Duplication-Aware Synthetic Data Regeneration

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Technology

IN

Faculty of Engineering

BY

Tarun Kumar Patel

INDIAN INSTITUTE OF SCIENCE
भारतीय विज्ञान संस्थान

Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012 (INDIA)

July, 2021

Declaration of Originality

I, **Tarun Kumar Patel**, with SR No. **04-04-00-10-42-19-1-17265** hereby declare that the material presented in the thesis titled

Duplication-Aware Synthetic Data Regeneration

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

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

Date: 10-July-2021 Student Signature: Tarun Kumar Patel

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

all my family members and friends

Acknowledgements

I am incredibly thankful to my advisor, Prof. Jayant R. Haritsa, for giving me a chance to work in the Database Systems Lab under his guidance. I learned a lot from his way of thinking, approaching the problem, and dissecting things as far as possible to get better understandings. Also, being a part of the Database Systems Lab helped me improve some of my traits due to a healthy, supportive and knowledgeable environment. I am also very much thankful to Anupam for guiding me throughout the project completion, and his endless support and valuable suggestion helped me a lot. I am also very grateful that I got excellent lab mates, who were always there whenever I am in need. Finally, thanks to all my family and friends for always being with me.

Abstract

Generating synthetic databases for testing database applications, which can simulate the client's side data warehouse as close as desirable, is a common requirement from database vendors as it helps to replicate the query processing environment. Hydra[7] generates synthetic data that is volumetrically similar to the real database with respect to an apriori given query workload. That is, assuming a common choice of query execution plans at the client and vendor sites, the output row cardinalities of individual operators in these plans are very similar in the original and synthetic databases. However, there are several potential (synthetic) databases, all of which satisfy these volumetric constraints. This work tries to capture another statistical property in addition to volumetric similarity. That is, the techniques proposed here generates a synthetic database that not only satisfies volumetric constraints but also matches the per-attribute duplication frequency distribution, which can lead to better analysis database engine performance on client's environment.

In this thesis, the term duplication distribution is introduced to define the distribution of duplicate values in the data. We present *DupGen*, a dynamic data generator that incorporates duplication distribution of data with volumetric similarity on generated synthetic data. *DupGen* achieves the exact similar duplication distribution for attributes of base relation and with some error on filter predicates applied on the base relation and in addition to that also ensures volumetric similarity at both base and intermediate nodes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Testing Enterprise Database Systems, which requires (a) rectifying bugs that surface, and (b) pre-assessing the impact of planned engine upgrades, on customer deployments continues to be a challenging problem. This is attributed to the unavailability of the client's data at the vendor-site. Therefore, database vendors often resort to generating synthetic databases that mimic, for the intended purposes, the behavior of the client data processing environments.

1.1 Related Works and Limitations

Data generation are of primarily two types, workload independent and workload dependent data generation. We are dealing with the class of workload dependent data generation, which takes schema information and query workload from client to generate synthetic data that adheres some property of client's database. Over the last decade, several client-centric data generators have been proposed, like MyBenchmark [14], DataSynth[11] and Hydra [7]. These tools focus on generating synthetic data that exhibits *volumetrically similar* behavior to the original database on the customer query workload. That is, for a query, the output row cardinalities of individual operators in the corresponding query plans are almost identical in original and synthetic data. For example, consider the following toy database with the following relations:

Employee(*empNo*,salary,companyID)
Company(*companyID*,cname)

On this schema, a simple SQL query is shown below:

```
SELECT * FROM Employee E,Company C
WHERE C.companyID=E.companyID
```

On executing this query on the original database residing in one of the well known commercial database, the query plan produced is as shown in Figure 1.1. Notice that the plan shows the input/output row cardinality corresponding to each operator in the plan tree. For achieving volumetric similarity, these cardinalities should be close for original and synthetic databases.

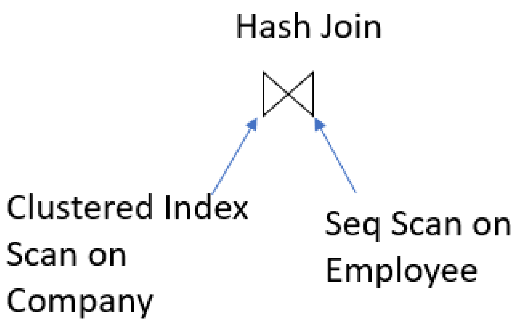


Figure 1.1: Query plan for above query

Limitation. While volumetric similarity captures an important data characteristic necessary for mimicking the client data processing environment, it lacks a very critical aspect of the data distribution and one kind of distribution is the *duplication distribution*. To illustrate this, we constructed two data sets D1 and D2, for our example schema, which differ only in the Employee.companyID column. Note that Employee.companyID is a foreign key column - that is, it takes values from the corresponding reference table Company. Therefore, values in Employee.companyID is a subset of Company.companyID. The difference in this column is specfied below:

D1: Employee.companyID has a uniform distribution over all the values present in Company.companyID.

D2: Employee.companyID has all identical values.

Both the datasets have 655 million and 82 million rows in Employee and Company table respectively. The example query, when run on these two data sets give identical physical query plans with same row cardinalities. However, in spite of satisfying volumetric similarity, they have significantly different running times, as shown in Table 1.1. As we can see for D1 the time was 18 minutes which increased to 28 minutes for D2. Note that these times were computed on identical hardware, database platform (a popular commercial engine) and system configuration. As per our judgement, the primary source of the time difference is the spilling behavior that happens while computing the hash table, which is used for performing join operation. The spilling mechanism depends heavily on the duplication distribution of the data. Hence, our focus is on capturing the duplication distribution in the synthetic data.

Distribution Type	**Running Time**
D1	18 min
D2	28 min

Table 1.1: Query Execution Time

1.2 Our Contribution

Motivated from the above observation, we propose ***DupGen***, a new data regenerator that aims at mimicking the duplication behaviour in the synthetic database. Our primary contributions are as follows:

Duplication Distribution Expression : A duplication distribution, for a set of attributes $\mathbb{A}$, is expressed as a 2-D vector that stores the information of how many value combinations corresponding to $\mathbb{A}$ occur with a certain frequency. For example, Given array a =[4,2,3,1,4], the duplication distribution will be $\{(2, 1), (1, 3)\}$ as only 4 is repeating two times and 1,2 and 3 (three elements) occurs only single time. Similarly, (d, f) vectors for D1 is {(8, 82 million)} as all 82 million values were uniformly distributed and for D2 is {(655 million, 1)}, since all values are identical.

Note that duplication distribution already encapsulates the total row-cardinality information. Therefore, ensuring matching duplication distribution implies volumetric similarity as well.

Further, we also propose a measurement, called *duplication-error* (S_e), for comparing the duplication distribution of two different data sets for a set of attributes.

Duplication Distribution Computation: All the database platforms give the information of input/output row cardinality for each operator in the query execution plan. However, the duplication distribution information is not provided explicitly. Therefore, to compute it, we use either (a) a non-invasive offline algorithm, where for each operator in the plan tree, a corresponding SQL query is constructed that returns the required duplication distribution at that operator, or (b) an invasive online algorithm, which computes the duplication distribution for each operator during the query execution itself. We have implemented this approach for open-source PostgreSQLv9.6 [6] database engine. Further, to make the computation efficient, we also have an approximation variant of the algorithm that gives approximate duplication distribution without having serious overheads on the query execution time.

Duplication Distribution Mimicking: Given the duplication distribution at each operator in the plan tree, *DupGen* aims to generate a synthetic database that also exhibits similar duplication behavior. Our current work is limited to ensuring duplication behavior at the various level of nodes of query plan tree. For getting duplication distribution, we could have used metadata, but since we need duplication distribution at all level of query plan tree and also ensuring duplication distribution at base nodes doesn't ensure duplication distribution at intermediate nodes. Also, metadata information doesn't provide the exact duplication distribution in many cases.

1.3 Organisation

The remainder of this thesis is organized as follows: An overview on duplication distribution and the distance measure is discussed in chapter 2. Further chapter 3, discussed the algorithm and implementation details of duplication distribution computation from the client site. The problem statement is further defined in chapter 4. Following chapter 5 gives the details of the data generation algorithm that aims towards mimicking duplication behavior. Next, chapter 7 discussed the experimental evaluation, and finally, we conclude with some brief ideas of future work.

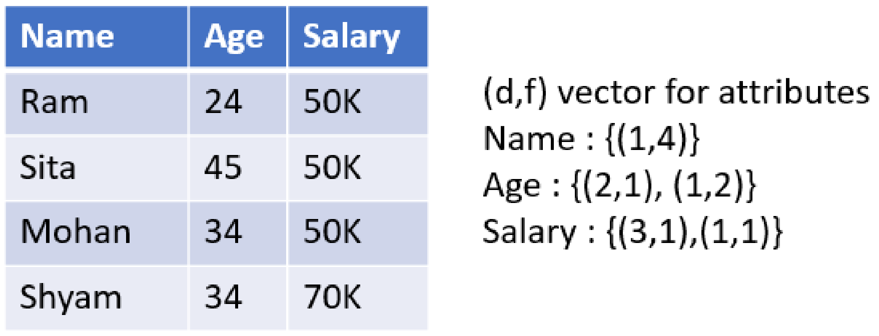
Chapter 2

Expressing Duplication Distribution

We first formally describe a duplication distribution with respect to a set of attributes. Further, we also give a measure of computing distance between two duplication distributions.

2.1 Duplication Distribution

A duplication distribution is expressed using a 2D vector $\{(d, f)\}$, where d represents the number of duplicates, and f denotes the number of attribute values having d duplicates. The duplication distribution in this paper is used interchangeably in terms of duplication vectors. For example see in figure 2.1, where duplication vectors for each attribute is given like $\{(d_1, f_1), (d_2, f_2), ...(d_n, f_n)\}$



Name	Age	Salary
Ram	24	50K
Sita	45	50K
Mohan	34	50K
Shyam	34	70K

(d,f) vector for attributes
Name : {(1,4)}
Age : {(2,1), (1,2)}
Salary : {(3,1),(1,1)}

Figure 2.1: Duplication distribution on all attributes of given table

Duplication vectors contains cardinality information as well, and can be expressed as

$$\|v\| = \sum_{i=1}^{\beta(v)} (d_i \times f_i)$$

where $v = \{(d_1, f_1), (d_2, f_2), ...(d_n, f_n)\}$ and $\beta(v)$ is number of (d, f) elements in v.

Summation of two duplication vector v_1 and v_2 can be defined as $v = v_1 + v_2$, such that for all common d values, f would be $f_{v1} + f_{v2}$, otherwise all (d, f) from both v_1 and v_2 are appended to v.

2.2 Bound on size of duplication vectors

For any attribute a in set of attributes $\mathbb{A}$ of Relation $\mathcal{R}$,let duplication vector on attribute a be v_a, then $\|v_a\| = |\mathcal{R}|$. The length of v_a is dependent on the size of $\mathcal{R}$. The number of entries in the distribution is equal to the number of distinct duplication frequencies for values occurring in attribute a. We express this number as β for a. It is easy to see that β is maximum when the duplication frequency distribution is of the type: $\{(1, 1), (2, 1), (3, 1), ..., (a_\beta, 1)\}$. This gives us the following condition:

$$1(1) + 2(1) + 3(1) + ..., \beta(1) = |\mathcal{R}| \tag{2.1}$$

This gives us $\beta = O(\sqrt{|\mathcal{R}|})$. Hence even for a trillion rows relation, the duplication frequency distribution of each attribute can be captured using few MBs of data in the worst case.

2.3 Distance between duplication vectors

For comparing duplication vectors, we need a distance metric, termed duplication distance. Given two duplication vectors v_1 and v_2, the duplication distance between v_1 and v_2 can be defined as

$$S_d(v_1, v_2) = D(a_{v1}, a_{v2}) = \sum_{i=1}^{len(a_{v1})} |a_{v1}[i] - a_{v2}[i]|$$

where v_1 and v_2 are transformed into array of integers a_{v1} and a_{v2}, such that both a_{v1} and a_{v2} are in sorted (descending) order and have same number of elements and $D(a_{v1}, a_{v2})$, gives summation of absolute distance between each corresponding element of a_{v1} and a_{v2}. Using sorted order gives the minimum distance between the duplication vectors proved in Lemma 1.

Transforming duplication vectors For distance calculation between two vectors, the basic requirement is to have same number of elements. So, to achieve this, first both v_1 and v_2 are transformed into array of integers, such that for each (d, f) in duplication vector v, d is added to array f times. Thus number of element in transformed duplication vector v is

$$|a_v| = \sum_{i=1}^{\beta(v)} f_i$$

, where $\beta(v)$ is size of duplication vector v

Since, for vectors v_1 and v_2, summation of f can be different, thus to have same number of elements extra $0s$ are appended to the array having less number of elements. All elements in transformed array are sorted in descending order.Example of duplication distance calculation is shown in figure 2.2. Duplication distance is further used by duplication error defined in Experiment section, which shows how much closer is v_1 to v_2, by normalizing duplication distance such that it ranges from 0 to 1.

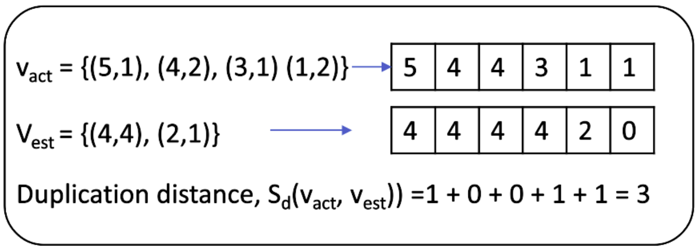

v_{act} = {(5,1), (4,2), (3,1) (1,2)}
5 4 4 3 1 1
V_{est} = {(4,4), (2,1)}
4 4 4 4 2 0
Duplication distance, $S_d(v_{act}, v_{est})$) =1 + 0 + 0 + 1 + 1 = 3

Figure 2.2: Example of duplication distance calculation

Lemma 2.1 *Given two array A and B of same size and distance between A and B is given by d(A,B) = $\sum_{i=1}^{\beta(A)} |A[i] - B[i]|$, where size of A and B is $\beta(A)(= \beta(B))$. Then sorted order of both A and B will give minimum distance than any other order.*

Proof: Let A be $[a_1, a_2, ...a_n]$ and B be $[b_1, b_2, ...b_n]$, sorted on decreasing order such that $a_1 \geq a_2 \geq ... \geq a_n$, similarly holding for values in B.

Now WLOG, select 2 random elements from both A and B, for swapping, let from A, p and q are picked and from B, r and s are picked, such that $p \geq q$ and $r \geq s$. Considering all four possibility of

1. $p \geq r$ and $q \geq s$
2. $p < r$ and $q > s$
3. $p > r$ and $q < s$
4. $p \leq r$ and $q \leq s$

Clearly case 1 is symmetric to case 4, similarly case 2 is symmetric to case 3. Now considering first case 1, here swapping gives same minimum distance as it is for sorted order, since $(p - r)$ + $(q - s)$ = $(p - s)$ + $(q - r)$. Now in case 2, $(p - s) \geq (p - r)$ and $(q - r) \geq (q - s)$, using $p \leq q$ and $r \leq s$, Therefore, $(p - r)$ + $(q - s)$ $\leq$ $(p - s)$ + $(q - r)$.Similarly, this holds for multiple swap as well. Thus, there are no possibilities of any other order to give minimum distance than sorted order.

□

2.4 Duplication Error

We have defined duplication error as an error metric to compare the duplication vectors normalizing the duplication distance(S_d) between the vectors by dividing it by twice the number of tuples(represented by a duplication vector). Let v_{actual} be actual duplication vector for some relation $\mathcal{R}$ and $v_{estimated}$ be estimated duplication vector for $\mathcal{R}$, then

$$S_e(v_{estimated}) = \frac{S_d(v_{actual}, v_{estimated})}{2 \times ||v_{actual}||}$$

The reason of dividing from 2, is to have in range between 0 and 1.

Metric bound on duplication error Consider the relation $\mathcal{R}$. We can define duplication vector for any attribute a in $\mathcal{R}$ in exactly two opposite distribution like $(|\mathcal{R}|, 1)$ and $\{(1, |\mathcal{R}|)\}$, where $|R|$ is the number of tuples in R. $\{(|\mathcal{R}|, 1)\}$, represents the case having all attributes values same. In contrast, $\{(1, |\mathcal{R}|)\}$, represents the case where all attribute values are distinct(which indicates that they are an opposite case of duplication distribution). As shown in figure 2.3, the duplication distance between these two duplication vector would be $2|R|$ - 2 and duplication error would be (1 - $\frac{1}{|R|}$), which shows duplication error would always be less than 1. Having twice in the denominator is a construct that helps to bound duplication error to 1 instead of 2.Similarly, accuracy for estimated duplication vector can be defined as 1 - S_e. Also having accuracy near to 1 means the distribution assigned to synthetic data is very close to actual and similarly accuracy is near to 0, means generated synthetic data have opposite distribution from actual(i.e for the actual case $\{(|\mathcal{R}|, 1)\}$, synthetic data distribution with duplication vector $\{(1, |\mathcal{R}|)\}$ will be the opposite distribution for the actual case).

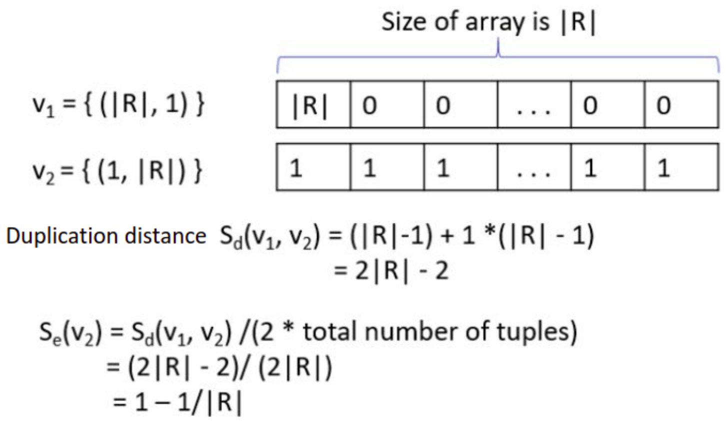


Figure 2.3: Bound on duplication error

2.5 Subset Property of duplication vectors

Given two duplication vectors v_1 and v_2, v_1 can be subset of v_2, iff each (d_i, f_i) of v_1 is present or can be generated using duplication vector v_2. For example

$$\text{Given } v_1 = \{(2,3)\}$$
$$\text{and } v_2 = \{(4,2),(2,1)\}$$

So, there are three values that have two duplicates in v_1 and in the case of v_2, there are also three values, but two of them have two duplicates and one of them is having four duplicates. Since, we can have 3 values from v_2, which have two duplicates, thus, v_1 is subset of v_2. This property is further used to formulate linear constraints for intermediate nodes of query plan.

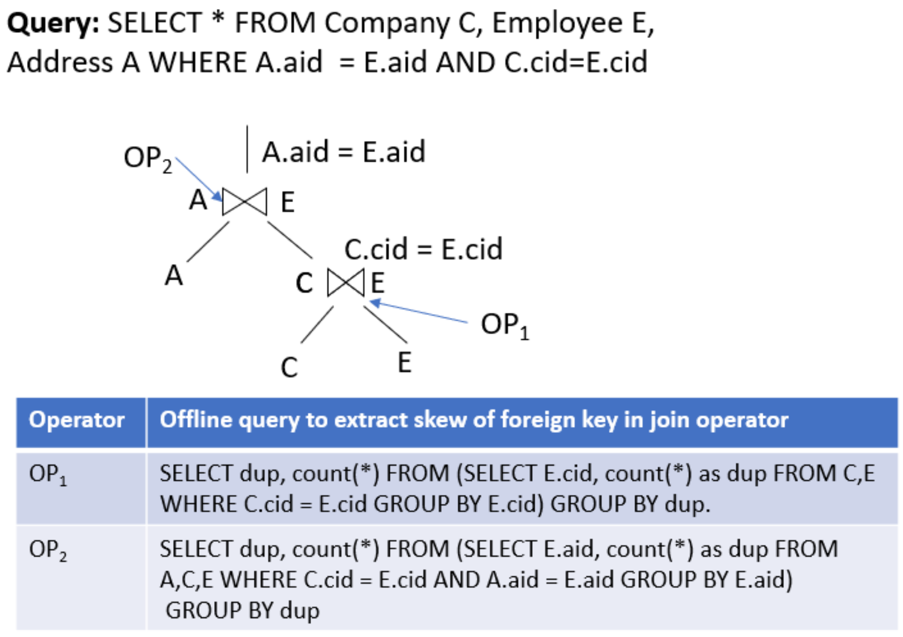
Chapter 3

Compute duplication distribution at client side

Input and output row cardinalities for all operators of query plan are directly available from the database engine, but duplication distribution information is needed to compute explicitly. We proposed three approaches to compute duplication distribution for each operator in query in the query workload.

3.1 Offline Approach

Using the query plan of each query in the query workload to get duplication distribution at all operators of queries. Getting each operator's sub-tree query got from the query plan. Applying GROUP BY with count(*) on a attribute of a relation gives frequency count of each distinct element in the relation and applying GROUP BY on the top of frequency count result of each element with count(*) gives duplication distribution for that attribute. Thus an offline version of decorrelated query can be formulated for each operator as shown in the figure 3.1.



Query: SELECT * FROM Company C, Employee E, Address A WHERE A.aid = E.aid AND C.cid=E.cid

Operator	Offline query to extract skew of foreign key in join operator
OP$_1$	SELECT dup, count(*) FROM (SELECT E.cid, count(*) as dup FROM C,E WHERE C.cid = E.cid GROUP BY E.cid) GROUP BY dup.
OP$_2$	SELECT dup, count(*) FROM (SELECT E.aid, count(*) as dup FROM A,C,E WHERE C.cid = E.cid AND A.aid = E.aid GROUP BY E.aid) GROUP BY dup

Figure 3.1: Duplication distribution using *Group By*

3.2 Online Approach

In online approach, we monitor the output of each operator while execution of the query and compute the exact duplication distribution for targeted attributes. We added a frequency counter(FC_1) in the output of each operator to have the frequency count of each element. Once the FC_1 is fully updated(i.e all elements have an entry in FC_1) then, another frequency counter(FC_2) on the top of FC_1 is applied for having duplication distribution of all attributes present in operator's output.

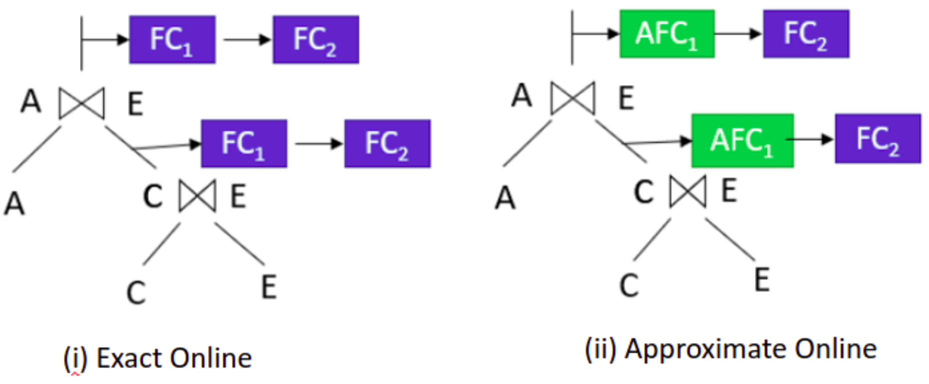


Figure 3.2: Online duplication Computation at operator's output

3.2.1 Exact Online Approach

In exact approach as shown in Figure 3.2(i), we calculate exact duplication distribution, using both frequency counters FC_1 and FC_2, where FC_1 keeps track of exact frequency count of elements and FC_2 runs on the result of FC_1, to get the duplication counts. Since there can be large amount of data, varying from very high to low frequency. Keeping track of low frequency elements can result in more space overhead. So to reduce the overheads of small frequency elements, approximate online approach is introduced.

3.2.2 Approximate Online Approach

In approximate online approach as shown in Figure 3.2(ii), instead of using frequency counter FC_1, we have used Approximate Frequency Counter (AFC). Approximate frequency counter uses Lossy Counting [9] as an approximation algorithm, to keep tracks of most frequent items only. Thus using Approximate frequency counter results in less number of elements after first frequency counting which reduces space overhead and further second frequency counter which runs over the result of approximate frequency counter, which is quite less in size compare to the result of frequency counter, thus also results in reducing time overheads. Since second frequency counter runs on approximate frequency count, it will give approximated duplication distribution. The elements which does not come in the result of approximate frequency counter end up as they are repeating only single time, just to have total number of tuples in database equals to number of tuples by duplication distribution.

Chapter 4

Problem Framework

We describe the problem framework for a single relation instance. Each relation in the database can be dealt with in the same manner. The duplication frequency distribution is matched at an attribute level so there is no dependency across two attributes within or across relations. Further, for simplicity, we are currently dealing only with mimicking duplication distribution of foreign keys of relations, which could further help us to handle joins between relation in future.

4.1 Problem Statement

Given set of cardinality constraints and duplication distribution($v_a = \{(d_i, f_i)\} | i \in [\beta(a)]$ for attribute a) of attributes(in our case only foreign keys are considered) in base relation($\mathcal{R}$), generate synthetic relation($\mathcal{R}'$), that have volumetric and duplication distribution, such that it follows similar duplication distribution for various nodes of query plan tree.

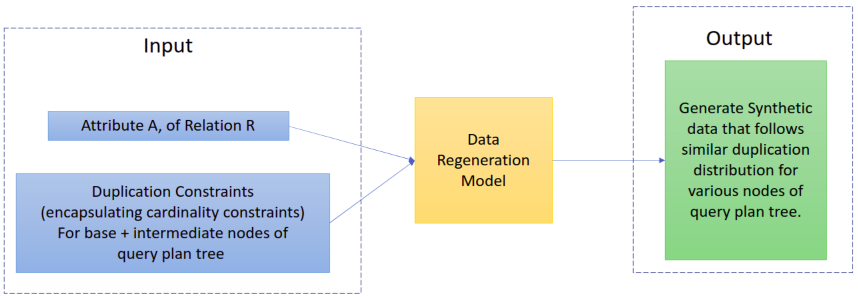

Input
Attribute A, of Relation R
Duplication Constraints (encapsulating cardinality constraints) For base + intermediate nodes of query plan tree
Data Regeneration Model
Output
Generate Synthetic data that follows similar duplication distribution for various nodes of query plan tree.

Figure 4.1: Problem Framework

Chapter 5

Data Generation at Vendor site

After computing duplication distribution at the client side, data generation can be done with the help of schema, query plan trees on client's query workload and duplication distribution data. This section deals with whole data generation process which involves LP formulation, distribution of duplication information and database summary generation, which can be used further to generate tuples. We termed our data generation model as *DupGen*.

5.1 Overview of DupGen

DupGen takes schema, query plans associated with duplication distribution of attributes annotated with each node of query plan and also duplication distribution of attributes of the base relations as input.For now only, foreign key attributes are considered for duplication distribution mimicking. *DupGen* aims to ensure volumetric similarity at all the intermediate nodes of query plan tree and in addition to that it matches the duplication distribution of base relations as well. And in the case of duplication distribution after filters applied on base relation, it tries to mimic duplication distribution at filter nodes as close as possible to the original duplication distribution.

The Figure 5.1 shows the end to end pipeline of the *DupGen*. *DupGen* mostly works on top of Hydra [7], with some additional features added in it to ensure mimicking of duplication distribution. The green shaded blocks are the modules where additional work is done which is to be discussed below. In LP formulation module, some duplication distribution related linear constraints are added discussed in section 6. The duplication distribution block is new addition to the Hydra pipeline which distributes duplication vector after the LP solving stage using duplication distribution algorithm discussed in section 7. And the data generation blocks deals with view summary generation and tuple generation discussed in section 8.

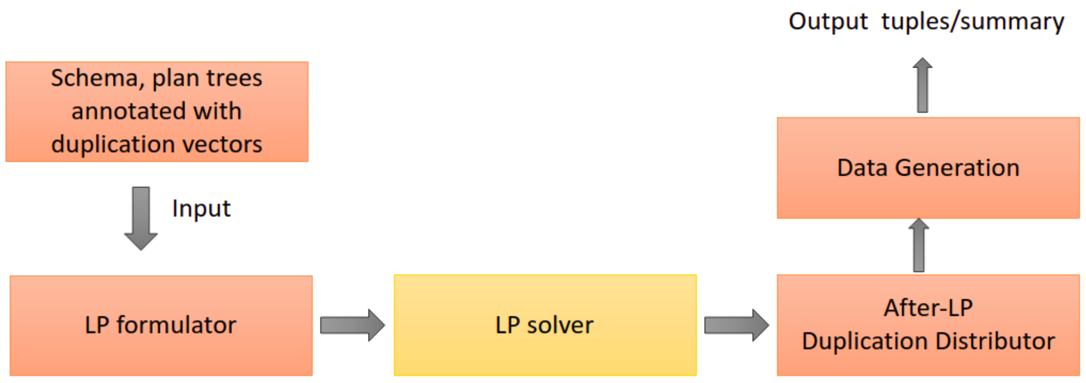

Output tuples/summary
Schema, plan trees annotated with duplication vectors
Input
Data Generation
LP formulator
LP solver
After-LP Duplication Distributor

Figure 5.1: DupGen Pipeline

Currently, we focus on ensuring the duplication distribution of the foreign key columns as they are integral for join processing, which can be used in future work of handling duplication distribution with joins. Based on the common assumption of considering queries with only primary-key foreign-key joins, the constraint at each intermediate node of the query plan can be treated as a filter condition on a specific view. Using this observation, Hydra[7] generates each relation independently by first constructing a view corresponding to each relation. The view comprises the relation's non-key attributes and is augmented with the borrowed non-key attributes of the relations. It depends on referential constraints (both directly or transitively).Like Hydra, *DupGen* also works on view level, which in the future will be helpful to handle joins with duplication distribution. Also, solving the problem at view level solves the problem at relation level inherently; that is, our actual problem statement. Consider an example, given two relation $COMPANY(\underline{companyID}, stocks)$ and $EMPLOYEE(\underline{empId}, salary, companyID)$, in which $EMPLOYEE$ references from $COMPANY$ using $companyID$ then COMPANY_VIEW and EMPLOYEE_VIEW and will be generated, and $stocks$ is borrowed attribute in EMPLOYEE_VIEW from $COMPANY$ associated with foreign key $companyID$.

COMPANY_VIEW(stocks)
EMPLOYEE_VIEW(salary,stocks)

Since view comprises of non-key attributes only, therefore both $COMPANY\_VIEW$ and $EMPLOYEE\_VIEW$ have only non-key attributes. Further given below are some example cardinality constraints applied on $COMPANY$ and $EMPLOYEE$ relation:

$$| \sigma_{salary>50K}(EMPLOYEE) | = 16520;$$
$$| \sigma_{salary>50K \wedge stocks>2000}(COMPANY \bowtie EMPLOYEE) | = 8000$$

Above cardinality constraints can be expressed in form of constraints applied on views such that it involves borrowed non-key attributes, as :

$$| \sigma_{salary>50K}(EMPLOYEE\_VIEW) |= 16520$$
$$| \sigma_{salary>50K \wedge stocks>2000}(EMPLOYEE\_VIEW) |= 8000$$

Expressing cardinality constraints in terms of views will also helps in join modelling in future.

5.2 LP formulation

The cardinality constraints and duplication distribution associated with each cardinality constraint and base relation are formulated as linear constraints. LP formulation requires partitioning of view into a set of regions described in HYDRA [7], such that for each *CCs*, there are some subset of regions that satisfies the tuple count.And the summation of tuples from each partitioned regions matches with relation cardinality. Also Hydra [7] uses sub-view optimization, to reduce the number of variables in LP. Sub-views in a Hydra are subset of views, constructed using some subset of the attributes from view.This is achieved as follows: Construct a "view-graph" by first creating a node for each attribute, and then inserting an edge between a pair of nodes if the corresponding attributes appear together in one or more CCs. Further, additional edges are added(if required) to make the view-graph to be chordal, a property required to ensure acylicity in the subsequent processing .Sub-views are identified as the maximal cliques in the view-graph. Similar to Hydra, *DupGen* also works on sub-views with additional constraint that all borrowed attributes of each foreign key must reside in any one of the sub-view.

5.2.1 Cardinality Constraints

Consider above example of EMPLOYEE and COMPANY relations, suppose there are two queries in workload, first one aims to get employees having salary less than 90K and working in companies having stocks less than 4000, and other aim to get employees having salary more than 50K and working in the companies having stocks more than 2000. With the help of CCs and after region partitioning, the Employee_view will be as shown in figure 5.2.

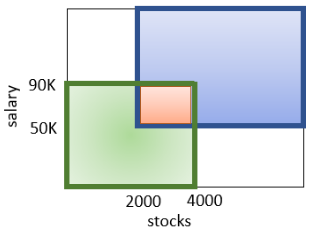


Figure 5.2: Region Partitioning in Employee View

And the LPs can be formed by combining regions such that they satisfy all CCs overlapping with the view. Additionally, satisfying the summation of all region cardinality equals to relation cardinality.

These cardinality constraints were also added in Hydra[7]

5.2.2 Duplication Constraints

Seeing query performance for different duplication distribution of foreign key, our current focus is to formulate duplications for only foreign key attributes. Still, our algorithm is general can be extended to work for all attributes. To divide duplication distribution of the foreign key into partitioned regions of view, the view i s divided into intervals of borrowed attributes corresponding to the foreign key, the intervals in view are referred as interval regions as show in figure 5.3(i). Each interval regions will have some duplication distribution , and summation of duplication distribution of all interval regions will be equal to duplication distribution of foreign key at base relation. As in figure 5.3(i), imaginary dimension for foreign key is added and its values are distributed in interval regions of borrowed attributes. If there are m borrowed attributes and a_i intervals in i^{th} borrowed attribute then number of interval regions in the view will be $\prod_{i=1}^{m} a_i$.Intervalisation leads to generation of new regions referred as intervalised regions.

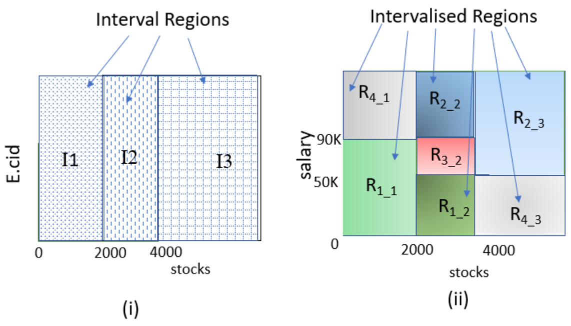


Figure 5.3: (i) Interval Regions (ii) Intervalised Regions

These interval regions and intervalised regions are generated for each foreign key of the view separately. Since a view can have multiple sub-views, and each foreign key will have all its borrowed attributes on one of the sub-view. So, these interval regions and intervalised regions will appear only in those sub-views that have borrowed attributes of foreign keys.

Constraints generated using duplication vector of base relation Let $I = \{I_1, I_2, ..., I_m\}$ be the set of interval regions and duplication vector on base relation be $v = \{(d_1, f_1), (d_2, f_2), ..., (d_n, f_n)\}$. Then, formulating duplication vector into linear constraints would be like dividing some integer fraction(x_{ji}) of frequency f_i for d_i in $\{(d_i, fi)\}$ to each interval region, such that summation of all fraction divided among the interval regions sums up equal to f_i:

$$\text{For each } I_j \in I$$
$$|I_j| = \textstyle\sum_{i=1}^{n} x_{ji} * d_i$$

$$\text{For each } f_i \in v$$
$$|f_i| = \textstyle\sum_{j=1}^{m} x_{ji}$$

Intervalised regions belonging to same interval should have summation of region cardinality equals to interval region cardinality.

$$\text{For each } I_j \in I$$
$$|I_j| = \textstyle\sum_{r \in I_j} r$$

And, regions divided into intervalised region should have summation of tuples equals to the partitioned region cardinality(region before intervalisation).

$$\text{For each partitioned region, } R_i \in \text{View } V$$
$$|R_i| = \textstyle\sum_{r \in R_i} r$$
$$\text{where r refers as intervalised region}$$

Constraints generated using duplication vector of cardinality constraints All cardinality constraints associated with a foreign key have a corresponding duplication vector of foreign key. Since, each cardinality constraints(CCs) overlaps some of the intervalised region, we can add some linear constraints for each CC's duplication vector. Let duplication vector some CC be $ccDup = \{(D_1, F_1), (D_2, F_2), ..., (D_m, F_m)\}$ and duplication vector of foreign key on base relation is $baseDup = \{(d_1, f_1), (d_2, f_2), ..., (d_n, f_n)\}$.

Algorithm 1, discusses the set of equations that can be added for duplication vector for each CC. The main idea behind this set of equations is for any (D_k, F_k) value in *ccDup*, the overlapping interval regions with CCs, must have atleast D_k duplicates repeating at least F_k

times. These equations are mainly added on the basis of subset property of duplication vectors, where CC duplication vector are subset of base duplication vector. So the duplication vectors of CC should be a subset of the union of all duplication vectors that each interval region will have.

Also, all the above variables follow non-negative constraints. Finally, all the linear constraints are passed to SMT solver for the solution.

Algorithm 1: Generating LP constraints with duplication vectors of CCs

```
Result: LP equations for duplication vectors in CCs
// List of equations
$equations \leftarrow []$;
for $(D_k, F_k) \leftarrow ccDup$ do
    $sumFx = 0$;
    for $(d_i, f_i) \leftarrow baseDup$ do
        if $d_i < D_k$ then
            continue;
        end
        for $I_j \in I$ that overlaps with CCs do
            $sumFx \leftarrow sumFx + x_{ji}$;
        end
    end
    $equations.add(sumFx \geq F_k)$;
end
```

5.3 After-LP duplication distribution

After completing the LP solving stage, we have the count of tuples for each partitioned region in the view. And, for each foreign key, we have interval regions and intervalised regions tuple count, and in addition to that, we have duplication distribution information for each interval. In this section, the duplication distribution of each interval region heuristically divided among the intervalised regions associated with that interval region. And at the end of the algorithm, each intervalised region will be associated with some duplication vector. Algorithm 2,**??** gives the After-LP duplication distribution algorithm(can also be termed as After-LP distribution algorithm)

Input to After-LP distribution algorithm The duplication distribution algorithm runs for a foreign key and tries to mimic its duplication distribution. The algorithm needs the interval regions and intervalised regions associated with the foreign key, map of CCs associated with intervalised region($CCToRegionMap$), duplication vector associated with each interval region($intervalRegionToDFvector$), and duplication vector associated with CC *conditionToDFVector*, number of total tuples in the view($viewTupleCount$). Before getting deep into the algorithm and some submodules of the algorithm are discussed below. Consider all d_i from the duplication vector as d values and similarly all f_i values as f values.

Mapping intervalised regions and cardinality constraints Each interval region overlaps with some set of CCs. Based on the overlapping nature of CCs inside the interval region with intervalised regions, each intervalised region is mapped with a set of CCs that overlap with it. And also, inside each interval, the relation between each CC is also constructed based on the overlapping nature of CCs with each other. CCs can overlap with each other in two ways, either partially or fully subsuming one another a shown in figure 5.4. As can be seen from figure 5.4 intervalised region R1 maps to CC1, R2 maps to both CC1 and CC2 and R3 maps to CC2 in partially overlapping case. Both the overlapping nature of CCs will be utilised to distribute same foreign keys to different intervalised regions(but in same interval region), by distributing some number of duplicates to each intervalised region.

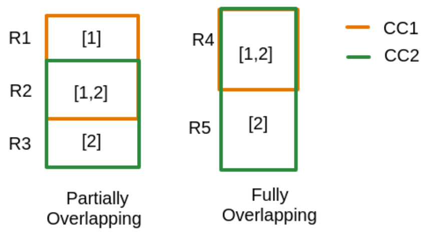


Figure 5.4: Overlapping nature of CCs

Greedy approach of choosing intervalised region The After-LP distribution algorithm proceeds with selecting the one of the interval regions among all interval regions, based on the maximum d value in the duplication vector of that interval region. Once the interval region is selected, then the next big thing is to distribute that (d, f)(coming from maximum d value and f corresponding to it) among the intervalised regions in the selected interval region. Since we have a map between intervalised regions and the overlapping regions, we use a greedy approach to distribute foreign keys among intervalised regions. The algorithm always selects the intervalised regions with a maximum number of overlapping CCs to distribute foreign keys. Once all foreign key values are assigned in that intervalised region, the next intervalised region is selected with the maximum CCs overlapping. This process goes on till all foreign key values are assigned to each intervalised region.

findAppropriateDF Once the algorithm knows, that some intervalised region is having maximum number of overlapping CCs, then all CCs are passed to $findAppropriateDF$. Since all CCs are associated with duplication vector of foreign key attribute, $findAppropriateDF$ find the appropriate (d, f) which can be deducted from all overlapping CCs. By appropriate (d, f), means the maximum d value(or duplicate count) that can exist on all CCs given as an input; and maximum f value for maximum d, that is available from all CCs. The appropriate (d, f) in algorithm is termed as $(appD, appF)$. Also after one more constraint is there for $(appD, appF)$, that it should be subset of $(dMax, fMax)$ (i.e the maximum (d, f) foreign key values that can be distributed in the current iteration of selected interval).By $(appD, appF)$ as a subset of $(dMax, fMax)$ means, $(appD < dMax)$ and $(appF < fMax)$.
Also, if no such $(appD, appF)$ found, then it will return -1. Scenarios like $findAppropriateDF$ returning -1 can happen since we are using LP solver and all require constraints are not added, ensuring the exact duplication distribution. So, LP solver can give the wrong duplication vector to the interval region compared to the original database's duplication vector. Thus in contrast, the distribution of the wrong duplication vector into intervalised regions leads to $findAppropriateDF$ to return -1.

Consider an example for finding appropriate (d, f) value.let there are two duplication vectors v1 and v2 and $(dMax, fMax)$ be (5,2). As a heuristic approach, the algorithm find interval region having maximum d value($dMax$) and associated f value($fMax$) in duplication vector among all interval regions and tries to distribute $dMax$ duplicates of $appF$ foreign keys.

$$\text{v1} = (4,2)\ (2,1)$$
$$\text{v2} = (3,4),(2,4)$$

Then ($appD$, $appF$) will be (3,2), since the maximum number of duplicates available on both duplication vector v1 and v2 is three and for and there are only two values in duplication vector v1 that are repeating more than(or equal to) three times, compare to duplication vector v2, which have four different values repeating three times. Since two different values repeating three times is common in both duplication vectors v1 and v2, therefore ($appD$, $appF$) = (3,2).

Algorithm 2: distributeDFvalues

```
   Result: Each intervalised region with some duplication vector
   Input: intervalRegions,intervalisedRegions,CCToRegionMap, CCToDFvector,
          intervalRegionToDFvector, viewTupleCount
 1 while intervalRegions.isNotEmpty() do
 2     // find interval region having maximum d value and associated f value in
         duplication vector
 3     currentInterval, dMax, fMax = maxDIntervalRetriever(intervalRegion)
 4     overlappingIntervalisedRegions =
         intervalRegionToIntervalisedRegionsMap.get(currentInterval)
 5     IR =
         findIntervalisedRegionWithMostOverlappingCCs(overlappingIntervalisedRegions)
 6     appD, appF = findAppropriateDF(CCToDFvector,IR, dMax, fMax)
 7     if appD == −1 then
 8         // No appropriate d value is left to distribute
 9         //(i.e dMax less than all d values available in overlapping CCs duplication
             vector in that intervalRegion
10         distributeToFutureFKVal(futurePKValDistribution, currentInterval,
             fkValBoundaryList);
11         intervalRegions.remove(currentInterval);
12     else
13         // distribute fMax fk values
14         distributeToVarFKValDistribution(varFKValDistribution,
             overlappingIntervalisedRegions, CCToDFvector, dMax, appD, appF,
             pkValBoundaryList, futurePKValDistribution, groupCCsPerInterval );
15         // update duplication vector of currentInterval by removing/updating
             (dMax,fMax)
16         updateDuplicationVector(intervalRegionToDFvector, currentInterval,
             toBeUsedF, dMax)
17     end
18     if intervalRegionToDFvector.get(currentInterval).isEmpty() then
19         intervalRegions.remove(currentInterval);
20     end
21 end
22 // distributes tuples inside each intervalRegion of futureFKValDistribution to left over
     tuples of intervalised region associated with that intervalRegion
23 distributeFutureFKValToLeftOver(futurePKValDistribution);
```

distributeToVarFKValDistribution Once $findAppropriateDF$ gives $appD$, and $appF$ and $appD$ is not equal to -1, then this function is called to distribute foreign key values among the intervalised region inside the corresponding interval region. The distribution of foreign key values can be understood as two cases.

Case 1 : $(appD < dMax)$ In this case, we know we have to to distribute foreign keys for $(appD, appF)$; that $appF$ number of foreign key values each repeating $appD$ times, should be distributed to the current intervalised region from which $(appD, appF)$ originated(i.e intervalised region with most overlapping CCs). The algorithm uses $fkValBoundaryList$ to generate foreign key values. For $(appD, appF)$, $appF$ foreign key values are picked from $fkValBoundaryList$ for current interval region. Those $appF$ foreign key values are assigned to intervalised regions with the most overlapping CCs. Since we need to distribute $dMax$ duplicates of $appF$ foreign keys in the current interval for the next iteration of selecting the interval with a maximum d value. So, to distribute the remaining duplicates of $appF$ foreign keys, overlapping property of CCs will be used. The algorithm picks the intervalised region having the maximum number of CCs and also overlaps with the current intervalised region. From the newly picked intervalised region, the maximum d value is determined using the $findAppropriateDF$ for the duplication vectors of all overlapping CCs in the newly picked intervalised region. If d value equal to $dMax$ found then distribute $appF$ foreign keys repeating $(dMax - duplicatesDone)$ times where $duplicatesDone$ is the number of duplicates of $appF$ foreign keys are present in that interval region. Else the intervalised region will have maximum d value(d'), which is less than $dMax$, so distribute $appF$ foreign keys repeating $(d' - duplicatesDone)$ times; since still $dMax$ duplicates are not done, again next intervalised region is picked in saame way and this process goes, until all $dMax$ duplicates are done for $appF$ foreign keys. Also, to keep track of what (d, f) value is distributed from respective CCs, deduction in duplication vectors takes place simultaneously. For each intervalised region, the distribution of foreign keys leads to the deduction of $((d, f)$ values from overlapping CCs with it.

Case 2: $(appD == dMax)$ In this case distribute $appF$ foreign keys repeating $appD(ordMax)$ times to the selected intervalised region. Since all $dMax$ duplicates are assigned to the selected intervalised region, the algorithm can proceed with the next iteration of selecting interval with maximum d value. And also, duplication vectors of overlapping CCs with the current intervalised regions will deduct $(appD, appF)$ from itself.

Both the cases distribute foreign key values among intervalised regions uses $fkValBoundaryList$ as a source of foreign keys. The $fkValBoundaryList$ list maintains the maximum bound of foreign keys that can be present in the interval region of the foreign key. The utmost bound is given by the total number of tuples in the current view(or relation). For example: if there are four interval regions, and the number of total tuples in that view is 100, then $fkValBoundaryList$ will be [1, 101, 201, 301, 401]. And, let's say two foreign key values have been used from first interval region, then modified $fkValBoundaryList$ will be [3, 101, 201, 301, 401].

Example of foreign key values distribution

Consider the example shown in figure 5.5, it contains an arbitrary interval region with the duplication vector (10,3) (i.e $(dMax, fMax) = (10, 3)$). There are three intervalised regions and three CCs. Intervalised region $R3$ is the region with the most overlapping CCs. So, first, $findAppropriateDF$ will have a set of CCs as an input that overlaps with intervalised region R3. $findAppropriateDF$ will give $(appD, appF)$ as (3,2), since (3,2) is having maximum d value that can be deducted from all overlapping CCs. Since $appF$ is two, we need to distribute two unique foreign key values in the selected intervalised region repeating $appD$ times and in the interval region repeating $dMax$ times. So first, it will distribute $appF$ foreign keys repeating $appD$ times in the intervalised region $R3$. Then for the remaining $(dMax - appD)$ duplicates, other intervalised regions will be used. So for now, $R3$ has two foreign key values repeating three times. And, the number of duplicates for two foreign keys remaining to distribute is seven(coming from $(dMax - appD)$).

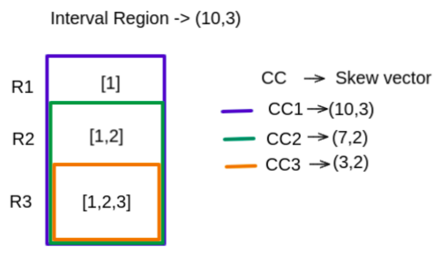


Figure 5.5: Arbitrary interval region with overlapping CCs

The algorithm looks for the next intervalised region that can be used to distribute the remaining duplicates. Since intervalised region, $R2$ have most overlapping CCs, and also it's some of the CCs overlaps with R3. The intervalised region $R2$ is passed to $findAppropriateDF$ to get the $(appD, appF)$ that can accommodate in R2. $findAppropriateDF$ returns (7,2) as $(appD, appF)$. Since already from CC1 and CC2, which overlaps with both $R1$ and $R2$ have been used to distribute (3,2) to intervalised region $R3$. Therefore $(appD - duplicationDone)$ (i.e $7 - 3 = 4$) will only distributed to intervalised region $R2$. So $R2$ will have same two foreign key values used in $R3$, with 4 duplicates of each foreign key. Still only 7 duplicates are done, and target is to complete 10 duplicates, next intervalised region will be picked that is $R1$. Passing

CC1 of $R1$ to $findAppropriateDF$ will give (10,2) as $(appD, appF)$. And since already (7,2) is distributed in CC1, the remaining 3 duplicates of each of the two foreign keys are added to the intervalised region $R1$. Once $dMax$ duplication of $appD$ is achieved, we know how much from each CCs needs to be deducted. For example, from CC3 (3,2), from CC2 (7,2), from CC1 (10,2) is to be deducted since foreign keys are distributed in this distribution only in these CCs.

All ten duplicates of two foreign keys are distributed in the current interval region, so the interval region still has (10,1) to distribute, which will be distributed in the next iteration as (10,1) will be left in CC1 only. So in the next iteration, $R1$ will have one more foreign key having ten duplicates since only $R1$ is left overlapping with CC1.

distributeToFutureFKVal If *appD* equals -1 or there is no more option left to distribute foreign key values among intervalised regions, it can reduce error. Therefore add all foreign key values, and their duplicate count is added to $futureFKValDistribution$. The *distribute-FutureFKValToLeftOver* function uses $futureFKValDistribution$ to distribute foreign keys among the intervalised region having less number of foreign keys then its tuple count. There is no specific rule to distribute tuples; just for each intervalised region number of the foreign key inside that intervalised region should be equal to the tuple count of that region. The foreign key values added because of $futureFKValDistribution$ results in error compared with the actual duplication vector of cardinality constraints.

Algorithm 3: Duplication distribution algorithm for each foreign key associated with view

```
Result: Each intervalised region with some duplication vector
Input: intervalRegions,intervalisedRegions,CCToRegionMap, CCToDFvector,
       intervalRegionToDFvector, viewTupleCount
1  fkValBoundaryList =[1]
2  temp = 0;
3  // Setting fk values boundary for each interval region
4  for $interval \leftarrow intervalRegions$ do
5      fkValBoundaryList.append(viewTupleCount + temp);
6      temp += viewTupleCount;
7  end
8  // Maps each intervalRegion region with fk values and its duplicate count
     futureFKValDistribution = Map()
9  // Maps each intervalised region with fk values and its duplicate count
10 varFKValDistribution = Map()
11 distributeDFvalues(varFKValDistribution,
     fkValBoundaryList,intervalRegions,intervalisedRegions,CCToRegionMap,
     CCToDFvector, intervalRegionToDFvector )
```

Summary of algorithm The algorithm starts with setting the boundary for each interval's foreign key values. The upper bound on the number of foreign key values associated with each interval region is the total number of tuples in the fact table(foreign key relation). Once the linear equations for join are added, then for each interval region associated foreign key boundaries available beforehand as a result of the LP solver.

After setting foreign key values boundaries for each interval region, the algorithm using $maxDIntervalRetirver$ greedily finds the interval region having a maximum d value in its duplication vector. Then, using $findAppropriateDF$, find the $appD$ and $appF$ for the selected interval region, which is the intervalised region with the most overlapping CCs, and all foreign keys are yet to be assigned to it. If $appD$ == -1, then put all leftover duplication vector of interval region to $futureFKValDistribution$ and remove that interval region from $intervalRegions$ list. Else use $distributeToVarFKValDistribution$ to distribute foreign key values with their duplicate counts among intervalised regions of selected interval and update duplication vector of interval region by removing (dMax,fMax) if $dMax = appD$ and $fMax = appF$; else update $(dMax, fMax)$ in duplication vector to $(dMax, fMax - appF)$. Once the duplication vector of all interval regions becomes empty (i.e. all interval regions are done with the distribution of foreign keys either to intervalised regions or $futureFKValDistribution$). In the end, all the foreign key values inside $futureFKValDistribution$ will be distributed to the intervalised regions corresponding to their interval region. As a result of the algorithm, we'll have a duplication vector associated with the intervalised region.

5.4 Data Generation

After distributing duplication vectors to each intervalised region of foreign keys in a view, we have a set of sub-views associated with some foreign keys and their corresponding intervalised regions. These sub-views are merged using the same sub-view-merging algorithm proposed in Hydra [7]. If an intervalised region(I_r) is part of some partitioned region(P_r) in a sub-view, and after merging all sub-views partitioned region(P_r) breaks down into multiple regions, then intervalised region(I_r) will map to some set of regions broken down from (P_r). Algorithm 4 deals with the mapping of all intervalised regions with their regions in the view constructed after merging all sub-views.

Algorithm 4: View Summary Generation

```
   Result: View Summary Generation
1  IRList = Intervalised Region List;
2  fkeysList = List of foreign key associate with view V ;
3  allRegions = regions after sub-view merging in a view
4  for $fkey \leftarrow fkeysList$ do
5      IRList = Intervalised Region List for fkey;
6      for $I_r \leftarrow IRList$ do
7          for $reg \leftarrow allRegions$ do
8              if $reg$ overlaps with $I_r$ then
9                  Map (fkey,reg) with $I_r$
10             end
11         end
12     end
13 end
```

Tuple generation The summary gives the list of intervalised regions mapped with regions for each view V. The tuples are generated using view summary. For each region r in a view summary. With region having cardinality x_r will generate x_r tuples. All x_r tuples will contain non-key attributes of view, which will get attribute values from region information, similar to

Hydra[7]. And, the foreign keys associated with the view will get foreign key values from intervalised regions associated with each region. The foreign key values follow the same duplication distribution obtained as a result of the duplication distribution algorithm.

Chapter 6

Experimental Evaluation

6.1 Experimental Setup

DupGen is implemented on Java, and PostgreSQL v9.6 as the database engine. The database and *DupGen* ran on a machine with 3.30GHz $\times$ 20 processor and 32GB of RAM with Ubuntu 18.04 as the operating system and Z3 as LP solver.

6.2 Size of duplication vector

This section deals with comparing the duplication vector size with the actual bound on maximum size of duplication vector. Table 6.1 shows that the bound on the size of duplication vector is quite less and on an average the size of duplication vectors are also quite smaller than the bound on duplication vector. Table 6.1 contains 4 relations from TPC-DS Benchmark database. For 1 GB TPC-DS database, the duplication data extracted was around 40 KB, which is quite less in comparison to actual database size.

Relation	**Min. size**	**Avg. size**	**Max. size**	**Bound** ($\beta = O(\sqrt{\lvert\mathcal{R}\rvert})$)	**Total tuples**
store_sales	6	257	924	1620	2.6 millions
catalog_sales	6	194	864	1195	1.4 millions
customer	5	24	37	317	0.1 millions
inventory	1	3	5	3428	11.7 millions

Table 6.1: Duplication vector size comparison

6.3 Duplication distribution at client site

We implemented online duplication distribution computation in the execution pipeline of PostgreSQL. Figure 6.1, shows the accuracy, time and memory analysis of duplication vectors for two cases of the approximate algorithm(with error parameter 0.001 and 0.0002) and one without approximation algorithm on 1 GB TPC-DS database. Having an error parameter as 0.0001 ensures that, for 10,000 tuples in the relation, those values will appear in the result, which has frequency counts of more than 10. By seeing the behavior of queries in the graph, we can understand that duplication distribution computation without approximation consumes more memory and time than the approximate. And, as we increase the approximation, we are approaching towards the lesser accuracy, memory, and time consumption. So having an approximation algorithm for computing duplication distribution has significantly reduced the time and space overheads, but with a trade-off of reduction in accuracy of the duplication vector. .

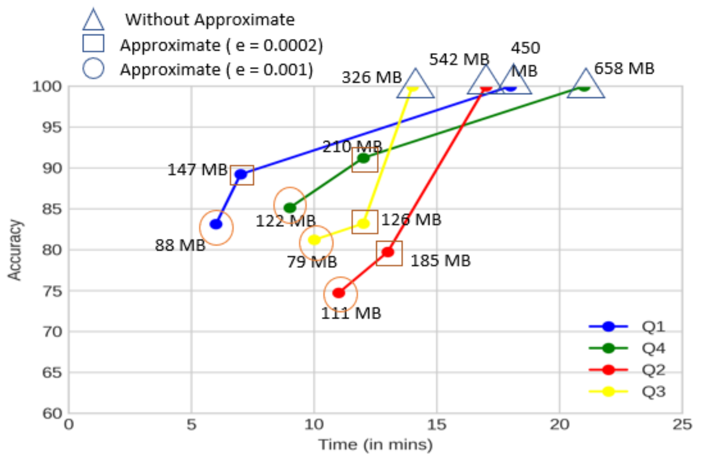


Figure 6.1: Accuracy, time and space analysis for online duplication distribution computation

6.4 Data generation at vendor site

The data generated by *DupGen*, mimics duplication vectors at base relation and tries to mimic duplication vectors for cardinality constraints generated from query workload. Data generation is done at view level. The data produced by our algorithm is 100% accurate on volumetric symmetry at all nodes of query plans. Furthermore, the duplication vector of all foreign keys in base relation matches with 100% accuracy with the duplication vectors of generated synthetic data.

For testing purpose, we have used the approximated duplication vectors. We approximated the size of duplication vectors to 10, for all duplication vectors having size more than 10. The approximation approach is quite naive for now. For restricting the size of duplication vectors to 10, since duplication vectors on sorted on the decreasing order of d values, so first nine (d, f) values are picked by default and for the last (d, f) we keep d as 1, f as remaining tuples in the duplication vector that are not covered in approximated duplication vector.

The accuracy of a duplication vector of intermediate nodes of query plan is evaluated by comparing duplication vectors of foreign key for each cardinality constraint. Figure 6.2 shows the accuracy of foreign duplication vectors on different cardinality constraints. The results shown in the figure 6.2 is based on 45 query workload with 84 cardinality constraints on intermediate nodes. Although, as can be seen from figure 6.2, there are some queries that are giving good accuracy, these CCs are coming from the intervalised regions which are having more number of overlapping CCs. And, the main reason behind the lower accuracy of duplication vectors at more intermediate nodes is because of less constraints for the duplication vector of intermediate node cardinality constraints.

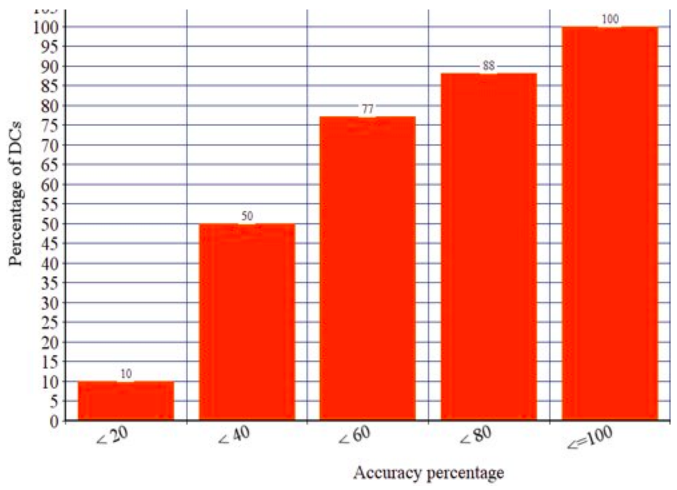


Figure 6.2: Accuracy at intermediate nodes of queries

Running Time For 45 queries, running time of view summary generation takes around 42 minutes for approximated duplication vectors. The major reason behind the difference between running time for actual vs approximated duplication vector is the LP solving stage. In case of actual duplication vectors, even running for 5 queries, the algorithm took around 6 hrs to complete, which clearly suggests that there is a need of an approximation algorithm.

In the overall algorithm as well, the LP solving stage consumes a lot amount of time compare to other stages of whole data generation algorithm.

Chapter 7

Conclusion and Future Work

7.1 Future Work

In experiments we have used approximated duplication vectors instead of actual duplication vectors. So instead of using current approximation technique, we need to find out some other alternative, which can limit the size of duplication vector to x, and also have minimum duplication error between approximated duplication vector and actual duplication vector.

Also second important thing is to handle joins. Currently, *DupGen* solves LP equations at view level, that is only one view is solved at a time by LP solver. Instead of solving all views in a single LP with some additional constraints for joins added to LP solver can handle joins. The join constraints should handle the case of having sufficient number of primary key values in regions of relation having primary key, so that those primary keys can be used to distribute as foreign keys in the relation having foreign key. Also finding possibilities of adding more constraints related to duplication vectors of CCs, which can help us to improve the accuracy of duplication error.

7.2 Conclusion

DupGen currently generates data that maintains volumetric similarity for all nodes in query plans and generates a duplication vector of all foreign keys at base relations with 100% accuracy. Having synthetic data that satisfies duplication distribution has various use cases like in the case Hash operator used for aggregation operation performed on synthetic data. The performance of the hash operator will be similar for both actual and synthetic data. In the future, several works are needed to be done like handling joins and finding an algorithm to approximate the duplication vector, and explore more constraints that can be added to the LP solver for better results.

Bibliography

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

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

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

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

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

[6] PostgreSQL: https://www.postgresql.org/ 4

[7] A.Sanghi and R.Sood and J.R.Haritsa and S.Tirthapura. Scalable and Dynamic Regeneration of Big Data Volumes. *EDBT Conference*, 2018. ii, 1, 14, 15, 16, 17, 27, 28

[8] 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

[9] G. S. Manku, R. Motwani. Approximate Frequency Counts over Data Streams. *VLDB, 2002.* 12

[10] 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.*

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

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

[13] 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.*

[14] 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.* 1

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

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

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

[18] J. W. Zhang, Y. C. Tay . A Collaborative Framework for Similarity Enforcement in Synthetic Scaling of Relational Datasets *ICDE, 2019.*