Towards Achieving Workload Scalability in Database Regeneration

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology
IN
Computational and Data Sciences

BY

Subhodeep Maji

Department of Computational and Data Sciences
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2021

Declaration of Originality

I, **Subhodeep Maji**, with SR No. **06-18-01-10-51-19-1-16553** hereby declare that the material presented in the thesis titled

Towards Achieving Workload Scalability in Database Regeneration

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

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

Date: 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

© Subhodeep Maji
June, 2021
All rights reserved

DEDICATED TO

My Parents

Mrs. Sucheta Maji and Mr. Subhashish Maji;

For their endless love and support

Acknowledgements

I want to express my 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 continuous support at every step.
I am thankful to Anupam Sanghi for mentoring and assisting me throughout my research journey. He has played a big part in the completion of this project with his constant support and motivation. I would also like to thank my lab mates for their valuable suggestions and continuous encouragement.
Finally, I would like to thank my parents, who have always supported me indefinitely.

Abstract

A fundamental requirement for testing the performance of database systems is to regenerate real databases synthetically such that certain pivotal characteristics of the original database are retained. The literature has several techniques which aim to achieve this by objectifying similar number of row cardinality at each operator of query execution for a given workload of query on the two databases. Hydra is the state-of-the-art work on this, which can generate a database summary with a run time independent of the size of the database to be regenerated. From this summary, tuples can be generated during query execution. However, it is limited on the workload size that is taken as input due to the underlying linear programming (LP) technique used in it which leads it to overwhelm the memory.

This work is derived from the basic principles of Hydra with a core focus on how the LP is formulated and solved. Instead of directly using a library for solving the LP, we propose the use of the classical *column generation* technique which *omits the requirement to store huge number of columns* of the coefficient matrix, which is given as the input to LP solver, thereby saving significant memory. The column generation problem is solved as an integer linear problem, the constraints of which are formed by using the Quine-McCluskey (QM) algorithm. We also propose optimizations over the QM algorithm for our particular case.

Hydra has a workload decomposition module over its LP formulation module which reduces the complexity of the problem. This module is not compatible with the optimizations proposed in this work. To show merit of the ideas presented, we compare our work without the decomposition module. In our experiments, it was found that the proposed technique is more efficient in terms of memory and can run for test cases where Hydra fails. However, Hydra with the optimization proved to be more space efficient than our approach.

Finally, we propose the use of Dantzig Wolfe decomposition as an approach to make the optimization proposed and the one already present compatible with each other.

Contents

Acknowledgements **i**

Abstract **ii**

Contents **iii**

List of Figures **v**

1 Introduction **1**

1.1 Workload Dependent Data Generation 1
1.2 Hydra 2
1.2.1 Limitations 4
1.3 Modified LP Solver 4
1.4 Related Work 6
1.5 Our Contributions 9
1.6 Organisation 10

2 Modified Solver **12**

2.1 Prerequisites: Simplex Algorithm 12
2.1.1 Phase 1 Simplex Algorithm in Hydra 13
2.1.2 Initialization for phase 1 simplex 14
2.1.3 Iteration 14
2.1.4 Termination 15
2.1.5 Degeneracy and Cycling 16
2.2 Prerequisites: Column Generation 17
2.3 Modified LP Solver for Hydra 17
2.3.1 Representing Columns in Hydra like Systems 17
2.3.2 Logical Boolean Expressions as ILP Constraints 18

2.3.3 Artificial Variable Reducer 20
2.3.4 Removing Cycling in Column Generation 22

3 Adapted Boolean Expression Minimizer **24**
3.1 Prerequisites: QM Algorithm 24
3.2 Modified QM with Minterms on the Fly 28

4 Implementation Details and Experimental Evaluation **33**
4.1 System and Implementation Details 33
4.2 Results 34
4.2.1 Peak Memory Usage Comparison 34
4.2.2 Time Comparison 36
4.2.3 Accuracy Comparison 38
4.2.4 Time and Memory with Increase in Number of Tuples 38
4.2.5 Comparison with Hydra With optimization 39

5 Extension **41**
5.1 Theoretical Extension 41
5.1.1 Prerequisite: Dantzig Wolfe Decomposition Algorithm 41
5.1.2 Prerequisite: Subview Optimization in Hydra 43
5.1.3 Applying Dantzig Wolfe to Hydra 44
5.2 Implementational Improvements 44
5.2.1 Parallel Programming 44
5.2.2 Alternate Algorithm for Lattice Traversal 45

6 Conclusion and Future Work **46**

Bibliography **47**

List of Figures

1.1 Example of CC Formation . . . 2
1.2 Example CCs on Age and Salary . . . 2
1.3 CCs from Figure 1.2 Represented in Attribute Space . . . 3
1.4 Equations Formed from Figure 1.3 in Matrix Form . . . 4
1.5 Original Hydra Architecture . . . 5
1.6 Proposed Architecture . . . 5
1.7 Regions in DataSynth . . . 8

2.1 Truth Table Formed from Example in Figure 1.4 . . . 20

3.1 Implications from Observations . . . 28
3.2 Intersection Region of {2, 3} Based on Figure 1.3 . . . 30
3.3 Lattice Structure Formed by $regions_l$ . . . 30

4.1 Memory Comparison for Table 1 . . . 34
4.2 Memory Comparison for Table 2 . . . 35
4.3 Time Comparison for Table 1 . . . 36
4.4 Time Comparison for Table 2 . . . 37
4.5 Time Break Up for Colgen for Table 1 . . . 37
4.6 Time Break Up for Colgen for Table 2 . . . 38
4.7 Memory Comparison with Scale 10 . . . 39
4.8 Time Comparison with Scale 10 . . . 40

Chapter 1

Introduction

Generating synthetic databases (DB) are often required for applications like testing of engines, performance impacts of planned upgrades, etc. We are particularly interested in regenerating a synthetic database from a fixed set of queries whose output row cardinality, along with the cardinalities at each intermediate operation for a given query execution plan to obtain the result, is known. The aim is to have the same (or very similar) row cardinalities at each operation of query execution when the same queries are run on the synthetic DB, to mimic the runtime performances of the DB. It is to be noted that no data from the original database is ever transferred to regenerate the database. The number of constraints that can be processed is limited by the amount of memory in the system. Our work focuses on scaling up the number of constraints with known row cardinality that is supported on the system, by making the solving technique more space efficient.

1.1 Workload Dependent Data Generation

The class of workload dependent data generation deals with generating database which is volumetrically, i.e. in the number of rows, similar to the original database for a given workload. The concept of cardinality constraints (CC) for capturing volumetric similarity was proposed by DataSynth[12]. A CC expresses that the output of a given relational expression over the generated database should feature the specified number of rows.

Workload dependent data generation frameworks [12, 7] take a database schema and Annotated Query Plans (AQPs) as input. An AQP is the annotated query execution plan of a given query which is represented as a tree. The leaves of the tree are tables that are involved in the query, and each upward edge signifies an operation (e.g., filter, join) on the current node, which is paired with a number representing the cardinality (number of rows returned) of the

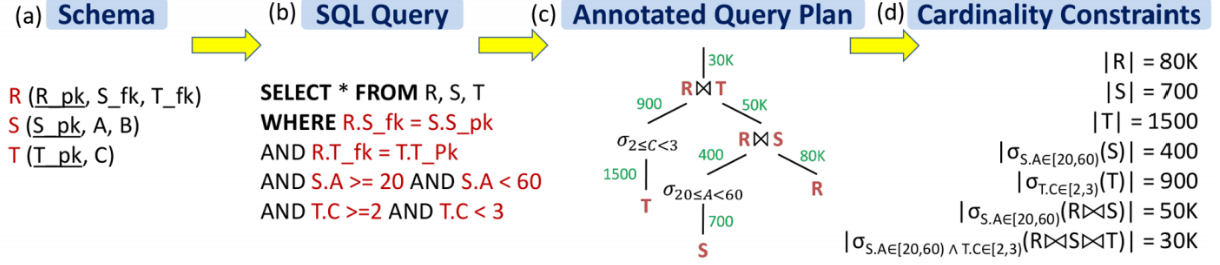

Figure 1.1: Example of CC Formation

result. As output, they produce a synthetic database that has similar volumes of data at each operators in the AQP.

From each annotated number, we can form a CC. An example of how CCs are formed is shown in Figure 1.1 for clarity. Figure 1a shows an example database schema on which an example query is run, and the resultant AQP is shown in Figure 1c. The CCs formed from the AQP are shown in Figure 1d.

1.2 Hydra

Based on the CCs retrieved from the AQPs, the space represented by the domain of the attributes of a denormalized table is partitioned into disjoint *regions* such that the CCs from the AQPs can be characterized to form a linear program (LP). The regions so formed may not be of a regular shape like rectangle and can even be non-convex. One such LP is formed per table. The partitioning is such that each point in a single region satisfies exactly the same CCs.

Consider the CCs given in Figure 1.2. The space represented by the CCs is given in Figure 1.3.

- $10 \leq Age < 65 \wedge 10k \leq Salary < 80k$; Cardinality : 210
- $50 \leq Age < 85 \wedge 40k \leq Salary < 70k$; Cardinality : 260
- $20 \leq Age < 40 \wedge 30k \leq Salary < 70k$; Cardinality : 50
- $30 \leq Age < 80 \wedge 40k \leq Salary < 60k$; Cardinality : 200

Figure 1.2: Example CCs on Age and Salary

The CC salary between 40000 and 60000 and age between 30 and 80 is represented in the blue bounding box (fourth CC in Figure 1.2). Note that inside each CC, there may be multiple

regions present where each region is labeled according to the CCs it satisfies. For example, region $\{1\}$ is satisfied only by CC_1 while region $\{1,3,4\}$ satisfies CC_1, CC_3, and CC_4. For each constraint, an equation can be formed consisting of the regions present inside it and equating it with the cardinality of the CC.

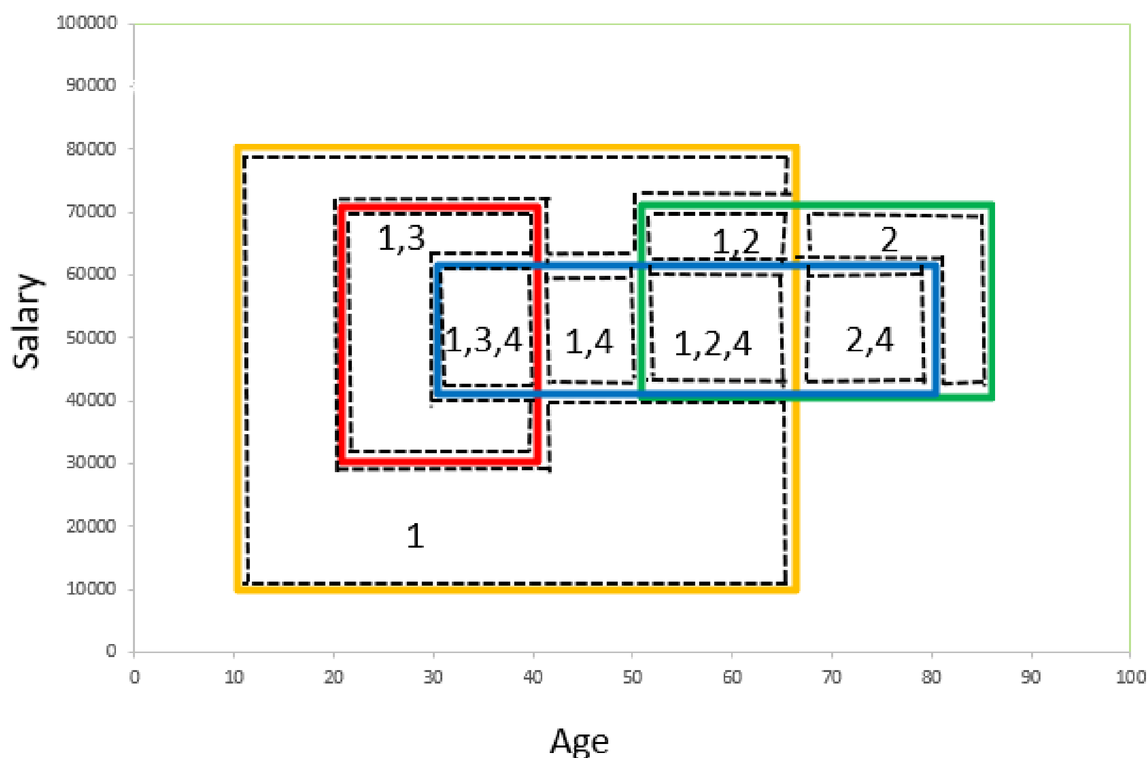

Figure 1.3: CCs from Figure 1.2 Represented in Attribute Space

For example, the CC discussed earlier can be represented as $x_{1,3,4} + x_{1,4} + x_{1,2,4} + x_{2,4} = 200$, where 200 is the cardinality of CC_4 and x_i represents the cardinality of region i. Likewise, there will be one equation for each constraint, and it can be represented in matrix form as shown in Figure 1.4. This matrix is called the coefficient matrix. The fourth row of the matrix corresponds to the constraint discussed.

The number of columns in the coefficient matrix is equal to the number of regions (each region has an associated variable), and *the column corresponding to a variable has the information of the CCs, which the region corresponding to the variable satisfies*. For example, consider column 2 corresponding to the variable $x_{1,3}$; the entries at positions 1 and 3 are equal to 1 while the rest of the entries in the column are 0. Note that for m CCs, there can be a maximum of

2^m columns representing the power set of the set of CCs, and thus the size of the coefficient matrix can be of dimensions $m \times 2^m$ in the worst case. Another interesting thing to note here is that all the entries in the coefficient matrix are either 0 or 1, which makes the problem a special case. This special case is exploited by introducing Boolean logic operators to represent the columns, required as part of the column generation module section 2.2.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_{1,3} \\ x_{1,3,4} \\ x_{1,4} \\ x_{1,2} \\ x_{1,2,4} \\ x_{2,4} \\ x_2 \end{bmatrix} = \begin{bmatrix} 210 \\ 260 \\ 50 \\ 200 \end{bmatrix}$$

Figure 1.4: Equations Formed from Figure 1.3 in Matrix Form

Hydra formulates this as a linear programming problem such that each CC forms a constraint without any optimization function. Note that the problem so formed is almost always underdetermined (i.e., the number of variables is more than the number of equations) and is always feasible since we know that there exists at least one solution.

1.2.1 Limitations

Hydra uses Z3[10] solver to find a feasible solution to the LP. All the constraints are explicitly added row-wise in the solver. Apart from that, the boundaries of the regions formed from the problem are also stored in the memory as multidimensional arrays and take significant space. It has been observed that for workloads with 200+ queries, the memory runs out on a machine with 16GB main memory. The fact that the coefficient matrix has only 0s or 1s is not exploited while solving the LP, and we wish to benefit from the limited domain of the numbers.

1.3 Modified LP Solver

We propose a modified LP solver that does not have the requirement to store the whole coefficient matrix and diminishes the space requirements without compromising on the quality of the solution. We also propose modifications to eliminate the requirement to store all the regions at the same time and generate them on the fly when needed.

The original and proposed architectures are shown in Figure 1.5 and Figure 1.6, respectively. It is to be noted that we are using the same pre-processing and post-processing steps as Hydra

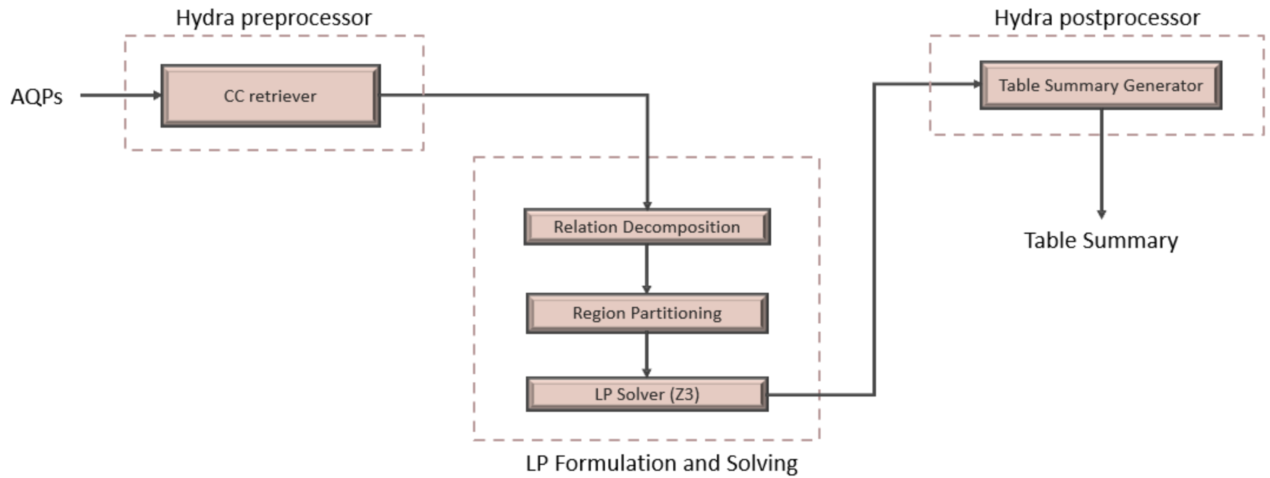

Figure 1.5: Original Hydra Architecture

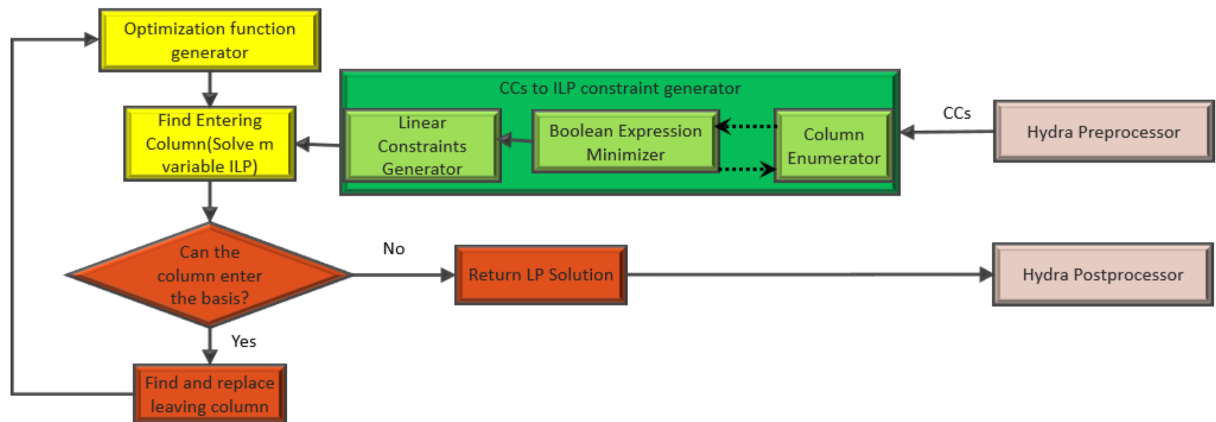

Figure 1.6: Proposed Architecture

while modifying the LP formation and solving methodology.

Step I: In the first step, a modified version of the QM[8] algorithm is used to minimize a Boolean expression, to represent all the columns in the LP in a compact manner. The terms in the expression are constructed from the regions formed for a given workload. All the regions

are known after doing region partitioning (1.5) in Hydra. However, we wish not to use that module because it stores how each region looks and takes up memory. Instead, we propose to probe regions to check for their presence. Since there can be 2^m regions for m CCs, probing in a naive way, which can have 2^m probings in the worst case, is impractical. We do probing in a systematic manner such that the results of certain probes may eliminate the requirement to probe other regions.

Step II: The minimized expression formed is used to form integer linear programming (ILP) constraints. These constraints signify how the columns of the coefficient matrix may look. Since all the columns of the coefficient matrix are not stored, these constraints are useful while generating new columns. Only the constraints are formed and stored in this step, and no ILP is solved. Note that the above two steps are done exactly once.

Steps III-IV: The ILP constraints thus formed are then passed on to the LP solver, which we have implemented using the Simplex algorithm and column generation technique. Only m columns of the coefficient matrix (this submatrix is called the basic matrix) and exactly m variables are stored at each point. In each non-terminating iteration of simplex, a new column is found to replace one of the columns already present. To find a new column, an ILP optimization function is solved (step 4), the constraints of which are always the same and are taken from the previous step, and a new optimization objective is formed by doing computations that are fundamental to the simplex algorithm. The column so generated is checked for potentially replacing one of the existing columns. The algorithm terminates when a newly generated column cannot replace the existing columns.

1.4 Related Work

The area of synthetic data generation has received attention over the past three decades. The work on synthetic data generation can be broadly classified into two categories; one not considering the query workload and the other dependent on the workload. In the former method, the approaches are majorly mathematical, where data distributions are used for generating databases(e.g. [13, 14]). In [16], the Data Generation Language is used to generate synthetic distribution using iterators. Here, the construction of a dependent table is bottlenecked by access to the referenced table. Several improvements [18, 23] were made to this technique with respect to the generation speed.

A lot of work has been done in the category of synthetic data regeneration dependent on workloads. RQP [17] takes a query and the result of the query as the input and reverse engineers

the original database table. QAGen [11] used cardinalities from a query plan tree. It first constructs a symbolic database (DB) and adds constraints derived from the query plan trees over the symbols. In a symbolic database the values in the DB are symbols or variables and constraints are put over the symbols according to the query. A Constraint Satisfaction Program (CSP) is used to get instantiation for the symbols used. This method is able to handle complex operators, as the CSP can handle generic constraints. However, a major limitation is the size of the table of the number of rows supported as the complexity of CSP grows rapidly with it. It more so concerns in today's world where DBs in Terabytra or petabytes are common. Also, the approach creates a DB of the size of the original table, and a table per query which further limits its usage. For example, for 15 queries on a single table, 15 tables will be generated with sizes proportional to the size of the original table, thus inflating the memory requirement. The limitation was somewhat addressed in MyBenchmark[15] which initially produces one symbolic table per query and later tries to merge them into a smaller number of tables, ideally 1. The merging of the DBs is done by reducing the problem to finding the classical maximum bipartite matching problem solution, which itself is solved in literature by reducing it to the classical maximum flow problem in graph theory. Push-relabel algorithm is used for solving the maximum flow problem. After the merging process, the number of tables can range from 1 to the original number of tables. However, they have experimentally shown that this number is usually low on the standard workloads of TPC-C,TPC-E, TPC-W, TPC-H. The work also proposes randomization and approximation with an error tolerance over the methods. The number of ways of joining the DBs is exponential and is the nth Catalan number of n tables and hence is large. Their preliminary result on this work considered only left-deep plans for joining (the DBs). They later improved it to accommodate bushy plans which allows the use of parallel processing. Although much more promising than QAGen with fine theory, the work is again limited by the number of tuples in the DB since the number of vairables in the CSP is dependent on it. The push-relabel algorithm used has a cubic time complexity on the number of nodes (which is dependent on the number of rows). Thus, for DBs having several billion tuples, this method can be too time-consuming and infeasible.

DataSynth [12] proposed Cardinality Constraints(CC) as means of expressing data characteristics. DataSynth provided a framework for producing a database that adheres to a set of CCs. In this framework, a denormalized table is constructed for each table, which comprises non-key attributes of this table and the non-key attributes of tables that it is connected to through referential integrity. The join expression is replaced by a single denormalized table using this construction, assuming only PK-FK joins. Next, the data space of the denormalized

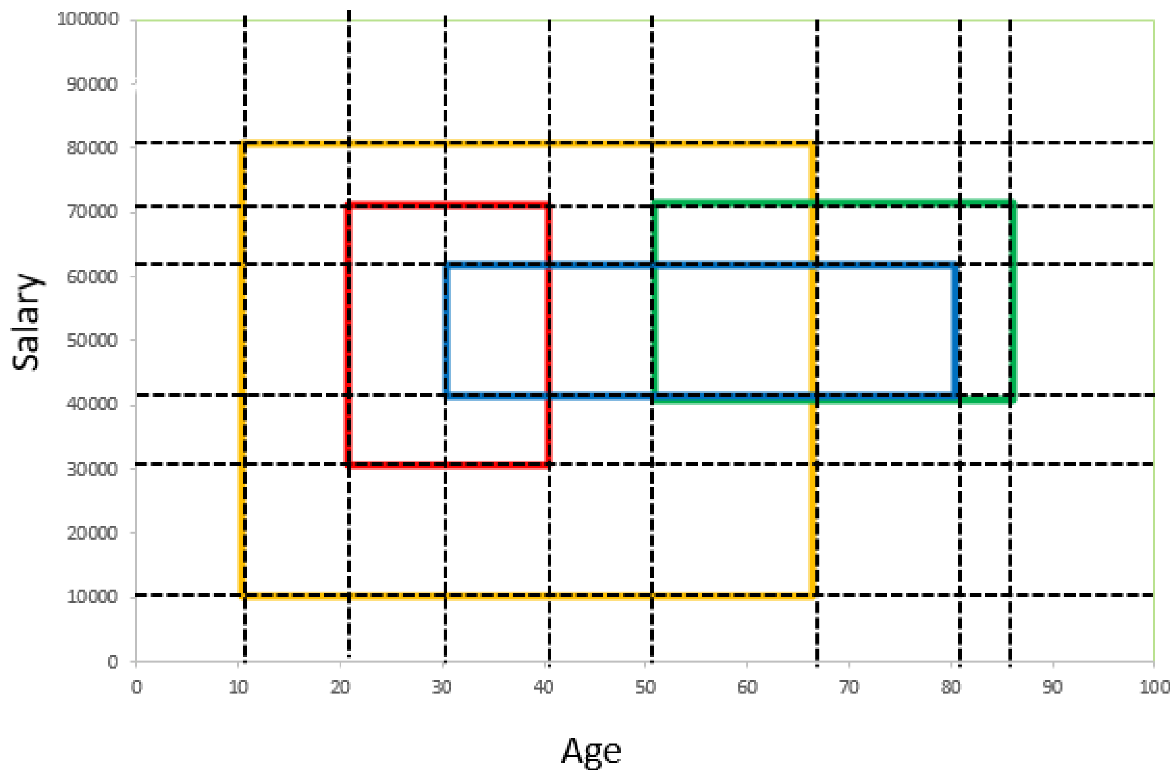

Figure 1.7: Regions in DataSynth

table is partitioned into a group of regions such that domain points in each region satisfy the same set of CCs. The regions are found simply by extending the boundary projections on the axes of the attribute space. Consider the example of CCs given in Figure 1.2, the regions for this problem is shown in Figure 1.7. Each rectangular box (formed by dotted lines) forms a region. The regions in this setting are trivially known and it is to be noted that the number of regions in this is not the minimum possible. E.g. 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. This method is independent of the number of tuples in the output of CCs, which merely remains as a number and thus independent of the size of the DB to be mimicked. However, it is limited in the query workload size that is supported due to its explosion in the number of variables because of the trivial region formation approach. The probabilistic way of generating tuples

also introduces some error, where the error is defined as the output cardinality mismatch in the CCs, the details of which can be found in [12]

Hydra [7] extended the above framework to provide scalability to the generation process. We discuss some of the primary changes which address the issues highlighted above. Firstly, Hydra proposes a new approach to partition the regions, drastically reducing the number of variables required in the LP. The formulation is optimal in terms of the number of regions that are formed, i.e., all the points that satisfy the same CCs belong to the same region. The regions formed after partitioning the space by CCs in Figure 1.2 is shown in Figure 1.3. The regions are labelled according to the CCs it satisfies. Notice that the regions are no longer (hyper) rectangles, and not even convex anymore. Another major change was in the data generation process which was made deterministic, resulting in fewer errors. It also provided dynamic data generation by constructing a database summary used for on-the-fly tuple generation during query execution.

1.5 Our Contributions

Our work is derived from [7] with a focus on reducing the memory required. The proposed modifications are as follows:

- **Use of Column Generation Algorithm:** In the earlier work, the Z3 library [1] was directly used after the formulation of LP. This way of solving had two specific bottlenecks: the region partitioning algorithm and the LP solver.
  We have proposed the *use* of Column Generation to reduce the memory footprint required for the LP solver. The memory required is traded for extra computations, which may result in extra compute time.

- **Constraint generation for Column Generation:** The constraints of how the columns look are known trivially in typical Column Generation problems.
  In our case we generate the ILP constraints from the columns. This is possible only because the columns have Boolean values and could not have been done for any set of general columns.

  Also, the space consumption of enumeration of all the columns at the same time is equivalent to the space required without column generation. We propose to

- **Use of QM Algorithm for Generating ILP Constraints:** QM Algorithm is used to

minimize Boolean expressions, which are typically present in digital circuits.
Due to the Boolean domain in our problem, we are able to represent the columns in our problem minimally using the algorithm. The bridge between minimal Boolean representation and ILP constraints is formed by transducing the former to the latter by representing logical operations like and, or, not, using ILP constraints. The conversion, though done independently, can also be found on the web.

- **Modification over QM Algorithm:** The QM algorithm in the domain's terminology requires all the minterms (which analogically are columns/ regions in Hydra) present at the same time. This would mean storing all the columns/ regions if this were to be used as-is.
  We analyze the algorithm and propose a modification to it so that not all the regions are required at the same time. We do so by using the CC intersection monotonicity property section 3.2.

- **Implementation of the ideas:** All the proposed ideas were implemented from scratch covering various edge cases. The module was successfully integrated into the existing Hydra pipeline.

1.6 Organisation

In chapter 2 we briefly discuss the mathematical background required to understand our work which includes an introduction to the *Simplex algorithm* and *column generation technique* in section 2.1 and section 2.2, respectively. It is followed by showing how we can generate minimized Boolean constraints to represent all the columns, given the columns, in subsection 2.3.1. The conversion of Boolean constraints to linear constraints so maintain compatibility with LP theory is shown in subsection 2.3.2. In subsection 2.3.4, we address the issue of degeneracy and cycling that are inherent issues to both simplex and column generation and occur in our experiments as well.

Until chapter 2, we assume that all the columns are available to feed into Boolean expression minimizer to get the Boolean constraints for the columns. Thus, step I mentioned in the previous section is not discussed, which is a modification to Boolean expression minimization algorithm, namely the QM algorithm. chapter 2 is complete in itself in the terms that the system can run completely with the modifications proposed.

chapter 3 first introduces the QM algorithm and then talks about the modifications made to the QM algorithm for our specific case so that the minterms (or equivalently the columns) are not required as input. We exploit the monotonicity property of CC intersection, i.e. if a given number of CCs do not intersect at all, then no superset of these CCs can intersect to do so.

In chapter 4, we discuss the system and implementation details, followed by the results.

In the discussions throughout the thesis, we have not considered an optimization already present in Hydra, which decomposes the LP, namely the subview optimization, by which the LP simplifies in terms of the number of variables. Experimentally, it was found to be better than just the optimizations proposed in this thesis. Note that the optimizations proposed here work better than Hydra without subview optimization. In chapter 5, we propose the theory to use both optimizations at the same time.

Chapter 2

Modified Solver

2.1 Prerequisites: Simplex Algorithm

One of the most popular algorithms to solve an LP is the simplex method. In its standard form, the simplex algorithm solves the following problem:

maximize

$$c^T x$$

subject to

$$Ax = b$$

$$x \geq 0$$

where, $A \in R^{m \times n}, x \in R^{n \times 1}, b \in R^{m \times 1}, c \in R^{n \times 1}$

Here, the x vector is variable, the value of which we need to find. The c vector denotes the optimization coefficient for x. The coefficient matrix (A), along with the RHS values (b vector), specify the constraints on the optimization function.

Let the number of constraints be m, and the number of variables be n. The feasible space represented by the constraints forms a convex polytope. Each variable has its dimension in the space. Each corner point in the polytope has at most m non-zero dimensions (variables) and at least n-m dimensions (variables) as 0. The solution corresponding to these feasible corner points is called a basic feasible solution (BFS).

Definition 2.1 (Basic Feasible Solution) *A solution to the simplex problem where at most m of the n variables take non-zero values.*

We state the fundamental theorem of LP below, considering the LP in the standard form.

Fundamental theorem of LP

- *If there exists a feasible solution to the LP, then there exists a basic feasible solution(BFS).*
- *If there exists an optimal solution (with respect to the optimization function) to the LP, then there exists an optimal BFS.*

Thus, there is at least one optimal solution for the given problem where at most m variables have non-zero values and simplex tries to find it. The simplex method is an iterative algorithm that starts at a BFS and iterates over other BFS till an optimal BFS is found. So, at all points, we are concerned only about m variables, called basic variables, and the rest of the variables that necessarily have 0 value are called non-basic variables. This can be mathematically stated as below-

$$Ax = \begin{bmatrix} A_B & A_N \end{bmatrix} \begin{bmatrix} x_B \\ x_N \end{bmatrix} = b$$

For BFS, $\mathbf{x_N} = \mathbf{0}$

$$Ax = \begin{bmatrix} A_B & A_N \end{bmatrix} \begin{bmatrix} x_B \\ 0 \end{bmatrix} = b$$

The subscripts B and N in the above equations refer to the basic and non-basic components of the respective matrix/ vector.

2.1.1 Phase 1 Simplex Algorithm in Hydra

The simplex algorithm needs a BFS to start with. However, finding a BFS is not trivial, and one of the ways to find it is using the phase 1 algorithm of Simplex. In this method, the problem to find a BFS is itself solved as an optimization problem. Artificial variables that were not originally part of the problem are introduced. A trivial solution to the artificial problem can be obtained with an optimization function to remove it, so that a BFS can be found for the original LP.

In the next subsections, we discuss the various phases of the simplex algorithm with respect to Hydra.

2.1.2 Initialization for phase 1 simplex

An artificial variable that was not originally part of the problem is introduced per constraint. A trivial solution to finding BFS with the new formulation will be to assign the right-hand side value of each constraint to the corresponding artificial variable. For example, let us say we added an artificial variable a_1 for constraint 1, the solution will have $a_1 = b_1$ where b_1 is the first value in the b vector. Thus A_B matrix (matrix consisting of the columns corresponding to the basic variables) would be an identity matrix. However, it is to be noted that the solution is in terms of variables that were not a part of the original problem. Thus, an optimization function is added, forcing the artificial variables to take 0. One of the ways of doing it would be to put an optimization function as *maximize* $d^T a$, where d is a vector having all $-1s$ (any negative value works equally well) and a is the vector of artificial variables. This can be represented mathematically as follows-

$$\begin{bmatrix} A_{original} & A_{artificial} \end{bmatrix} \begin{bmatrix} x \\ a \end{bmatrix} = \begin{bmatrix} A & I \end{bmatrix} \begin{bmatrix} x \\ a \end{bmatrix} = b$$

Note that some of the columns of the identity matrix could be present in the original problem itself, and no artificial variable or columns are introduced for such columns. Thus, the above mathematical formula represents a system where no column of the identity matrix was already present.

2.1.3 Iteration

In each iteration of the simplex, an entering variable (and its corresponding column) is chosen, which will 'enter' the basis, and a leaving variable (and its corresponding column) is chosen to 'leave' the basis, thus updating the basis. This updating of the basis is done till none of the termination conditions (described in the next section) are met.

Let c_B denote the optimization coefficients for the basic variables. $c_B \times A_B^{-1} \times A_i$ is calculated, where A_i denotes the ith column of the A matrix. Any column (and its corresponding variable) with a negative value of $c_B \times A_B^{-1} \times A_i$ - c_i can enter the basis, where c_i is the optimization coefficient of the entering column. Since there can be multiple such columns, any column can be chosen either in a deterministic or random manner. Some of the deterministic rules based on heuristics are the largest coefficient rule, largest increase rule, and first negative value. The details of these rules can be found in [9].

To find the leaving variable index, the entries of b vector are divided by the entries of the entering column and the index having the least non-negative value leaves the basis. Note that there are $n - m$ choices for an entering column and m choices for the leaving column.

2.1.4 Termination

The termination of the iterations can happen when any of the following is detected -

- Alternate optimum
- Unboundedness
- Infeasibility
- Cycling
- No entering column

Since Hydra is not particularly interested in the type of solution and any solution giving the BFS would do, the iterations can stop when we detect that the algorithm is iterating on alternate optimums. Unboundedness is the case when the optimal value is either ∞ or $-\infty$. Infeasibility is detected when there are no values of x such that all the constraints are satisfied. Both the cases can never occur in Hydra, as the constraints are generated from a real database which has itself as the solution. The fourth case of cycling is said to happen when we reach the same basis after some iterations. If the entering and leaving variables are chosen in deterministic ways, then the same basis will be reached after some iterations. One way to overcome cycling is to use Bland's rule, which has rules for entering and leaving variables in case of ties based on their indexing. More details about it can be found in [9]. We encounter cycling in our experiments done using column generation, and the details of how the cycle was broken are given in subsection 2.1.5. If no column can enter the basis, then the current solution is the optimal solution and the iterations are terminated.
A high level simplex algorithm is shown in algorithm 1.

Algorithm 1: Phase 1 Simplex Algorithm

```
  Result: BFS of the problem
1 basis = Identity matrix
2 optimizationCoefficient = [0]*numVariables
3 for Column in Identity matrix do
4 |  if Column is not part of Original problem then
5 |  |  Set negative Optimization coefficient for the corresponding variable
6 |  end
7 end
8 potentialColumn = getEnteringColumn(basis, optimizationCoefficients)
9 while potentialColumn can enter the basis do
10 |  leavingIndex = findLeavingIndex(basis, potentialColumn)
11 |  basis[leavingIndex] = potentialColumn
12 |  potentialColumn = getEnteringColumn(basis, optimizationCoefficients)
13 end
```

2.1.5 Degeneracy and Cycling

Degeneracy is said to happen when there are multiple potential leaving variables. In such a case, one of the basic variables takes zero value in the next iteration. When degeneracy occurs, there is no improvement in the objective value function in the next iteration. The algorithm comes out of degeneracy by itself to move towards the optimum, and thus the computations are simply wasted, but it does not affect the convergence of the algorithm. There are no proven ways to avoid degeneracy.

Cycling is said to happen if we get to the same set of basis after some iterations. In contrast to degeneracy, the algorithm gets stuck in this case and does not terminate. Cycling can be overcome in simplex, and one of the ways to do so is using Bland's rules which orders all the columns and takes the ordering into account. Interested readers may refer to [9] for details.

We encounter the issue of both cycling and degeneracy in our experiments. While degeneracy only causes performance degradation with respect to running time and gets out of the degeneracy condition in subsequent iterations, cycling is like an infinite loop and requires special treatment. The details are present in subsection 2.3.4.

2.2 Prerequisites: Column Generation

Column generation is a technique used to solve simplex algorithm where the number of variables is too large and the columns can be represented mathematically as some pattern, set or permutations. It is to be noted that the representation should cover all the possible columns in the original problem, but the representation should not consist of any column that was not in the original problem. Continuing from the context of the previous section, instead of storing all the columns, we represent all the "permitted" columns in a symbolic sense with variables e_1, e_2, e_3 ... e_m (vector e) and thus to find an entering variable, instead of calculating $c_B A_B^{-1} A - c_i$, we would rather need to find a possible value of column e such that $c_B A_B^{-1} e - c_i$ has an entry that is less than 0, where A_B is the basic matrix, c_B is the coefficient of optimization for basic variables. If we have a way of representing all the set of permitted columns mathematically(using constraints), then we can find an entering column by minimizing $c_B A_B^{-1} e$ (so that we can get a negative entry in $c_B A_B^{-1} e - c_i$) such that those constraints are satisfied. Now the problem has reduced to finding a way to represent all the permitted columns. It is to be noted that the new optimization problem of finding a new column deals in only m variables. We can incur more computational costs when solving an LP using column generation but save on the space required to store all the columns (sans columns in the basis).

For more details and example on the column generation problem, please refer to [9].

2.3 Modified LP Solver for Hydra

2.3.1 Representing Columns in Hydra like Systems

It is to be noted that even though column generation is a well-established technique, it is traditionally not used without knowing the constraints on the columns beforehand. The integration of the method with Hydra is not trivial. In typical cases, the constraints required representing all the permissible columns, are already known (e.g. in the classical cutting stock problem), but in the case of Hydra we generate constraints given the columns. Hydra uses the 'region partitioning' algorithm [7], which splits the space of attributes of a table as explained in section 1.2 and thus generates all the possible columns. We wish to generate constraints on these columns so that the new LP Solver is compatible with the column generation method. Such a representation would help in saving the memory used in the LP Solver. Another memory bottleneck occurs while computing the regions using region partitioning algorithm; the same has been addressed in chapter 3.

Recall from section 1.2 that we have a column (and its corresponding variable) for each region in the attribute space. Following the theory from the simplex algorithm, we can say that at most m of those regions will have non-zero cardinalities, and thus the solution obtained would be skewed in terms of the distribution of tuples.

The first thing to note while forming constraints for the columns present is that all the values are either 0 or 1. *We exploit this very fact of values being boolean to form constraints. It is to be noted that for the system to work with column generation, the constraints should be represented as linear constraints, so that the system can work with the column generation module. This would not have been possible for generic column sets if the domain of the values of columns were non-boolean, e.g., real numbers or integers.*

So, representing all the columns as constraints reduces to finding a way to represent boolean terms, with the terms derived from columns. For example, if the column is [1, 0, 1, 0], the corresponding term would simply be '1010'. These terms are called minterms in the context of digital logic. Now, we would like to represent all the derived minterms in a compressed or minimized format. *This is the same as solving the problem of boolean expression minimization*, a well-studied problem in digital logic, assuming that we can convert minimized boolean expressions as linear constraints. One of the most popular algorithms which are scalable in terms of length of minterms (or length of columns) and implementable on a computer is the QM algorithm. A brief about the algorithm is written in section 3.1, and more details can be found in [8]. We can use the QM algorithm directly in this approach. However, to address memory issues arising from region partitioning, we also propose modifications on the QM algorithm for our particular case (where the minterms are obtained from Hydra). In the next section, we show, how to convert minimized boolean expressions to linear constraints.

2.3.2 Logical Boolean Expressions as ILP Constraints

Column generation is a technique used to solve a simplex algorithm where the number of variables is too large. The columns can be represented mathematically as some pattern, set, or permutations. Continuing from the context of the previous section, instead of storing all the columns, we represent all the "permitted" columns in a symbolic sense with variables e_1, e_2, e_3 ... e_m (vector e) and thus to find an entering variable, instead of calculating $c_B B^{-1} A - c$, we would rather need to find a possible value of column e such that $c_B A_{A_B}^{-1} e - c_e$ has an entry which is less than 0. If we have a way of representing all the set of permitted columns mathematically(using

constraints), then we can find an entering column by minimizing $c_B A_B^{-1} e$ (so that we can get a negative entry in $c_B A_B^{-1} e - c_e$) such that those constraints are satisfied. Now the problem has reduced to finding a way to represent all the permitted columns. It is to be noted that the new optimization problem of finding a new column deals in only m variables.

We wish to form linear integer constraints on the symbolic entering so that we can use the existing ILP solving technqiues to solve the problem. Let us first take a look at how linear constraints can be used to represent boolean operations. Boolean operations are relevant as all the values in the coefficient matrix are either 0 or 1. We want to establish the fact that 'logical and' ($\wedge$), 'logical or' ($\vee$), and 'logical not' ($\neg$) operations can be mimicked for any number of boolean variables using integer linear programming (ILP). To do so, we *make the variables boolean* by using ILP and restricting the domain of all the variables between 0 and 1.
Now, to put a constraint that y = $e_1 \wedge e_2 \wedge ... \wedge e_n$, we can add two linear constraints :

$$\Sigma e_i - n \times Y \leq n - 1$$

$$\Sigma e_i - n \times Y \geq 0$$

Proof: It is sufficient to show that when Y=1, all the e_i are 1 and when y=0 at least one of the $E_i = 0$. It is to be noted that Σe_i ranges from 0 to n. If Y=1, then from the second equation $\Sigma e_i \geq n$ which implies that all $e_i = 1$ (since $e_i \leq 1$). Y = 0 from the first equation $\Sigma e_i \leq n - 1$ implies that not all the e_i values are equal to 1.
Similarly to put a logical or condition such that y = $e_1 \vee e_2 \vee .... \vee e_n$, we can add two linear constraints like:

$$n \times Y - \Sigma e_i \leq n - 1$$

$$n \times Y - \Sigma e_i \geq 0$$

Proof: Y = 1 from first equation $\Sigma e_i \geq 1$ implies that at least one of $e_i = 1$. Similarly Y = 0 from the second equation $\Sigma e_i \leq 0$ implies that all the variables are 0.
To put a constraint that y = $\neg e$ (not e). We can put a constraint e + Y = 1. Note that 'e' and 'Y' are integers and range from 0 to 1, thus exactly one of 'e' and 'Y' is equal to 1 at a time.
Building on the basics, any compound formula derived using $\wedge$ and $\vee$ and $\neg$ can be represented using ILP (potentially using temporary variables).
For example to represent z = $(e_1 \wedge \overline{e_2}) \vee (e_3 \wedge e_4 \wedge e_5)$, we can write the following ILP. Temporary variables $Y_1 = e_1 \wedge \overline{E_2}$ and $Y_2 = e_3 \wedge E_4 \wedge e_5$ have been used here. We use one temporary variable for each conjuction.

Var1	**Var2**	**Var3**	**Var4**	**output**
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	**1**
0	1	0	1	**1**
0	1	1	0	0
0	1	1	1	0
1	0	0	0	**1**
1	0	0	1	**1**
1	0	1	0	**1**
1	0	1	1	**1**
1	1	0	0	**1**
1	1	0	1	**1**
1	1	1	0	0
1	1	1	1	0

Figure 2.1: Truth Table Formed from Example in Figure 1.4

Writing boolean expressions with disjunctions ($\vee$) between conjunctions ($\wedge$) of variables is called the sum of product (SOP). We wish to represent the permitted columns in Hydra in SOP form and then form ILP constraints on them. We can reduce our problem of representing new columns to minimize boolean expressions so that we can form an ILP in the way shown above. We introduce boolean variables equal to the number of entries in a column. For each permissible column (consisting of only 0s and 1s), we put a 1 in the truth table of boolean function and then reduce this boolean function.

Continuing from the example in Figure 1.3, the permitted columns are [1,0,0,0], [1,0,1,0], [1,0,1,1], [1,0,0,1], [1,1,0,0], [1,1,0,1], [0,1,0,1], [0,1,0,0] which are the columns of the coefficient matrix and is also shown in Figure 1.4. The corresponding truth table for this is shown in Figure 2.1.

2.3.3 Artificial Variable Reducer

An LP problem in Hydra can be thought of as finding a non-negative solution to Ax = b, where A is the coefficient matrix for the equations, x is the vector of variables, and b is the vector of right-hand side (RHS) values. Since in each iteration at most one artificial variable can be removed, the processing time of the simplex method depends on the number of artificial

variables present. We use the special property of LP in Hydra of having only 0, or 1 as a coefficient to reduce this number.

The aim is to assign values to the variables such that there is a BFS, either with or without an artificial variable for an equation. Note that there can be at most one artificial variable per equation. As a preprocessing step, the algorithm first finds equations with RHS equal to 0 and assigns all the variables as 0 in them. Since these variables have been assigned a value, they are removed from any other equation they come in. In the next steps the algorithm picks equations having minimum b(call it b_i) value, greater than 0, and assigns a variable(call it x_{ij}) with that value, if an unassigned variable in the corresponding equation is present, otherwise an artificial variable(a_i) is introduced in the corresponding equation. If a variable is assigned a value, then all the other *remaining variables* in the corresponding equation become 0 and the variable is removed from every other equation it appears in by subtracting the value of RHS with the value of this variable. Note that at no point RHS value becomes negative, this is so because we are always taking the minimum b_i, thus also making sure that all the variables get

positive values.

Algorithm 2: Artificial Variable Reducer

```
1  Input Cardinality constraints
   Result: Assigned variables, artificialVars
2  artificialVars = ϕ
3  for equations with RHS = 0 do
4  |  assign variables = 0
5  |  remove variables from other equations
6  end
7  while min. b_i > 0 exists do
8  |  if an unassigned variable x_ij exists then
9  |  |  x_ij = b_i;
10 |  |  assign remaining variables = 0
11 |  |  remove remaining variables from other equations;
12 |  |  for equation k having x_ij do
13 |  |  |  b_i -= x_ij
14 |  |  |  remove x_ij from eqn. k
15 |  |  end
16 |  else
17 |  |  add an artificial variable a_i ;
18 |  |  assign a_i = b_i
19 |  |  add a_i to artificialVars
20 |  end
21 end
```

2.3.4 Removing Cycling in Column Generation

The issue of cycling (subsection 2.1.5) has been encountered in our case as well. Bland's rules are applicable only for the simplex algorithm where the columns are enumerated and the entering columns can be ordered (by their indexing), but in column generation, we have information only of the columns in the basis plus the entering column only. In theory, since all the columns are equivalent to boolean values, we can choose the ordering to be dictated by the value it represents. For example, consider two columns [1, 0, 1, 0] and [1, 0, 0, 1], the first column represents the number 10 in decimal (considering left most value to be the most significant bit) and the second value denotes 9; thus the latter would come before the former in the ordering.

This fact can be used if we can generate all the possible entering columns.

A workaround to finding all the entering columns is devised by spending more computations when a cycle is detected. Recall that in column generation, an entering column can be found by finding a column having $f = c_B \times A_B^{-1} \times e$ - e_i value less than 0, where e is the representative entering column. If there were no cycling, to generate an entering column, simply putting an optimization function that minimizes f with the constraints generated to represent columns would do. To generate all the possible entering columns, we first minimize f to find the first entering column. Let the value of f found be f_1. To find the next possible entering column, the same constraints and the same optimization function are put with an additional constraint that $f > f_1$. Other constraints and optimization objectives ensure that we find an entering column, and the last constraint ensures that no repeated columns are returned. Similarly, to generate i^{th} additional column, constraint $f > f_i$ is put, this is done till there are no more columns possible. In our implementation, instead of finding values the ordering of the entering columns, we simply pick any random column, and it successfully breaks the cycle.

Chapter 3

Adapted Boolean Expression Minimizer

3.1 Prerequisites: QM Algorithm

Boolean functions are used extensively in digital circuits, which have wide ranging applications. Such functions are minimized before their use for performance and cost optimization without compromising on the quality of the solution. 'Don't care' terms which can represent either 0 or 1 at the same time which are usually present while minimizing circuit problems are not present in our functions . One of the common ways to minimize boolean functions includes minimizing using boolean identities like idempotent, distributive, and absorption laws. Another popular way to do the same is using Karnaugh maps, also popularly knows as K-map. However, both the techniques are not scalable with respect to the number of variables, thus for solving large problems, the Quine-McCluskey algorithm (QM) is used. In this section, we discuss the important steps of the QM algorithm, followed by our modifications to it to make it more efficient in the context of our problem in the next section.

QM Algorithm [8] was first proposed by W. V. Quine in 1952 and later improved by E. J. McCluskey Jr in 1956. The algorithm can work with any number of variables and can be implemented on a computer, the running time and space required depend on the number of variables. We will first describe the setup of the problem with the expected input and output with an example, alongwith defining some terms in our context for ease of understanding.

- **Input variables** - A set of boolean variables defining the domain of the problem and the length of each (min)term of input.

- **Minterm** - A product term having all the input variables either in normal or complemented form.

- **Implicant** - An implicant is a term which when set (equal to 1) also implies that the output is 1. Every minterm is an Implicant.

- **Prime Implicants (PIs)** - An implicant of the given function which cannot be covered by an implicant having fewer variables (normal or complemented), i.e., the removal of any variable from from this term would make it a non-implicant. A PI can potentially represent several minterms.

- **Essential Prime Implicants (EPIs)** - Prime implicants covering minterms which are not covered by any other PIs, thus the inclusion of these PIs are absolutely necessary.

The algorithm requires to know all the minterms or equivalently the presence of the columns which are permitted in the analogical world of Hydra. Consider the example given in section 1.2. The permitted columns in it are [1,0,0,0], [1,0,1,0], [1,0,1,1], [1,0,0,1], [1,1,0,0], [1,1,0,1], [0,1,0,1], [0,1,0,0]. We have 4 *input variables* in this problem which is equal to the length of each column. The boolean expression that we want to minimize is:

$$z = 1000 \vee 1010 \vee 1011 \vee 1001 \vee 1100 \vee 1101 \vee 0101 \vee 0100.$$

There are 8 *minterms* in the expression, each representing a column. The minimized expression of this problem is:

$$z = 10xx \vee x10x$$

where x signifies that the position can be both 0 or 1. The PIs are $10xx$, $x10x$, and $1x0x$. The EPIs are $10xx$ and $x10x$. $1x0x$ is a PI but not EPI since all the terms covered by it, i.e, $1000, 1001, 1100, 1101$, are covered by some other PI. All the EPIs are always part of the minimized expression while a subset of PIs which are not EPIs are selected. The selection is equivalent to solving the classical Set Cover problem.

A high-level QM algorithm is given in Algorithm 3. We explain the important details of it which are relevant to our work. In the first step, all the minterms are grouped according to the number of 1s present in it. For example, in the running example, the groups would be as follows -

- group 0 - {}

- group 1 - {1000, 0100}

- group 2 - {1010, 1001, 1100, 0101}
- group 3 - {1011, 1101}
- group 4 - {}

The ith group can have a maximum of $\binom{n}{i}$ elements. Iteratively the elements from *consecutive groups* are 'merged' to form new elements and deleting the old ones till no more merging is possible, at which point the remaining terms are the Prime Implicants. Two terms are merged if and only if they have all the values same except at one position, and after merging a '-' is placed in place of the differing bit and is added to the appropriate group for the next iteration and the terms which merged are deleted. For example, terms 0100 and 1001 cannot be merged as they are differing at 3 positions but 0100 (from group-1) and 0101 (from group-2) can be merged to form a new element 010- which will then be put to group-1 (since it has one bit set).

In the *modified QM Algorithm (MQM)* presented in section 3.2, we modify lines 4-6 of Algorithm 3 so that 'region partitioning' algorithm can be skipped which takes up a lot of memory while finding all the regions. The rest of the steps remain the same. Till the 24th line, PIs are found, and then first the EPIs are found and then a subset (which need not be a proper subset) from PIs that are not EPIs are selected as part of the minimized expression. This minimized expression is passed on to the column generation module which first makes linear constraints from the minimized expression and then solves the LP.

Algorithm 3: QM Algorithm

```
Result: Minimized Boolean Expression
input: Minterms
1  numVars = number of variables
2  groups[numVars + 1] = dictionary
3  primeImplicants = []
4  for minterm in Minterms do
5      numOnes = getNumberOfSetBits(minterm)
6      groups[numOnes].add(minterm)
7  end
8  while true do
9      for i=0..numvars do
10         groupI = groups[i]
11         groupJ = groups[i+1]
12         for termI in groupI do
13             for termJ in groupJ do
14                 if Difference of 1s in termI and termJ ==1 then
15                     newTerm = combine(termI, termJ)
16                     groups[i+1].add(newTerm) // Considered in next iteration of while
                        loop
17                 end
18             end
19             if termI does not combine with any termJ then
20                 primeImplicants.add(termI)
21             end
22         end
23     end
24     if No terms are merged then
25         break
26     end
27 end
28 essentialPIs = getEssentialPIs(primeImplicants, minterms)
29 reducedPIs = getReducedPIs(essentialPIs, primeImplicants, minterms)
30 return essentialPIs + reducedPIs
```

3.2 Modified QM with Minterms on the Fly

Using the QM algorithm, we would get the constraints formed on the entering column, and using the minimization objective function mentioned in section 2.2 with these constraints, we can find an optimized entering column which is equivalent to solving one iteration of simplex in the original problem. We propose a modification to the algorithm so that all the minterms need not be known at the same time, thus eliminating the need to do region partitioning. We propose to generate minterms (equivalently regions) on the fly. This is derived from the observation that absence of certain *intersection of CCs* imply the absence of certain columns. We would like to point that the absence or presence of certain regions (or columns) do not imply the absence or presence of any other region (or column). This is summarized in Figure 3.1. For m CCs (or m variables), all the 2^{2^m} possibilities of columns are permitted[1].

Observation	Implication
Region is present	No Implication
Region is absent	No Implication
Intersection Region is present	No Implication
Intersection Region is absent	No superset Intersection Region can be present

Figure 3.1: Implications from Observations

But if we know that certain combination of CCs do not intersect at all, then we can squeeze out more information about the columns that are possible. A column with entries as 1 and 0 imply the overlap between CCs having entries as 1 and the negation of CCs having entries as 0. E.g the column [1,1,0,0] is equivalent to the region {1,2} and can be interpreted as the intersection of $CC_1 \wedge CC_2 \wedge \neg CC_3 \wedge \neg CC_4$, where $\neg$ is the negation operator and $\neg CC_i$ represents all the space that is not covered by CC_i. We now introduce the concept of intersection regions ($region_l$). We call a region as 'intersection region' if while computing it, we do not consider all the CCs (a normal region has information about either the presence or absence of all the CCs) and it is labelled using only a subset of CCs. Also, while computing $region_l$, we never check for the absence of a CC. For example, an intersection region can be formed by CC_1 and CC_2 but not with CC_1 and $\neg CC_2$. The presence of a $region_l$ does not give any extra information, but the absence of it does. The absence of a $region_l$ implies the absence of all the (normal and intersection) regions where the CCs involved in the $region_l$ are all present, i.e., any superset

[1] There can be 2^m regions and each can either be present or absent

region having these CCs and more. For example, if we search for the $region_l$ of CC_1, CC_2 and find it absent, then no region can have $\{1, 2\}$ together in its label, i.e region $\{1, 3\}$ or region $\{2, 3\}$ is possible but region $\{1, 2\}$ or region $\{1, 2, 3\}$ are impossible. In columnar world it means any column of the type [1,1,x,x] is not permitted, where x can take value of either 0 or 1.

We use the above idea while forming the groups in the QM algorithm. Instead of knowing all the minterms in the very beginning and forming groups, we generate groups on the fly as and when needed by 'probing' regions (in case its corresponding intersection region exists). For group-1 we need to find the presence or absence of regions satisfying exactly one CC, e.g $\{1\}$, $\{2\}$ etc. To find (probe) if region $\{1\}$ is present or not, we need to find if there exists a region which satisfies CC_1 but not CC_2, CC_3, CC_4 (assuming 4 CCs). All the regions of group-1 are necessarily probed. While probing for group-2, we also check if intersection regions are absent (intersection regions for CCs in group-1 are trivially present), if they are absent we won't be probing the relevant regions (and intersection regions) in subsequent groups. The dependency of intersection regions can be represented as a lattice structure. The lattice structure formed for the CCs in Figure 1.3 is shown in Figure 3.3. Notice that each layer in this structure represents a group in the QM algorithm. The light green (or orange) circle below a intersection region signifies the need to probe the intersection region (or not). The green arrow (or red sign) above the region signifies its presence (or absence).

There are no circles below the 1-sized and 2-sized combinations because they need to be necessarily checked and there is no decision to be taken. We notice that in Figure 1.3, CC_2 and CC_3 do not intersect (see Figure 3.2), thus we put a red sign on the top of region $\{2, 3\}_l$ in the lattice structure. The absence of $\{2, 3\}_l$ (intersection region formed from CC_1, CC_2) also implies the absence of $\{1, 2, 3\}_l$ (and thus of $\{1, 2, 3\}$ as well) and $\{2, 3, 4\}_l$ and they need not be probed. Similarly, $\{1, 2, 3, 4\}_l$ need not be probed since $\{1, 2, 3\}_l$ (or $\{2, 3, 4\}_l$) is not present. In the columnar world, it is equivalent of saying that no column of type [x,1,1,x] can exist where x can be either 0 or 1. It is to be noted that the absence of a region in group i eliminates the need to probe i regions in the next group (if i $\leq n/2$) which is supposed to have a snowballing effect, and the effect is larger when the absence is detected in the earlier stages.

This approach of region probing is helpful because it removes the need for doing region partitioning. Region partitioning is the algorithm used in Hydra to partition the whole space of attributes into regions. Also, each region and its structure are explicitly stored in the memory, but after on the fly generation, only regions (corresponding to the columns of basic solution) having non-zero cardinalities need to be generated. It is to be noted that even with this opti-

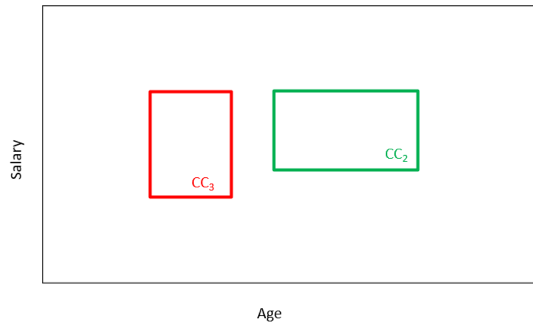

Figure 3.2: Intersection Region of {2, 3} Based on Figure 1.3

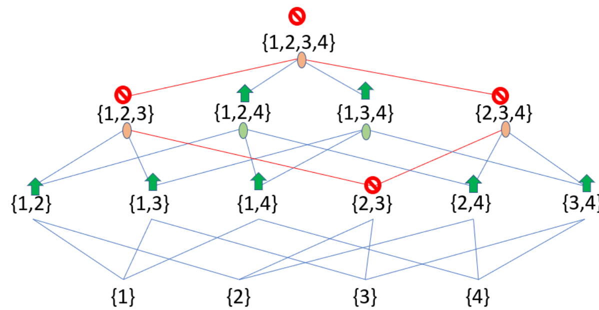

Figure 3.3: Lattice Structure Formed by $regions_l$

mization, the number of regions probed can be exponential giving the algorithm an exponential time complexity. The memory required may also be exponential in the worst case as all the elements of a group are produced at a time (if required) which can in the worst case go up to $\binom{n}{n/2}$. However, there is also an option to sacrifice time for memory by not producing the whole group at once. In our implementation, we produce the whole group at a time.

Algorithm 4: Iterative region generator

```
   Input: basicRegions
   Output: regions
1  basicRegions = getCCInSpace()
2  intersectionRegions = basicRegions
3  while intersectionRegions.size() != 0 do
4      intersectionRegions = getNextIntersectionRegions (basicRegions,
        intersectionRegions)
5      regions = getRegions (basicRegions, intersectionRegions)
6  end
7  Function getNextIntersectionRegions(basicRegions,
    currentIntersectionRegions):
8      List<Region> intersectionRegions;
9      Map<regionLabel, Boolean> isIntersectionLabelPresent;
10     Map<regionLabel, Region> labelToRegion;
11     for region in currentIntersectionRegions do
12         List<Region> supersetRegions = getSupersetRegions(region)
13         for supersetRegion in supersetRegions do
14             if supersetRegion is not empty then
15                 isIntersectionLabelPresent.add(supersetRegion.label, true)
16                 labelToRegion.put(supersetRegion.label, supersetRegion)
17             else
18                 isIntersectionLabelPresent.add(supersetRegion, false)
19             end
20         end
21     end
22     for {label, region} in labelToRegion do
23         intersectionRegions.add(region)
24     end
25 return intersectionRegions
```

```
1  Function getRegions(basicRegions, currentIntersectionRegions):
2      List<Region> newRegions
3      for region in intersectionRegions do
4          potentialRegion = region for i = 1...numCC do
5              if i not in region.label then
6                  potentialRegion -= basicRegions[i]
7              end
8          end
9          if potentialRegion is not empty then
10             newRegions.add(potentialRegion)
11         end
12     end
13 return intersectionRegions
```

Chapter 4

Implementation Details and Experimental Evaluation

4.1 System and Implementation Details

We have implemented the new solver design, including the main *LP solver* (using simplex algorithm), *column generation* (ILP) module, *the modified QM algorithm* (using the intersection monotonicity property) to replace the existing modules of region partitioning and the existing LP solver, namely Z3 solver. The new implementation has been done in approximately 1500 lines of code in Java and has been completely integrated with the existing Hydra code base. Google OR-Tools [2] has been used for column generation in each iteration of the simplex algorithm. All the numerical computations are done in fractions for numerical stability. In the next subsection, we compare our performance against Hydra.

The AQPs for our system were obtained from queries on the TPC-DS [3] decision-support benchmark database. We have taken the same workload or set of queries as in Hydra [7]. The workload in Hydra featured 131 distinct queries distributed over tables, which were created by customizing the 99 queries of the TPC-DS benchmark, such that only non-key filter predicates and PK-FK joins were retained, and all nested queries were separated into independent subqueries. Although not relevant to our work, the AQPs for these were generated on PostgreSQL [6] engine. In our experiments, we focus on individual tables and pick CCs only for them.
The TPC-DS benchmark is a popular and widely accepted evaluation standard for complex queries in the database community. The design has been made keeping large volumes of data and real-world business questions in mind [5]. There has been an evolution of the benchmarks, and TPC-DS is one of the active ones as of today [4]. More details about the benchmark can

be found in [5].

All the experiments were run on DELL system, having 16GB of memory and 2.8GHz 8 core processor running Ubuntu 18.04. The memory allocated for running the Java process in all the cases was 13 GB.

4.2 Results

In this section, we compare the memory and time taken for our proposed method and Hydra without the subview optimization for a fair comparison of the methods. We also show the time break up between the QM algorithm and LP Solving in the new solver, which takes up the majority of the time. We do this experiment for two different tables from the original Hydra paper on queries also taken directly from the paper. We do the experiments with the number of CCs as an independent variable, with 5 CCs as the step size. When the number of CCs are incremented from x to $x + 5$, all the CCs in the former case are present in the latter case as well. In other words, we consider a superset of the CCs while increasing the number of CCs. The considered test cases for comparing the methods are the same and were allotted an equal amount of resources.

4.2.1 Peak Memory Usage Comparison

Graphs Figure 4.1 and Figure 4.2 show the peak memory usage for two tables with the increase in the number of CCs.

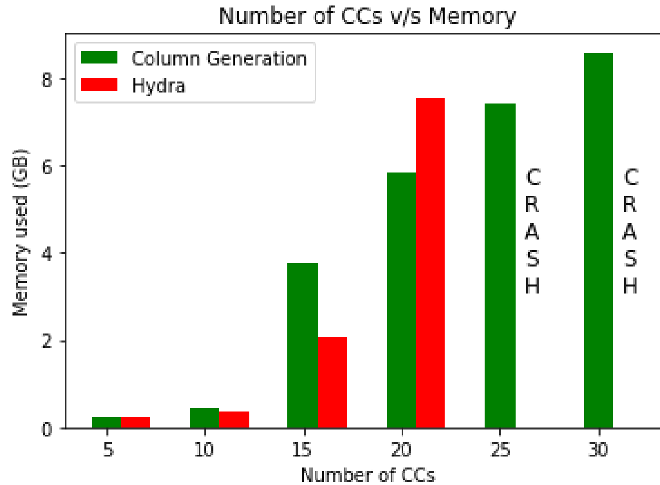

Figure 4.1: Memory Comparison for Table 1

Hydra runs out of memory while partitioning the attribute space to form regions. The main memory bottleneck for our method is in the QM algorithm module, specifically the lattice

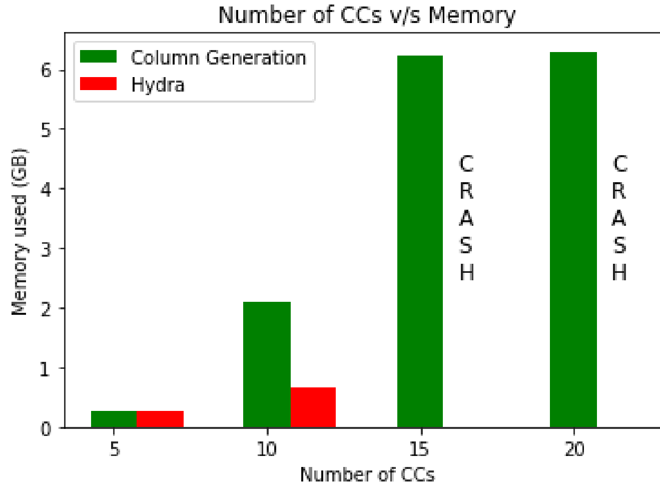

Figure 4.2: Memory Comparison for Table 2

traversal, to find the regions.

Inferences and analysis: It can be observed that the memory requirements for Hydra increase much faster with the number of CCs. Also, there is no strict theoretical relationship of memory consumption between the two methods, as shown in Figure 4.1 for 15 CCs and Figure 4.2 for 10 CCs, where the proposed method takes more memory. Note that the extra memory consumption is not due to the column generation module, which deals in a lesser number of columns strictly, but due to the lattice structure computation (Figure 3.3) in the QM algorithm.

The *crash can occur at different points on different tables* due to other factors that have not been closely discussed, like the number of attributes in the table, the physical shape of the regions, etc. Recall that regions are multidimensional physical structures with the dimension being equal to the number of attributes, and they need not be any regular simple structures like a rectangle (e.g. region $\{1, 3\}$ in (Figure 1.3)). The number of attributes for table 1 is 20, while that for table 2 is 35, which can be the reason for an early crash in the latter case.

The peak memory usage for our method is *monotonically increasing* but not strictly monotonical. The memory bottleneck for our module is in the QM algorithm component, as the lattice structure of intersection regions is traversed. Since we process one layer of lattice at a time, the peak memory usage is somewhat dictated by the highest number of $region_l$ in a layer, which may or may not increase with the number of CCs. For example, consider a case where one in which we have $region_l$ $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, where the first three $region_l$ belong to layer 1 and the last $region_l$ belongs to layer 2. Now, consider two cases, one in which with

the increase in the number of CCs $region_l$ {4} gets added while in another $region_l$ {1, 3} gets added (in addition to the old $region_l$). In the first case, the number of $region_l$ in layer 1 and layer 2 are four and one respectively, and in the second case, it is three and two. Since the peak memory usage would be approximately defined by the maximum $region_l$ in a layer, there would not be an increase in peak memory usage in the second case while there would be an increase in the first case. This behaviour of roughly equal memory usage is also seen in table 2, when increasing from 15 to 20 CCs.

4.2.2 Time Comparison

Figure 4.5 and Figure 4.6 show the break up of total time for the column generation method for QM algorithm and Column generation. The crashes that are shown are due to memory limitations mentioned in subsection 4.2.1. The times are plotted on a logarithmic scale.

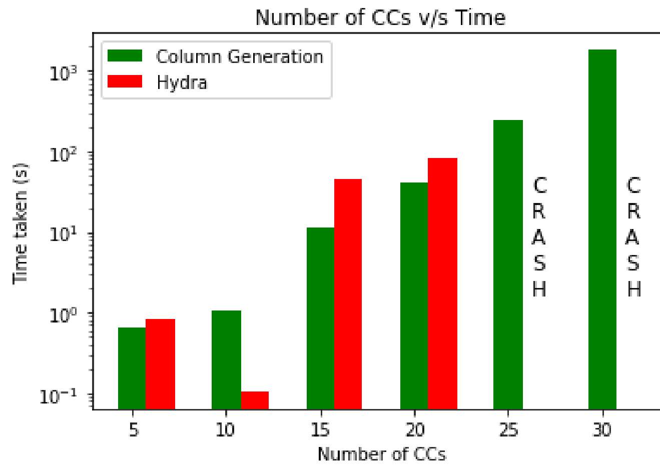

Figure 4.3: Time Comparison for Table 1

Inferences and analysis: The times of both methods do not have any significant differences. Also, since the process of regenerating DB is an offline process, the difference becomes insignificant. There is an 'outlier' value for Hydra in table 1 with 10 CCs. This is because the LP Solver converged to a solution quicker in that case. Since Simplex is an iterative algorithm where 'steps' are made to improve the optimization value at each step (except when degeneracy or cycling is detected), and the improvement depends on the combination of variables (or columns) in the current basis, the number of iterations and thus the running time depends on how the basis changes. In this particular case, the steps taken by the algorithm were good, but in general, it can be expected that with more variables, the solver should take more time.

Depending on the heuristics applied and the steps taken in each iteration, the algorithm

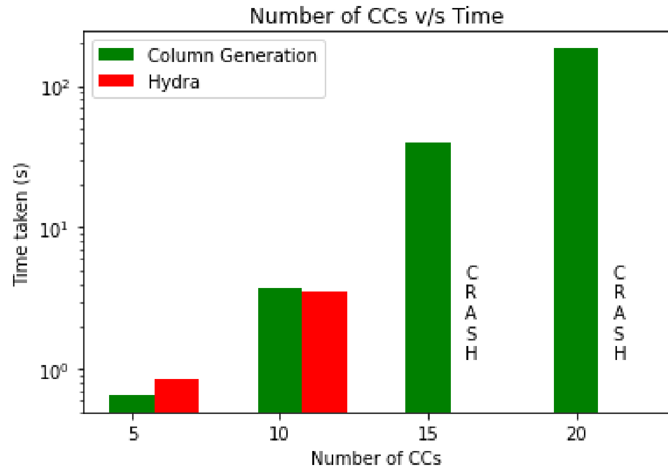

Figure 4.4: Time Comparison for Table 2

may converge fast or slow for Hydra. However, for Column Generation, the time variability would be low firstly due to the smaller size of LP being solved and secondly because the QM takes the bulk of the time, which is expected to increase with the number of CCs. The time portion taken by both the modules is shown in Figure 4.5 and Figure 4.6.

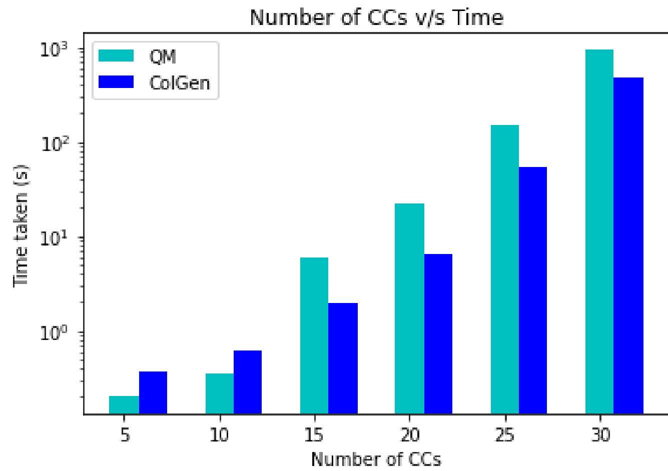

Figure 4.5: Time Break Up for Colgen for Table 1

The break of time clearly shows that most of the time is spent minimizing boolean expressions using the QM algorithm. It was noticed that the majority of time in QM algorithm was spent in finding the regions by traversing the lattice structure. Note that the y-axis has been plotted on a log scale.

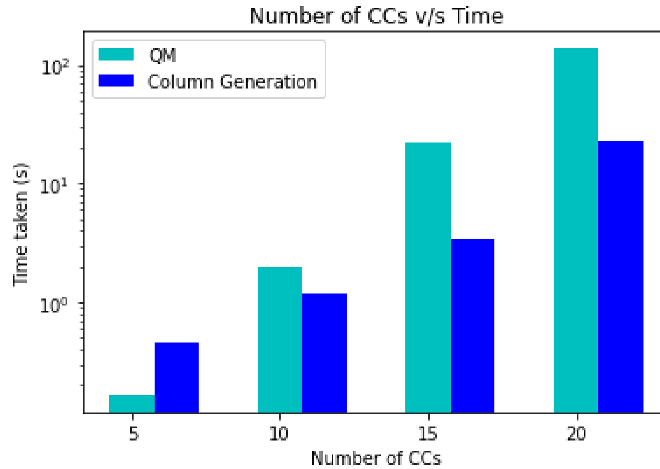

Figure 4.6: Time Break Up for Colgen for Table 2

4.2.3 Accuracy Comparison

It is to be noted that the number of tuples returned for each CC should be a positive integer; otherwise, it is not meaningful. In Hydra, the problem is formulated as an Integer Linear Program, which returns only optimal integral solutions, which is what we want. At the same time, in the new LP, only the column generation part is integral (in fact boolean), while the main LP can return non-integer values because it has been formulated for Simplex Algorithm. Note that the solution, even in the new solver, cannot be negative as positive values are assumed in the standard formulation of simplex itself. Experimentally, it was found that almost all the values had 0 error, and a very few had minimal (absolute values ≤ 5) errors in the new solver. The small errors on a relative basis (to the actual values) are negligible and are harmless with respect to the objective of performance testing.

4.2.4 Time and Memory with Increase in Number of Tuples

Although the time and space requirements are independent of the numbers in RHS of the CCs in both Hydra and the proposed method, i.e., row cardinality, we anyway did a sanity check to ensure that is the case. The experiments were done with each row cardinality scaled up 10 times.

Figure 4.7 and Figure 4.8 confirm data scale independence for memory and scale, respectively.

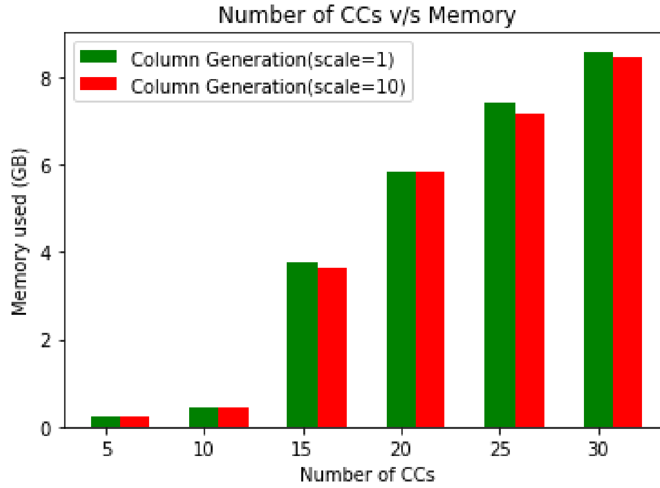

Figure 4.7: Memory Comparison with Scale 10

4.2.5 Comparison with Hydra With optimization

All the experiments done on Hydra with relation (LP) decomposition optimization (Figure 1.5) for the scales shown in subsection 4.2.1 and subsection 4.2.2 resulted in runtimes lower than 10 seconds and peak memory usage of less than 1 GB.

The result is so because the LP is broken into smaller segments, and then the memory-intensive 'region partitioning' step is done in each segment. This optimization of decomposition is not compatible with the optimizations proposed in the thesis, and for a fair comparison of the ideas presented, we make a comparison without optimization in Hydra. The incompatibility comes from the presence of non-boolean values in the optimization in Hydra. Along with 0 and 1, -1 also occurs in the coefficient matrix of the LP, due to which column generation module cannot be used as-is since the representation of columns is done only for boolean values.

However, to address this issue, we have proposed a theory as an approach to make both the optimizations (decomposition and methods proposed here) compatible in subsection 5.1.3.

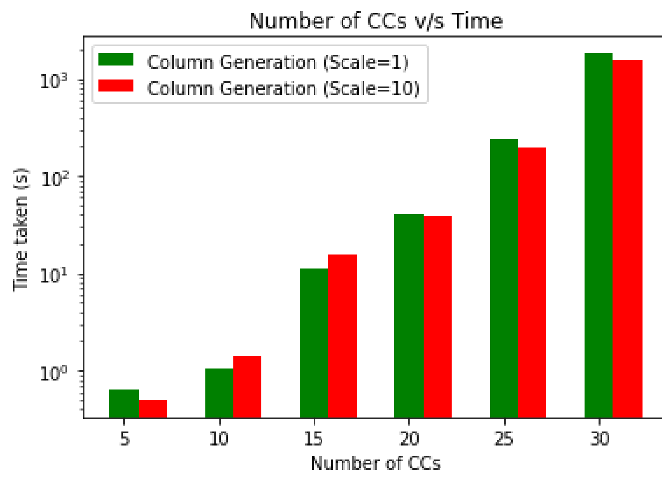

Figure 4.8: Time Comparison with Scale 10

Chapter 5

Extension

5.1 Theoretical Extension

5.1.1 Prerequisite: Dantzig Wolfe Decomposition Algorithm

Dantzig Wolfe Decomposition algorithm (DW) (referred from [9]) is a more efficient method (in terms of time) for solving LPs which have a special block angular structure. In this structure, many of the constraints are clustered (the clusters are foreknown to solving). All of the constraints in a cluster have a non-zero coefficient for only a few variables. The same rule applies for the remaining constraints and they cannot have non-zero coefficients for variables already within some cluster. Each such cluster is called a subproblem. There can be equations that do not fit the above definition and can have non-zero coefficient values for any variable; those constraints are part of the master problem.
The matrix below shows the same visually. Here A is the original matrix divided into k segments, where A_i form the subproblems and L forms the master problem.

$$\mathbf{A} = \begin{bmatrix} \mathbf{L}_1 & \mathbf{L}_2 & \cdots & \mathbf{L_k} \\ \mathbf{A}_1 & & & \\ & \mathbf{A}_2 & & \\ & & \ddots & \\ & & & \mathbf{A_k} \end{bmatrix} \tag{5.1}$$

We say that the $\mathbf{x} \in R^n$ is divided into N-segments,$\mathbf{x_1} \in R^{n_1}, \mathbf{x_2} \in R^{n_2} ... \mathbf{x_k} \in R^{n_k}$, where $\mathbf{x_i} \in R^{n_i}$ is the vector of variables associated with the ith segment having n_i variables.

We have two types of constraints:

1. Subproblem: These are the intra-segment constraints that only exist on a set of variables corresponding to a single segment. Each A_i in the above formulation forms a subproblem.

2. Master Problem: These are the inter-segment constraints that exist over variables corresponding to more than one segment(denoted by L). These can also be termed coupling or connecting constraints.

Note that if we do not have any master constraints, then all the subproblems are disjoint (since they do not have any common variables) and can be solved independently for optimization function defined on their subset of variables. This independence and disjunction property of subproblems form the intuition behind the algorithm. Assume that the master constraints are not present, then the subproblem forms a space in the form of a convex polytope (like in Simplex) and each subproblem will have a set of corner points defined on its variables. Now if we add the master constraints, it will reduce the feasible space but linearly only. Due to the addition of new constraints, the space will have new corner points, but the new corner points can be written as a convex combination of the old corner points (corner points for each subproblem without the master subproblem).

If we write the original problem as:

maximize

$$c^T x$$

subject to

$$Ax = b$$

$$x \geq 0$$

$$x \in X$$

where X is composed of all combinations of corner points in the subproblems, e.g., if there are 3 subproblems having 3, 5, 7 corner points, the total number of corner points in X will be 3*5*7 = 105. Let us denote the corner point X by breaking it into disjunct parts according to the subproblems. If the corner points of problem i is represented as X_i, then X can be represented as $\{X_1, X_2, .., X_k\}$. Since x can be represented as a convex combination of X_is, we

can write

$$x = \sum_{i=1}^{k} \lambda_i \times X_i$$

where

$$\sum_{i=1}^{k} \lambda_i = 1$$

$$\lambda_i \geq 0$$

(since the combination is convex)

Assuming that all the corner points for the subproblems are known, we can reformulate the problem with λ_i as the variables, as shown below.

maximize

$$\sum_{i=1}^{k} c^T(X_i\lambda_i)$$

subject to

$$\sum_{i=1}^{k} AX_i\lambda_i = b$$

$$\sum_{i=1}^{k} \lambda_i = 1$$

$$\lambda \geq 0$$

Note that it was assumed that all the corner points of all the subproblems were precomputed, but that is costly to compute and store and thus is not done. *Instead, in each iteration a corner point is generated for each subproblem using the column generation technique.* Solving multiple smaller subproblems is better than solving the problem as a whole by applying the Simplex Algorithm without breaking down the problem. Also, since the subproblems are independent, parallel processing can be easily used. The advantage of DW is that at step, we are solving an optimization problem in the fewer number of variables and constraints for each subproblem.

5.1.2 Prerequisite: Subview Optimization in Hydra

We have compared our method with Hydra in the whole thesis where the coefficient matrix has only boolean values. Hydra has an optimization on top of this LP formulation to form an alternate LP with an equivalent solution with respect to the cardinalities returned. It first

constructs a graph with the attributes of the table as vertices and edges are present between two vertices if there is a query involving a pair of attributes. If more than two attributes are involved in the query, then edges are put between every pair of the involved attributes. Any cycles in the graph having a length greater than 3 are broken by adding artificial edges in the 'diagonal' of the cycle. In this graph, maximal cliques are found and equations for the concerned CCs are added. The equations that are added are like the equations that have been discussed throughout the thesis, i.e., having boolean values. This method would decompose the LP in independent components except for the fact that two cliques may share one or more attributes and the solutions from the two LPs (from the two cliques) may give incosistent results for the common attributes. To ensure consistency in the solution additional constraints are added across cliques which are known as consistency constraints. The coefficient of variables in the coefficient matrix can take values from 0, -1, 1. We do not discuss the additional details of consistency constraints here; the details can be found in [7].

5.1.3 Applying Dantzig Wolfe to Hydra

As pointed in subsection 5.1.2, the coefficients in the matrix do not remain as boolean after the subview optimization and the modifications presented in this thesis cannot be directly used with the subview optimization. However, the structure of the LP follows the structure of DW (subsection 5.1.1), where the consistency constraints are the master constraints (connecting the cliques) and the intra clique constraints form the subproblem. All the theory and results shown in the thesis correspond to solving a subproblem in this proposed framework.

5.2 Implementational Improvements

5.2.1 Parallel Programming

The two most time-consuming parts of the whole pipeline are the Modified QM algorithm (MQM) and the Column Generation module. Both the modules have tasks that can be termed as 'embarrassingly parallel'.

The first step of the MQM, i.e. grouping of terms is already implemented in parallel (Algorithm 4 and Algorithm 5), which is the bottleneck of the algorithm when the lattice is 'dense'. Recall that this part replaces lines 4-6 in Algorithm 3. The rest of the algorithm from line 8 onwards can be parallellized.

Although the CG module is easily parallellizable, little benefit is expected out of it due to the scale of the problem being solved. The time requirement for CG is majorly due to the number of iterations and lesser due to the time taken in each iteration.

5.2.2 Alternate Algorithm for Lattice Traversal

There are several ways to travel the lattice structure mentioned in section 3.2. While going from one level to the next we can have the following strategies-

- Check for all the supersets of the intersection regions in this level
- Use a candidate generation and pruning strategy to check for only the necessary supersets like in [18].

We are using the first approach currently, and a change may speed up the execution. To get an intuition on how the second algorithm may be more efficient, consider the following example where the presence of intersection regions $\{1, 2\}$ and $\{2, 3\}$ has been detected while $\{1, 3\}$ is absent. In the first case, intersection region $\{1, 2, 3\}$ will be checked while in the second case, firstly $\{1, 2, 3\}$ will be made as a potential candidate in the candidate generation step and will be removed in the pruning stage after detecting that $\{1, 3\}$ is absent (thus $\{1, 2, 3\}$ can not be present. This implementational change should save computation time for the $region_l$ that are not checked. Note that the amount of memory that can be saved is depends on the lattice structure. In the first case the labels of the regions (but not the whole regions) which might not be calculated in the second approach are stored.

Chapter 6

Conclusion and Future Work

We have approached the problem of Database regeneration for mimicking the output cardinalities at each operation of processing with respect to pre-defined workload and AQPs, with the objective of making the system more memory efficient. We have dived into Linear Programming (LP) theory and considered the special case of LP in our case. We proposed the use of the column generation technique, which eliminates the requirement to store the complete coefficient matrix of the LP by using linear constraints. The constraints are generated by solving a boolean expression minimization (BEM) problem which takes the columns as input. We also propose modification over the QM algorithm, which is used to solve BEM using monotonicity property of set intersection.
No LP decomposition is done to make the LP simpler in our approach. For a fair comparison we compared our method with Hydra without decomposition algorithm. In the experiments, it was seen that the requirement of memory for the latter case grows faster than the former. Comparison with the decomposition algorithm in Hydra proved to better than our approach in terms of memory.

We have proposed the use of Dantzig Wolfe decomposition (subsection 5.1.3) as future work to combine the benefits of both the optimizations proposed in this thesis and already present in Hydra. We have also proposed implementational improvements (section 5.2) which are expected to have better run times.

Bibliography

[1] Z3 Solver: https://github.com/Z3Prover/z3 9

[2] Google OR-Tools: https://developers.google.com/optimization 33

[3] TPC-DS: http://www.tpc.org/tpcds 33

[4] TPC Benchmarks: http://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp 33

[5] TPC-DS Specifications: http://tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v3.1.0.pdf 33, 34

[6] PostgreSQL: https://www.postgresql.org/ 33

[7] A.Sanghi and R.Sood and J.R.Haritsa and S.Tirthapura. Scalable and Dynamic Regeneration of Big Data Volumes. *EDBT Conference*, 2018. 1, 9, 17, 33, 44

[8] E.J.McCluskeyJr. Minimization of Boolean Functions. *Bell System Technical Journal*, 1956. 5, 18, 24

[9] G. Srinivasan. Operations Research : Principles and Applications. *PHI Learning Private Limited*, 2010. 14, 15, 16, 17, 41

[10] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *Proc. of the Intl.Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008 4

[11] C. Binnig, D. Kossmann, E. Lo and M. Tamer Özsu. QAGen: generating query aware test databases. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 2007.*

[12] A. Arasu, R. Kaushik, and J. Li. Data Generation using Declarative Constraints. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 2011.* 7

[13] N. Bruno and S. Chaudhuri. Flexible Database Generators. *Proc. of the 31st Intl. Conf. on Very Large Data Bases, 2005.* 1, 7, 9

[14] J. Gray, P. Sundaresan, S. Englert, K. Baclawski and P. J. Weinberger. QuicklyGenerating Billion-record Synthetic Databases. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 1994.* 6
6

[15] E. Lo, N. Cheng, W. W. K. Lin, W. Hon and B. Choi. MyBenchmark: generating databases for query workloads. *The VLDB Journal, 23(6), 2014.* 7

[16] N. Bruno and S. Chaudhuri. Flexible Database Generators. *In Proc. of 31st VLDB Conf.,2005, pgs. 1097-1107.* 6

[17] C. Binnig, D. Kossmann, E. Lo. Reverse Query Processing. *In Proc. of 23rd ICDE Conf.,2007, pgs. 506-515* 6

[18] Rakesh Agrawal, Ramakrishnan Srikant,Fast Algorithms for Mining Association Rules *Proceedings of the 20th VLDB Conference Santiago, Chile, 1994* 45