right] = k_j$$

Figure 3.1: Simple LP formulation for constraint set $\mathbb{C}$ for a relation whose total size must be equal to k.

3.1.2 Reducing the Size of the LP

We now present an LP with fewer variables. We first note that in the simple formulation, the variables corresponding to two points $t_1, t_2 \in \mathcal{D}$ that behave identically with respect to a constraint C_j (i.e. $\sigma_j(t_1) = \sigma_j(t_2)$) can be combined together as $(x_{t_1} + x_{t_2})$, for the purposes of satisfying constraint C_j. If this is true with respect to every constraint, i.e. for $j = 1 \ldots m$, $\sigma_j(t_1) = \sigma_j(t_2)$, then there is no need to treat t_1 and t_2 separately, i.e. the two variables x_{t_1} and x_{t_2} can be merged into a single variable $(x_{t_1} + x_{t_2})$ in every equation, leading to fewer variables in the LP. By repeating this variable merging process recursively until it is no further possible, we arrive at a vastly reduced LP. To formalize this idea, we start with the following definitions. For constraint C and point $t \in \mathcal{D}$, let $C(t)$ be an indicator variable:

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

Definition 3.1 *For two points $p, q \in \mathcal{D}$ and a set of constraints $\mathbb{C}$, we say $pR^{\mathbb{C}}q$ if for each $C \in \mathbb{C}$, $C(p) = C(q)$.*

Lemma 3.1 *For set of constraints $\mathbb{C}$, relation $R^{\mathbb{C}}$ is an equivalence relation on $\mathcal{D}$.*

Proof: 1 We first note that the relation is reflexive, since for each $p \in \mathcal{D}$, $pR^{\mathbb{C}}p$. Similarly, it can be easily seen that the relation is symmetric. For transitivity, suppose that for $p, q, r \in \mathcal{D}$, $pR^{\mathbb{C}}q$ and $qR^{\mathbb{C}}r$. Note that for each $C \in \mathbb{C}$, it must be true that $C(p) = C(q)$ and $C(q) = C(r)$. It must be true that $C(p) = C(r)$, showing that the relation is transitive.

Definition 3.2 *Given a set of constraints $\mathbb{C}$, a partition*[1] *$\mathbb{P}$ of $\mathcal{D}$ is said to be a* valid partition *if for each block $b \in \mathbb{P}$ and any two points $p, q \in b$, it must be true that $pR^{\mathbb{C}}q$.*

Once we obtain a valid partition $\mathbb{P}$ of $\mathcal{D}$ subject to $\mathbb{C}$, the LP can be formulated as shown in Figure 3.2. Instead of a variable for each point $t \in \mathcal{D}$, there is now a single variable x_b for each block $b \in \mathbb{P}$, representing the number of tuples lying in this block.

$$\text{For each } b \in \mathbb{P}, x_b \geq 0$$

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

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

Figure 3.2: Reduced LP formulation for constraint set $\mathbb{C}$ for a relation whose total size must be equal to k.

Note that the total number of variables in the reduced LP shown in Figure 3.2 is equal to the number of blocks in the partition $\mathbb{P}$ and is much smaller than the number of variables in the original LP, shown in Figure 3.1. Since the complexity of solving an LP increases with the number of variables in it, we desire an LP with as few variables as possible, and hence we desire a valid partition of $\mathcal{D}$ with as few blocks as possible. We say that a valid partition $\mathbb{P}$ with respect to $\mathbb{C}$ is an *optimal partition*, if it has the smallest number of blocks from among all valid partitions of $\mathcal{D}$ with respect to $\mathbb{C}$.

Lemma 3.2 *The quotient set* [2] *of $\mathcal{D}$ by $R^{\mathbb{C}}$ is the unique optimal partition of $\mathcal{D}$ with respect to constraint set $\mathbb{C}$.*

[1] A partition of a set $\mathcal{D}$ is a set of subsets of $\mathcal{D}$ such that every element $x \in \mathcal{D}$ is in exactly one of these subsets.

[2] The set of equivalence classes of a set $\mathcal{D}$ with respect to an equivalence relation Q is called the quotient of $\mathcal{D}$ by Q.

Proof: **2** Let $\mathbb{P}_1$ denote the quotient set of $\mathcal{D}$ by $R^{\mathbb{C}}$. Clearly $\mathbb{P}_1$ is a valid partition from the definition of equivalence class. Further, let $\mathbb{P}_2$ ($\neq \mathbb{P}_1$) denote a valid partition such that $|\mathbb{P}_2| \leq |\mathbb{P}_1|$. This implies that there exist two points $p, q \in \mathcal{D}$ such that p and q are in different blocks in $\mathbb{P}_1$ but in same block in $\mathbb{P}_2$. Now, if p and q belong to different blocks in $\mathbb{P}_1$, then they are not related (by definition of equivalence class). But, in $\mathbb{P}_2$ they are present in the same block, which implies $\mathbb{P}_2$ cannot be a valid partition. Hence, $\mathbb{P}_1$ is the smallest valid partition or in other words, it is the optimal partition.

3.2 Deriving the Optimal Partition

We now present an algorithm to derive the optimal partition of $\mathcal{D}$ with respect to $\mathbb{C}$. Each constraint $C \in \mathbb{C}$ is in DNF, and is expressed as the union of many smaller "sub-constraints". Each sub-constraint is the conjunction of many per-attribute constraints, and each per-attribute constraint is a constraint on the values that a single attribute could take. For example, the following constraint on two attributes A_1, A_2: $((A_1 \leq 20) \wedge (A_2 > 30)) \vee (A_1 > 50)$, is divided into two sub-constraints: $(A_1 \leq 20) \wedge (A_2 > 30)$ and $(A_1 > 50)$.

Definition 3.3 *For a sub-constraint C and dimension i, let C^i denote the restriction (projection) of C to dimension i. Further, let $C_1^i = \bigwedge_{k=1\ldots i} C^k$ denote the restriction of C to dimensions $1, 2, \ldots, i$. For instance, if $C = (A_1 \geq 1 \wedge A_1 \leq 2 \wedge A_2 \geq 4 \wedge A_2 \leq 5 \wedge A_3 > 6)$ then $C^2 = (A_2 \geq 4 \wedge A_2 \leq 5)$, and $C_1^2 = A_1 \geq 1 \wedge A_1 \leq 2 \wedge A_2 \geq 4 \wedge A_2 \leq 5$.*

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

Definition 3.4 *A constraint C is said to split a block $b \subset \mathcal{D}$ if there exist two points $p_1, p_2 \in b$ such that $C(p_1) = true$ and $C(p_2) = false$. If C splits b, then refining b by C partitions b into two subsets $b^+(C) = \{x \in b | C(x) = true\}$ and $b^-(C) = \{x \in b | C(x) = false\}$.*

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

Input: Universe $\mathcal{D}$, set of sub-constraints $\mathbb{C}$

Output: An optimal partition $\mathbb{P}$ of $\mathcal{D}$ subject to set of sub-constraints $\mathbb{C}$

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

Lemma 3.3 *Algorithm Optimal Partition($\mathcal{D}, \mathbb{C}$) returns the optimal partition of $\mathcal{D}$ with respect to set of sub-constraints $\mathbb{C}$.*

Proof: **3** For $1 \leq i \leq n$, let $\mathbb{C}_1^i = \{C_1^i | C \in \mathbb{C}\}$. We show by induction on i that after the ith iteration of the outermost for loop in the algorithm, $\mathbb{P}^i$ contains an optimal partition of $\mathcal{D}$ with respect to $\mathbb{C}_1^i$. Since $\mathbb{C}_1^n = \mathbb{C}$, it follows that after n iterations, $\mathbb{P}^n$ contains an optimal partition of $\mathcal{D}$ with respect to $\mathbb{C}$. We consider $i = 0$ as the base case. We can view the set $\mathbb{C}_1^0$ as a set of "always true" constraints, and hence $\mathbb{P}^0$, which consists of only one element, $\mathcal{D}$, is the smallest partition that satisfies $\mathbb{C}_1^0$.

For each constraint $C \in \mathbb{C}$, $C_1^i = C_1^{i-1} \wedge C^i$, hence C_1^i is either identical to C_1^{i-1} or is more restrictive. It follows that an optimal partition of $\mathcal{D}$ with respect to C_1^i must be a refinement of an optimal partition of $\mathcal{D}$ with respect to C_1^{i-1} [1]

For the inductive step, suppose that for $i > 0$, $\mathbb{P}^{i-1}$ is the optimal partition of $\mathcal{D}$ with respect to $\mathbb{C}_1^{i-1}$. We consider two cases. In the first case, consider a block $b \in \mathbb{P}^{i-1}$ such that b was not split by C^i, for any $C \in \mathbb{C}$. Then it can be seen that b must be an element of the

[1] We say that a partition $\mathbb{P}_1$ refines another partition $\mathbb{P}_2$ if for each block $b_1 \in \mathbb{P}_1$, there is a block $b_2 \in \mathbb{P}_2$ such that $b_1 \subseteq b_2$.

optimal partition of $\mathcal{D}$ with respect to $\mathbb{C}_1^i$. Note that in the algorithm, such a block b is added in its entirety to M' for each constraint, and finally b will be preserved in M and in $\mathbb{P}^i$. In the second case, consider a block b which is split by a set of constraints $\{C^i | C \in B\}$, where here $B \subset \mathbb{C}$. Consider the constraints in B in some order. In processing the first constraint $C \in B$, C^i will cause b to be partitioned into a set of smaller blocks, and these will further partitioned (recursively) by later constraints. Suppose that $B(b)$ denotes the set of blocks obtained thus starting from b. We note the following: (1) For any two distinct blocks $b_1, b_2 \in B(b)$ and points $p_1 \in b_1$ and $p_2 \in b_2$, it must be true that p_1 is not related to p_2 by relation $R^{\mathbb{C}_1^i}$ – blocks b_1 and b_2 must have resulted from splitting along some constraint in B. (2) For any block $b' \in B(b)$ and any two points $p, q \in b'$, it must be true that $pR^{\mathbb{C}_1^i}q$. Suppose this was not the case, then the block b' would have been split by some constraint along dimension i. This proves the inductive step, that $\mathbb{P}^i$ is the optimal partition of $\mathcal{D}$ with respect to set of sub-constraints $\mathbb{C}_1^i$.

From Sub-Constraints to DNF Constraints So far, we have assumed a set of sub-constraints, i.e. each is a conjunction of many per-attribute constraints. Next, we show how to extend this to a set of constraints in DNF. Given a set $\mathbb{C}$ of constraints in DNF, we generate the set of sub-constraints resulting from the constraints in $\mathbb{C}$ to form a new set of sub-constraints $\mathbb{C}'$. We then construct the optimal partition $\mathbb{P}'$ of $\mathcal{D}$ subject to $\mathbb{C}'$ using Algorithm Optimal Partition. Note that each sub-constraint $C' \in \mathbb{C}'$ is stricter than the constraint in $\mathbb{C}$ that it was derived from. Hence it follows that $\mathbb{P}'$ is a valid partition of $\mathcal{D}$ subject to $\mathbb{C}$, but not necessarily an optimal partition. The optimal partition subject to $\mathbb{C}$ is derived as follows.

1. For each block $b \in \mathbb{P}'$, assign a label $\ell(b)$ equal to the set of all constraints in $\mathbb{C}$ that a point in block b satisfies. Note that this can be computed using a single pass through each constraint in $\mathbb{C}$, and evaluating the constraint over an arbitrary point $p \in b$.

2. Let $\ell(C')$ denote the set of all distinct labels $\{\ell(b) | b \in \mathbb{P}'\}$. For each $l \in \ell(C')$, merge all blocks in $\mathbb{P}'$ whose label equals l into a single block. Return the resulting partition $\mathbb{P}$ of $\mathcal{D}$.

We claim that $\mathbb{P}$ is an optimal partition of $\mathcal{D}$ subject to $\mathbb{C}$. To see this, note that no two blocks in $\mathbb{P}$ can be merged together to keep the partition valid with respect to $\mathbb{C}$ – since two distinct blocks in $\mathbb{P}$ have different labels, it follows that there is at least one constraint $C \in \mathbb{C}$ such that one of the two blocks satisfies the constraint while the other does not. To see that $\mathbb{P}$ is a valid partition with respect to $\mathbb{C}$, note that for any block $b \in \mathbb{P}$ and two points $p, q \in b$, p and q satisfy an identical set of constraints within $\mathbb{C}$, and hence $pR^{\mathbb{C}}q$.

Consistency Constraints Like in [1], we need to put additional consistency constraints. Since we applied CCs on sub-views rather than views, it is possible that for two sub-views whose attribute sets are not disjoint, the data distributions for the common attribute(s) may be different. To ensure same distributions for common attribute(s) across sub-views, we may need to further refine the partition generated from the above procedure, and add additional LP constraints.

We explain this by the following example. Consider a pair of sub-views V_1 and V_2 with attributes $\mathbb{A}_1$ and $\mathbb{A}_2$ respectively, such that $\mathbb{A}_1 \cap \mathbb{A}_2 \neq \emptyset$. Further, let $\mathcal{D}^1 = \prod_{i \in \mathbb{A}_1} \mathcal{D}_i$, $\mathcal{D}^2 = \prod_{j \in \mathbb{A}_2} \mathcal{D}_j$ and $\mathcal{D}^{1,2} = \prod_{k \in \mathbb{A}_1 \cap \mathbb{A}_2} \mathcal{D}_k$. Processing CCs on V_1 and V_2 leads to LP constraints on $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively, and suppose the respective partitions obtained are $\mathbb{P}_1$ and $\mathbb{P}_2$. In order to keep the two sub-views consistent, we first have to ensure that the region boundaries for the partitions of V_1 and V_2 in $\mathcal{D}^{1,2}$ are consistent with each other. In order to achieve this, we need to refine $\mathbb{P}_1$ and $\mathbb{P}_2$ so that they have common boundaries along dimensions $\mathbb{A}_1 \cap \mathbb{A}_2$. We consider the union of the "split points" of $\mathbb{P}_1$ and $\mathbb{P}_2$ along dimensions $\mathbb{A}_1 \cap \mathbb{A}_2$, i.e., values along which the universe is split along these dimensions by the partitions. For each block in $\mathbb{P}_1$ (or in $\mathbb{P}_2$), we refine this block until it no longer crosses such a split point. Then we add constraints so that the sizes of sub-views V_1 and V_2 along the different regions of $\mathcal{D}^{1,2}$ that result from the split points are equal to each other.

For the example we considered in Figure 1.6, the LP constraints can be written as:

$$\begin{aligned} x_4 + x_5 + x_8 &= 10K \\ x_3 + x_5 + x_6 &= 20K \\ x_2 + x_3 + ... + x_7 &= 50K \\ x_1 + x_2 + ... + x_9 &= 80K \\ y_3 + y_5 &= 30K \\ y_2 + y_3 + y_4 + y_5 &= 50K \\ y_1 + y_2 + ... + y_6 &= 80K \end{aligned}$$

And the four consistency constraints are also be written in the same way as we wrote earlier.

$$\begin{aligned} x_1 &= y_1 \\ x_2 + x_3 &= y_2 + y_3 \\ x_4 + x_5 + x_6 + x_7 &= y_4 + y_5 \\ x_8 + x_9 &= y_6 \end{aligned}$$

Chapter 4

Database Summary Generator

This component takes the LP solution for each view as the input, and generates the database summary from it. The LP solution gives us the row cardinalities for all the regions in the partitions at a sub-view level. Let b_i^j represent the i^{th} region of the j^{th} sub-view, and let the value of the associated variable be k_i^j. So the LP solution is a vector of the form $\langle b_i^j, k_i^j \rangle, \quad \forall i, j$.

Recall that a sub-view is a projection of the view along some dimensions. Therefore, in order to generate the view, we need to map the sub-view solution obtained from the solver back to the original space. Accordingly, the summary generator component in PM is responsible for the following three tasks:

1. Constructing a solution for complete views
2. Making views consistent with respect to each other
3. Extracting relation summaries from view summaries

4.1 Constructing Solution for the View

As discussed previously, in DataSynth this component was implemented using a sampling algorithm. In marked contrast, PM *deterministically* generates the view solution – this approach permits us to operate purely in the summary space, and results in elimination of the time and space overheads incurred by DataSynth.

In order to merge the sub-view solutions to obtain the collective solution for the complete view, we first order the sub-views according to an ordering algorithm, and then iteratively build the view-solution by aligning and merging the next sub-view solution in the order. Let $\mathbb{S}$ denote the list of all sub-view solutions, and *viewSol* be the final view solution that we wish

to compute. Algorithm 2 describes the process for constructing *viewSol* from $\mathbb{S}$.

Algorithm 2: View Solution Construction

```
1 𝕊 ← OrderSubViews(𝕊)
2 viewSol ← ∅
3 for s in 𝕊 do
4     viewSol, s ← Align(viewSol, s)
5     viewSol ← Merge(viewSol, s)
```

We now describe in turn the ordering, aligning and merging algorithms.

4.1.1 Sub-view Ordering

Ordering is implemented through a greedy iterative algorithm where we can start with any sub-view. Subsequently, at iteration i, let the set of visited sub-views until now be $\mathbb{S}$. A sub-view s from outside this set can be chosen only if s satisfies the condition that on removing the common vertices between s and $\mathbb{S}$ in the view graph, there exists no path between the remaining vertices of s and the remaining vertices of $\mathbb{S}$.

For example, consider the markov network for a view be as shown in Figure 4.1. As we can see, this has three cliques: AEC, ECD and BED. Here, if we start with sub-view AEC, then we are bound to choose ECD before BED because on removing the common attribute E between BED and AEC, there still exists the edge DC connecting the two remaining components. Note that, ECD satisifies the required condition. Once ECD is chosen, then BED can be picked next.

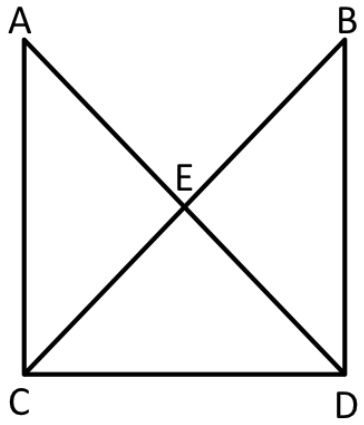


Figure 4.1: Markov Network

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

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

(a) Sub-view Solution

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

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

(b) View Alignment

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

(c) Merged View Solution

Figure 4.2: Align and Merge Example

4.1.2 Aligning

After obtaining the merge order, in every iteration we merge the next sub-view solution (say s) in the sequence to the current *viewSol*, after a process of alignment. The alignment algorithm is a two step exercise. We show the two steps for the running example of Figure 1.6. This example had two sub-views. We show how to align these:

1. First, we order *viewSol* and s on the common set of attributes. For instance, let the

solution to the LP for the example be:

$$x_1 = 30K \qquad x_2 = 20K \qquad x_3 = 10K$$
$$x_5 = 10K \qquad x_7 = 10K$$
$$y_1 = 30K \qquad y_3 = 30K \qquad y_4 = 20K$$

with all other variables being zero. This solution can be represented as shown in Figure 4.2a, where both the sub-view solutions are ordered on the common attribute A and NumTuples represent the row cardinalities.

2. Our addition of consistency constraints during LP formulation ensures that the distribution of tuples along the common set of attributes is the same in the various sub-views. Therefore it is easy to see that the sum of NumTuples values in any interval of the common attribute is the same for the solutions under alignment. For example, in Figure 4.2a, the total number of tuples with $A = [40, 60)$ is 30K in both solutions. Likewise, other values in A also have matching total number of tuples across the solutions.

   The align step splits the rows in these solutions such that the corresponding rows in both solutions have the same number of tuples. Let $viewSol_i$ and s_j represent the i^{th} and j^{th} rows in $viewSol$ and s, respectively. Further, let k_i and l_j represent the NumTuples values in $viewSol_i$ and s_j, respectively. Given this, the Align algorithm is as shown in Algorithm 3.

Algorithm 3: Alignment Procedure

```
1 i ← 0,   j ← 0
2 while viewSol_i exists do
3 |   if k_i < l_j: SPLIT(s, j, k_i)
4 |   else if k_i > l_j: SPLIT(viewSol, i,
  |    l_j)
5 |   i ← i + 1,   j ← j + 1
```

The split procedure takes a LP solution as input along with a row number and an integer. Say the input solution is X, the row number is r and the integer is $card$. Let the NumTuples attribute value in r^{th} row of X be n. This procedure returns the solution X after splitting its r^{th} row (X_r, n) into $X_r : (X_r, card)$ and $X_{r+1} : (X_r, n - card)$.

The sub-view solutions of Figure 4.2a are shown in Figure 4.2b after undergoing the alignment process. We can see here that both solutions have identical NumTuples in the corresponding rows.

4.1.3 Merging

This is the last step where we simply merge the two solutions obtained after alignment through a join of the two solutions on the common attributes. For example, the aligned solutions of Figure 4.2b are merge-joined to deliver the final view solution of Figure 4.2c.

4.2 Making Views Consistent

Using the above procedure, we can obtain all the view solutions. However, since each solution was obtained independently, there could be inconsistencies across the solutions. For example, in Figure 1.1, since *R_view* consists of attributes borrowed from *S_view* and *T_view*, it should not feature any values that are not present in the corresponding attributes of these latter views.

Since the dependencies between views can be direct and transitive, we need to follow an ordering in which the views are made consistent. For this purpose, we carry out a topological sort on the dependency graph and iteratively make the current view consistent with its predecessors. Since a topological sort is employed, we can handle dependency graphs that are DAGs unlike DataSynth which was restricted to tree traversals.

To make a pair of views $\mathcal{V}_i$ and $\mathcal{V}_j$ consistent with each other, where $\mathcal{V}_i$ is dependent on $\mathcal{V}_j$, we iterate over the rows in the view solution of $\mathcal{V}_i$ and look for the value combination that each row has for the attributes that $\mathcal{V}_i$ borrowed from $\mathcal{V}_j$. If that value combination is not present in the view solution of $\mathcal{V}_j$, we add a new row in its solution with the corresponding NumTuples attribute set to 1. This therefore leads to an additive error in the total number of tuples in the view as compared to the original AQP in the client. But we hasten to add that the error is a fixed number of rows, determined by the nature of the constraints and the LP solution, and *not* by the data scale. Therefore, at Big Data volumes, the relative error can be expected to be miniscule.

The view integration component is present in DataSynth as well, but since the view solutions comprised of complete database instantiations, and not summaries, the time and space overheads incurred for making the views consistent were comparatively huge.

4.3 Constructing Relation Summary

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

For the attributes that are common between the summarized relation set and the corresponding view solution set, the value combinations and corresponding cardinalities are directly borrowed. What remains are the foreign key attributes. For filling a foreign key attribute fk, we first need to see the view corresponding to the relation that the foreign key refers to. Say the view thus obtained is $\mathcal{V}_j$. Now, to fill the fk value in the r^{th} row of $\widetilde{\mathcal{R}_i}$, we first extract the value combination in the r^{th} row of view solution of $\mathcal{V}_i$. From this value combination, we project the attributes corresponding to $\mathcal{V}_j$, and denote it by v. Now, we iterate over the solution set of $\mathcal{V}_j$ and compute the cumulative sum of the cardinality entries till we reach v. This sum gives us the fk value corresponding to the r^{th} row of $\widetilde{\mathcal{R}_i}$, and we thus obtain $\widetilde{\mathcal{R}_i}$ for each relation $\mathcal{R}_i$. The set of relation summaries gives us the database summary. We already saw a sample database summary in Figure 1.7. (The figure shows the PK columns instead of the number of tuples purely for simplicity.)

Chapter 5

Tuple Generator

The Tuple Generator component resides inside the database engine, and needs to be explicitly incorporated in the engine codebase by the vendor. As a proof of concept, we have implemented it for the Postgres engine by adding a new feature called *datagen*. On enabling this feature, whenever a query is fired, the executor does not fetch the data from the disk but is supplied by the Tuple Generator instead in an *on-demand* manner, using the database summaries generated earlier.

Each row in the relation summary has a value combination and an associated NumTuples entry. We will consider the pk values to be the row numbers of the relation. Therefore, to get the r^{th} tuple of a relation $\mathcal{R}_i$, the pk is chosen as r and the rest of the attributes come from the relation summary. We iterate over the rows of $\widetilde{\mathcal{R}_i}$ and take the cumulative sum of the NumTuples entries until the sum exceeds r. Say the cumulative sum exceeds the value r in j^{th} row of $\widetilde{\mathcal{R}_i}$. So the rest of the attributes of the r^{th} tuple are precisely the same as those present in the j^{th} row of $\widetilde{\mathcal{R}_i}$. For example, the 120^{th} row of S relation in Figure 1.7, would be $\langle 120, 20, 15 \rangle$.

Note that this form of tuple generation is found to be much faster than DataSynth because they assign values to each attribute *conditioned* on the values assigned to the prior attributes. The rate of tuple generation is reported in Section 6.5.

Chapter 6

Experiments

Having presented the salient features of the PM tool, we now move on to its empirical evaluation. PM is completely written in Java, running to over 15K lines of code. We use the popular Z3 [6] for solving LPs. For a fair comparison of PM's performance against DataSynth, we extended the implementation of DataSynth to also handle constraints in DNF.

Database Environment The TPC-DS [10] decision-support benchmark is used as the foundation for our experiments. The default database size was 100 GB on which we executed a complex query workload, WL_c, featuring 131 queries. These queries were created by mildly modifying the native TPC-DS queries so that they were compatible with our assumptions on cardinality constraints.

While PM was able to comfortably handle the above scenario, the same was not true for DataSynth, and we therefore had to scale down the database size and/or simplify the query workload for it to reach completion in some cases. The simplified workload is denoted by WL_s in the sequel.

System Environment We used PostgreSQL v9.3 [11] engine for our experiments, with the hardware plaform being a vanilla HP workstation with a 3.2 GHz 16-core processor, 32 GB memory and a solid-state hard drive.

6.1 Quality of Volumetric Similarity

We begin by investigating how closely volumetric similarity, as modeled by the operator output cardinalities, is achieved between the client and vendor sites. As discussed earlier, DataSynth

incurs errors in satisfying cardinality constraints due to two reasons: (1) the probabilistic sampling technique, and (2) maintenance of referential integrity. In contrast, PM incurs only the referential integrity errors.

The first of these two sources is dependent on the data scale while the second error is dependent only on the nature of CCs and the quality of the LP solution. This means that, on increasing the scale of the data, the sampling errors (in absolute terms) increase while the integrity errors remain constant.

However, the relative percentage error (RE) decreases with increasing the scale. We define RE as follows:

$$\text{RE} = \frac{|\text{ observed cardinality - actual cardinality }|}{\text{actual cardinality}} \text{ x } 100$$

When DataSynth and PM were run on the 100 GB database with the WL_s workload, 311 cardinality constraints were obtained from the AQPs (the number of *variables* in the constraints were, of course, hugely different for the two techniques, as explained in Section 6.2). For the databases materialized from these constraints, we have shown in Figure 6.1, the percentage of CCs that fall within a given relative error of volumetric similarity.

We have plot the percentage of cardinality constraints against RE suffered in meeting these constraints. Out of the 311 CCs, the plot includes the 271 constraints that had the required output cardinality > 70. For the remaining 40 constraints, we computed the absolute error (AE), computed as |observed card. - actual card.|, since RE magnifies even for small absolute errors. The maximum AE for remaining constraints was 512 in DataSynth and 38 in PM while the average AE was 69.3 in DataSynth and 4.1 in PM.

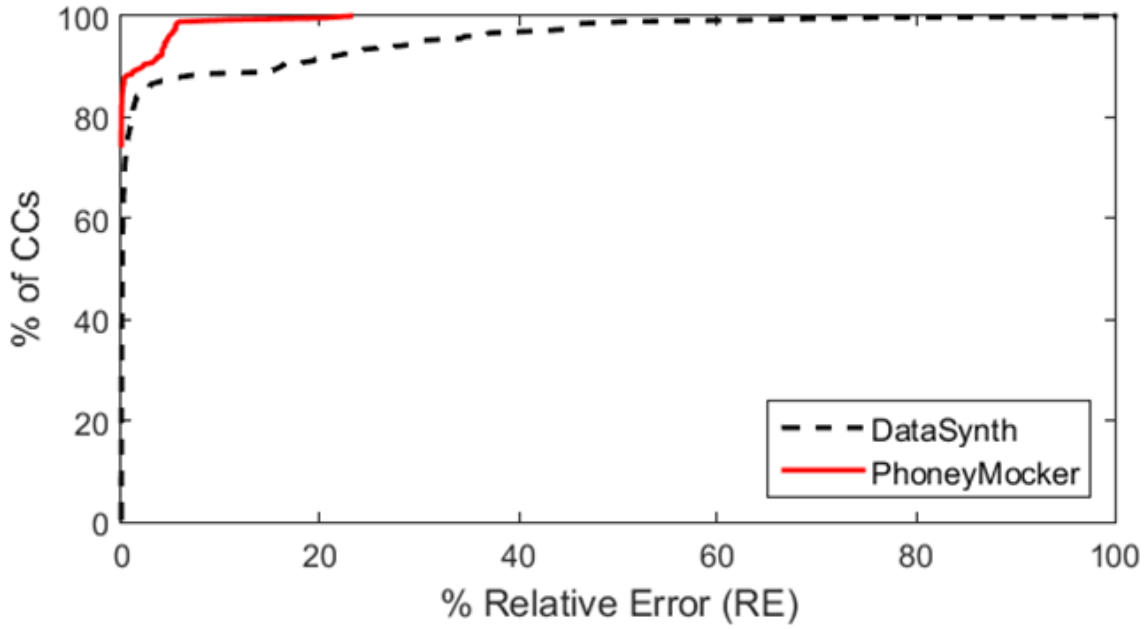


Figure 6.1: Quality of Volumetric Similarity

From the plot, it can be seen that PM satisfies virtually *all* the CCs within a relative error of 10%, whereas DataSynth goes up to more than 60% relative error to achieve a similar CC coverage.

As an aside, it is interesting to note that DataSynth has to contend with both *negative* and *positive* relative errors, due to its sampling core – in fact, we found that about one-third of the CCs suffered negative relative errors. In contrast, PM only generates positive errors due to the addition of extra tuples that are required to satisfy referential integrity. From a practical standpoint, it is perhaps preferable to have positive errors since they induce greater stress on the data processing elements in the engine.

Finally, even with regard to referential integrity alone, we find that the number of extra tuples that are required to be added by PM are substantially smaller than those injected by DataSynth. This is because the integrity errors are *amplified* by the impact of the sampling errors. This effect is quantified in Figure 6.2, where the number of extra tuples inserted is plotted on a log-scale for representative TPC-DS tables. We see here that PM is often an order-of-magnitude smaller with regard to the addition of these spurious tuples.

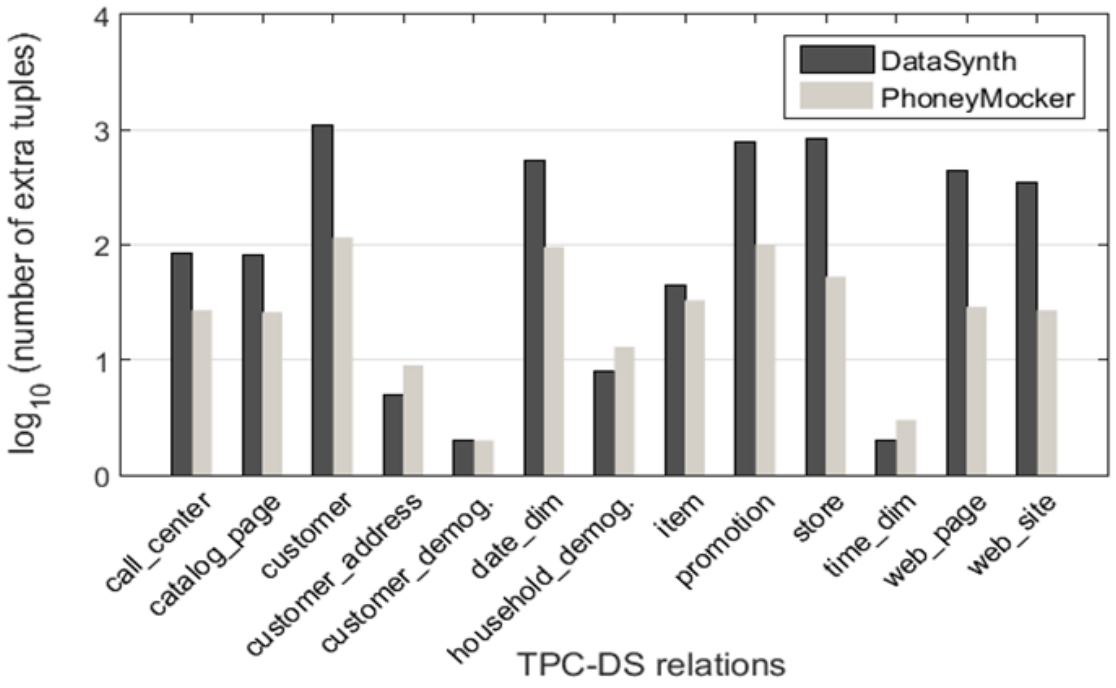


Figure 6.2: Extra tuples for Referential Integrity

Also note that the sampling errors in DataSynth leads to negative errors also. By negative errors we mean that the actual card. is greater than the observed card. for the CC. The RI errors are always additive in nature, which are comparatively better than the negative errors because the negative errors underplay the volumetric flow, which may prevent the volume related performance bugs from surfacing. In our experiments we found, DataSynth has negative errors for 37% of the CCs.

As we mentioned earlier that RI errors are absolute in nature. These are the errors that the dimension tables incur if the fact tables (who possess foreign key referring dimension tables) end up getting a value combination that is not consistent to the dimension table. In our experiments we also found that the errors due to sampling technique also escalates the RI errors. We show this observation in Figure 6.2. It shows the comparison of the number of extra tuples added to the dimension tables in the outputs generated from workload-1. We can easily see that PM, in general, has lesser number of extra tuples to be added.

6.2 Scalability with Workload Complexity

We now turn our attention to comparison of the complexity of the underlying LP that is formulated in PM and DataSynth. Since the LP complexity is proportional to the number of

variables in the problem, we compare this number for the two techniques. Further, since the LP complexity is independent of the database size, we show the comparison only for the 100 GB instance.

The number of LP variables for a representative set of TPC-DS relations, including the major fact and dimension tables (catalog_sales, store_sales, item) is captured, on a log-scale, in Figure 6.3 for the WL_c workload. We observe here that the LPs formulated using the region-partitioning strategy in PM have *several orders of magnitude* fewer variables than the corresponding LPs derived from the grid-partitioning in DataSynth. For instance, consider the catalog_sales table – the number of variables created by DataSynth was almost 5.5 million, which is reduced to 1620 by PM.

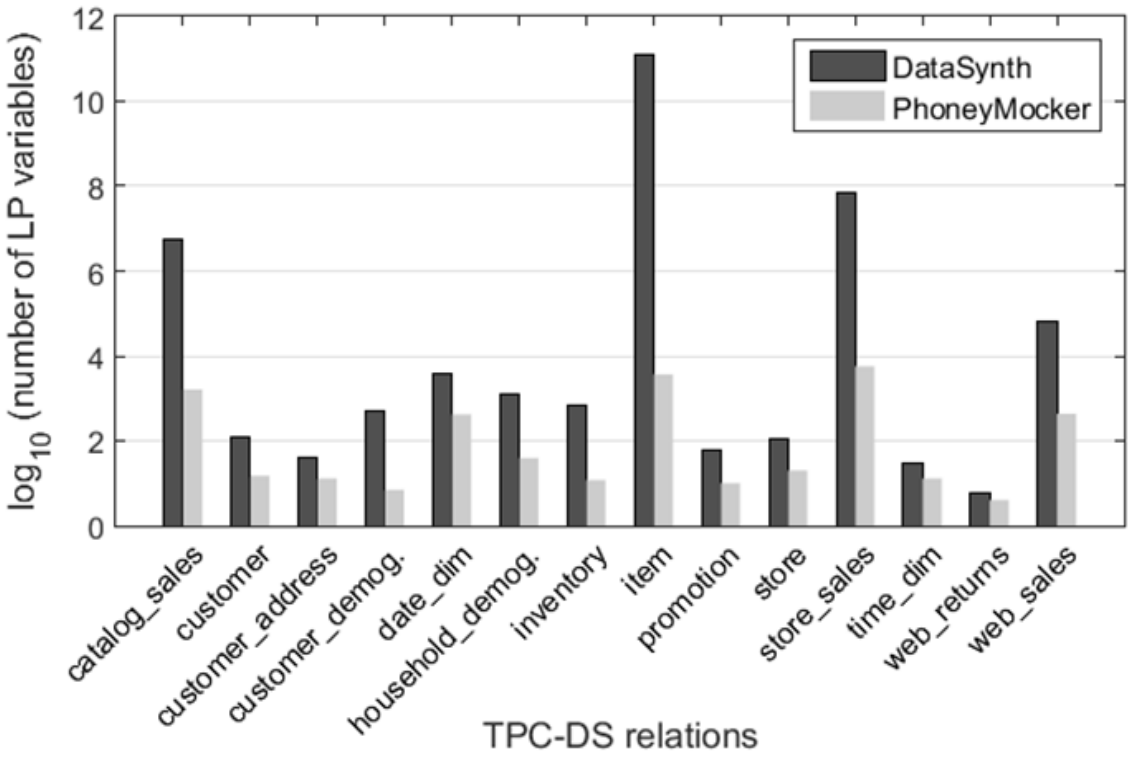


Figure 6.3: Number of LP variables (WL_c)

From an absolute perspective also, the large number of variables created by DataSynth is a critical problem since the LP solver crashed in handling these cases. In marked contrast, PM's LP has only a few thousand variables which were easily solvable in less than a minute. Further, even when we switched to the simple workload, WL_s, the LP solution time for DataSynth was almost an hour, whereas PM completed in a few seconds. These statistics are quantitatively shown in Table 6.1.

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

Table 6.1: LP Processing Time

Since, workload-2 is a superset of workload-1, the number of variables in the LPs of the former are more. As we can see from the figure, for the three relations- catalog_sales, item and store_sales in workload-2, DataSynth's formulation has over a million variables. The solver was unable to handle such a complex LP and got crashed. On the other hand, PM's LP had only a few thousand variables which were easily solvable in less than a minute.

6.3 Scalability with Materialized Data Size

This experiment compares the data instantiation times, post-LP solution, of DataSynth and PM on the WL_s workload. While PM, in principle, due to its summary-based approach, does *not* have to instantiate the data immediately, we assume in this experiment that the vendor requires complete materialization of the data. In essence, the results here capture the difference in speed between the sampling-based approach of DataSynth and the alignment-based technique of PM.

The experimental results are shown in Table 6.2, where we also present, for comparative purposes, the performance with 1 GB and 10 GB databases, apart from the default 100 GB database. We see here that there is a huge reduction in the materialization time of PM at all database scales. Further, even in absolute terms, PM is able to output a 100 GB database in just over 10 minutes, whereas DataSynth takes 42 hours for the same task.

Size (in GB)	**DataSynth**	**PM**
1	28 min	16 sec
10	4 hours	114 sec
100	42 hours	644 sec

Table 6.2: Data Materialization Time

The preprocessing step before formulating the LPs which include the views construction and decomposition into sub-views has been taken from DataSynth directly. The running time of this step is around 4s irrespective of size of database. Figure **??** shows the running time of

the subsequent components for the three data scales. LP Formulation & Solving time includes the time for running the respective partitioning algorithm for all the sub-views, constructing LP constraints from the CCs, adding the consistency constraints and finally solving the LPs. The post-processing time includes the time for constructing the view solutions from the LP solution, which in DataSynth is based on the sampling technique and in PM is based on the deterministic aligning technique. This also includes the time for making the views consistent and finally extracting relation summaries from them. We show the post processing time in PM as a summation of two entries. Here, the second entry is the time to generate the static dump from the summary if one desires. Particularly, the system is ready for query execution even without this additional time. As we can see that first entry does not scale with the different database sizes. This shows that PM is truly data scale independent. On the contrary, DataSynth experiences a linear growth in the post-processing time with size of data to be generated.

6.4 Scalability to Big Data Volumes

In our next experiment, we validated the ability of PM, thanks to its summary-based technique, to scale to Big Data volumes. To demonstrate this, we modelled an exabyte-sized (10^{18} bytes) data scenario as follows: We used CODD to obtain the optimizer-chosen plans at the exabyte database scale for all the workload queries. To get AQPs for this database, we executed the obtained plans on the 100 GB instance and scaled the intermediate row counts with the appropriate scale factor. PM was able to formulate and solve the LPs (one per relation) and generate the database summary in around 150 seconds. Once the summary is generated, the database can go ahead and start injecting the workload queries since the data can be produced dynamically.

6.5 Dynamism in Data Generation

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

Rel. Name	**Size**	**Row count**	**Scan rate** (in GB per sec)		**Scan rate** (in million tuples per sec)	
	(in GB)	(in million)	**Disk**	**Dynamic**	**Disk**	**Dynamic**
store_returns	3	29	0.19	0.38	1.81	3.62
web_sales	10	72	0.23	0.40	1.67	2.88
inventory	19	399	0.18	0.26	3.73	5.39
catalog_sales	20	144	0.43	0.42	3.13	3.00
store_sales	34	288	0.20	0.39	1.71	3.31

Table 6.3: Data Access Rate

The results of this experiment are shown in Table 6.3 for the five biggest relations in the 100 GB database instance. We see here that the tuple generator is not only competitive with a materialized solution, but is in fact typically *faster*. Therefore, using dynamic generation can prove to be a good option since it can help to eliminate the large time and space overheads that are incurred in (1) dumping generated data on the disk, and (2) loading the data on the engine under test.

Chapter 7

Conclusions

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

In future, one can extend the PM framework by covering a richer set of query operators, such as grouping functions.

Bibliography

[1] A. Arasu, R. Kaushik and J. Li, "Data Generation using Declarative Constraints", *ACM SIGMOD*, 2011. ii, iii, 1, 3, 5, 10, 12, 13, 21

[2] A. Arasu, R. Kaushik and J. Li, "DataSynth: Generating Synthetic Data using Declarative Constraints", *VLDB*, 2011. iii, 1, 3

[3] Ashoke S. and J. R. Haritsa, "CODD: A Dataless Approach to Big Data Testing", *VLDB*, 2015. 10, 11, 14

[4] C. Binnig, D. Kossmann, E. Lo and M. Tamer Ozsu, "QAGen: Generating Query-Aware Test Databases", *ACM SIGMOD*, 2007. iii, 1, 2

[5] E. Lo, N. Cheng, W. W. K. Lin, W. Hon and B. Choi, "MyBenchmark: generating databases for query workloads", *PVLDB*, 2014. 1

[6] L. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver", *TACAS*, 2008. 14, 29

[7] R. E. Tarjan and M. Yannakakis, "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs", *SICOMP*, 1984. 13

[8] R. S. Trivedi, I. Nilavalagan and J. R. Haritsa, "CODD: COnstructing Dataless Databases", *ACM DBTest*, 2012. 10

[9] TPC-H. www.tpc.org/tpch 2

[10] TPC-DS. www.tpc.org/tpcds 29

[11] 2013. PostgreSQL. www.postgresql.org/docs/9.3/static/release.html 29