Performance Testing Environments For Database Systems

Anupam Sanghi
M.E, CSA, IISc Bangalore
SR No: 04-04-00-10-41-14-1-11143

June 14, 2016

ME Project Report

Abstract

Database systems' testing, in the state of the art uses synthetic data and workload generators such as TPC-H and TPC-DS. Synthetic generators are used because the real customer data is hard to obtain due to its sensitive nature as well as huge size. But these synthetic benchmarks usually have a fixed schema and workload, and also they do not provide much flexibility in data generation except for the size of the database. Due to these limitations, they often fail to capture realistic scenarios. In this project, we have constructed a *workload-dependent* synthetic data generator, which ensures that the query performance is similar on the real and simulated environments for a predefined workload. By similar query performance, we mean that the executor does similar volumetric processing of data for each node in the execution plan tree. The generation technique is independent of the size of the database and has a unique feature of generating and supplying data *on-the-fly*. This eliminates the data loading and storing overheads and can can therefore be helpful in futuristic *big data* scenarios as well, where these features are indispensable. Since, the generator does not require any static data storage, it is well adapted to the philosophy of CODD (COnstructing Dataless Databases), a tool that helps to do compile-time system testing by constructing *dataless databases*.

1 Introduction

Effective testing of database engines and applications is predicated on the ability to easily construct alternative scenarios with regard to the database contents [1]. In the state of the art, it involves executing a set of queries (known as *query workload*) on synthetic databases. A query is processed in two phases - at *compile-time*, the *query optimizer* picks an optimal plan from a set of plans. Thereafter, the executor runs the chosen plan on the database, also called as *execution-time* processing, to give the result. Therefore, testing can be broadly categorized into *compile-time testing* and *execution-time testing*. There are several synthetic benchmarks like TPC-H [2], TPC-DS [3] that are commonly used to carry out performance tests in various domains. But unfortunately, these benchmarks are far from realistic customer scenarios. Due to its sensitive nature, the customer data is also hard to obtain. This therefore results in unidentified bugs before deployment [4]. Database performance testing tries to capture the performance-related bugs in the engines.

The major **limitations** of techniques that use synthetic benchmarks are as follows:

- **Fixed schema and workload:** The synthetic data generators possess a fixed schema and workload, which fail to accommodate various realistic settings.
- **Inability to tune the generated data:** Not much flexibility is given in generation. The data usually has a fixed predefined distribution, which might again fail to adapt to the real scenarios.
- **Infeasible at big data scale:** In today's era of so-called *big data* systems, the data is of prohibitively large sizes. It is easy to see that the traditional testing techniques become completely impractical simply because of the *time*

and *space* overheads that arise due to the tedious data loading and storing processes.

Motivated by the above problem, [1] proposed a tool called **CODD**[1] (COnstructing Dataless Databases), that took the first step towards alleviating the problem of effective testing of big data systems, by implementing a new metaphor of *"data-less databases"*. It aims to provide a platform to construct virtual databases with no static (stored) data, using which query execution can be simulated easily. This can therefore overcome the above mentioned limitations.

1.1 Compile-time Testing

CODD currently supports *compile-time* testing aspect of the database systems[2]. At compile-time, the *query optimizer* uses the *meta-data* characteristics of the database to pick the optimal *plan* for execution. Thus, this stage does not require the actual data to be present in the system. CODD leverages this property by providing a platform to directly construct the metadata. Therefore, database environments with the desired metadata characteristics can be efficiently simulated without persistently generating and/or storing their contents. Correct simulation implies obtaining identical *plan diagrams* [5], indicating that the optimizer behaves identically on the real and simulated scenarios. Another unique feature of CODD is its support for automated scaling of meta-data instances, which can be used to mimic futuristic scenarios. For instance, consider the situation where a database engineer wishes to evaluate the query optimizer's behaviour on a futuristic big data set-up featuring "yottabyte"(10^{24} bytes) sized relational tables. Obviously, just generating this data, let alone storing it, is practically infeasible even on the best of systems, but using CODD this can be easily modelled within a few minutes on a vanilla laptop [6].

1.2 Execution-time Testing

Unlike compile-time, execution-time testing does need the *actual data* to be present, at least transiently, in the system. Therefore, solving the problem for execution-time testing becomes challenging. To handle this, we used the technique of workload-dependent data generation to obtain synthetic data that can mimic the performance of the real database on a predefined workload. By mimicking, we mean that the executor does similar volumetric processing of data for each node in the execution plan tree in both real and simulated environments. Further, our generation mechanism does not depend on the size of the database, something indispensable in big data scenarios.

1.3 Applications

Volumetric execution-time testing as mentioned above has numerous applications like:

- **Component Testing:** On introducing a new component/technique into the engine, testing its performance is required. This can be done easily by using our generator to get data with the desired properties. This data can be supplied to the executor on demand, which can forward it to the newly added component.
- **Resolve customer issues:** This application was motivated from HP Enterprise. Client organizations outsource the testing of their database applications. Here, the data generator can serve as a platform that can provide synthetic data possessing the desired properties of the client's data. This would eliminate the requirement of getting access to the client's production system and yet can resolve the performance related problems of the engine. For example, solutions might be (a) propose a hardware configuration, (b) new algorithm for an operator for efficient execution.
- **Testing on Futuristic Scenarios:** If the organisations want to perform testing on futuristic scenarios, where the data size is up/down-scaled, then this can also be done easily by scaling the inputs given to CODD and it will ensure that the generated metadata and synthetic data mimics the desired scenario.

1.4 Prior Work

There are various techniques in the literature that deal with synthetic data generation. Some techniques like [7, 8] focus on making the generation process parallelizable and scalable for pre-defined data distributions. But, these fail to resemble the real scenarios as they do not capture the correlations that exist between the attributes within and across tables. Also there has been some work done in the direction of generating

[1]In archaic English, *cod* means "empty shell", symbolising the data-less context.

[2]Currently CODD supports several industrial-strength optimizers, including IBM DB2, Microsoft SQL Server, Oracle, HP SQL/MX and PostgreSQL.

data from the statistical metadata [4], but this also loses the non-trivial correlations and therefore fail to mimic the real scenarios. The idea of using annotated query plan trees was introduced by Binnig et. al. in a tool called QAGen [9, 10], and was further extended in a tool called MyBenchmark [11, 12]. This work was based on the approach called *symbolic query processing.* It constructs a symbolic database[3] on a per query basis, to which constraints are added depending on the input plans and finally these symbolic databases are integrated and instantiated by using *constraint satisfaction program.* Though this technique is able to handle a larger set of queries, it defeats our purpose of making an on-the-fly data generator since it would require N symbolic databases for N input trees in the very first step of the algorithm.

Our implementation is based on the work done by Arasu et. al. [13, 14] as the technique is in principle independent of the size of the generated database. We have modified their technique to:

- **support on-the-fly data generation**: In the earlier technique, a view is created corresponding to every relation and a *linear program* (LP) is formulated for each view separately. Thereafter, from the LP solution, the views are instantiated and relations are constructed from them. The modified technique does not require instantiating the views, rather we create relation summaries. These summaries (much smaller in size than the actual relations) are capable of supporting on-the-fly generation.

- **handle larger class of schemas**: The earlier technique handled only *snowflake schemas.* We have extended it to support any schema with non-cyclic dependencies between the relations. The snowflake schema is a special category of non-cyclic dependencies, where the dependency graph is a tree.

- **reduce the error in satisfying the constraints**: The errors in the earlier technique were an outcome of two cases: (i) errors as a result of maintaining view consistency and (ii) sampling technique used for data generation. We have made the generation algorithm deterministic, which ensures that no errors due to sampling gets added up.

- **accept generic projections**: The earlier work did not handle generic projection operators. They were limited to the cases where the projections are *non-overlapping.* We have proposed a new technique that can handle any kind of projection operators.

Our generator takes the database *schema* and a set of ALQP as the input. From these ALQP set, we generate a *cardinality constraint* for each operator in the plan tree. Further, these constraints are converted into a *linear program*, which then is passed on to the *LP solver.* The result from the LP solver is finally fed into the *relations generator* that has the capability of supplying tuples on demand.

We verified our work on TPC-DS benchmark, where we constructed 14 queries by modifying the original queries to suit the assumptions that the algorithm requires. In our results, we found that the query performance, with regard to the volumetric processing at each node of the plan tree, was very similar on the real 10 GB TPC-DS database and our synthetically generated one.

1.5 Roadmap

In the rest of this document, we shall first discuss the concept of ALQPs and cardinality constraints along with some notations that will be used often in Section 2. Further, we go on and formally define the problem and state the assumptions made in Section 3. Thereafter, Sections 4 and 5 discuss the architecture of the data generator along with a detailed description of its various components. In Section 6, we propose an algorithm for handling generic projections. Section 7 includes an empirical evaluation of our generator and finally we conclude with various open problems that can be explored.

2 Preliminaries

2.1 Annotated Logical Query Plan (ALQP)

An ALQP gives the sequence of operators in the skeleton of a query plan along with the input and output cardinalities for each relational algebraic operator. For example, consider a database having relations: R (a, b), S (p, q, r) and T (x, y) and the following query:

[3]A symbolic database is like a regular database, but its attribute values are symbols (variables) not constants.

```
SELECT *
FROM R, S, T
WHERE S.q = T.y
AND R.a = S.p
AND T.x <= 30
AND R.b <= 80
AND S.r > 100
```

For the above query, a possible ALQP in shown in Figure 1.

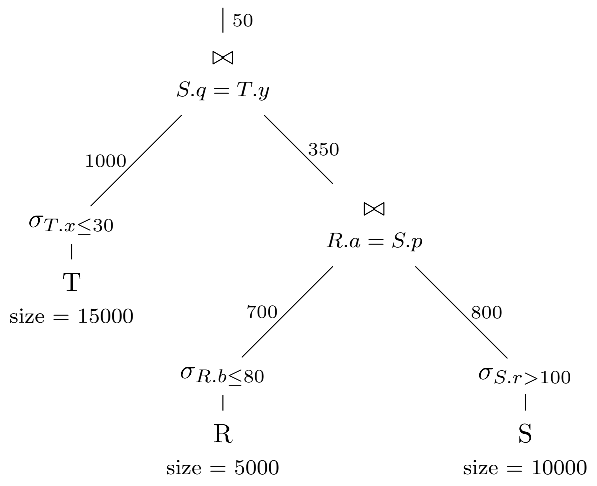

Figure 1: Annotated Logical Query Plan

2.2 Cardinality Constraints

Assuming that $\mathcal{R}_1, ..., \mathcal{R}_l$ are the set of relations in the database, each operator in the ALQP corresponds to a *cardinality constraint* that can be written in the following form:

$$|\pi_{\mathbb{A}}\sigma_{\mathcal{P}}(\mathcal{R}_{i_1} \bowtie ... \bowtie \mathcal{R}_{i_p})| = k$$

where $\mathcal{A}$ denotes a set of attributes, $\mathcal{P}$ is a selection predicate, and k is a non-negative integer.

For example, the constraints corresponding to the ALQP shown in Figure 1 are:

$$|T| = 15000 \tag{2.1}$$

$$|R| = 5000 \tag{2.2}$$

$$|S| = 10000 \tag{2.3}$$

$$|\sigma_{T.x\leq 30}(T)| = 1000 \tag{2.4}$$

$$|\sigma_{R.b\leq 80}(R)| = 700 \tag{2.5}$$

$$|\sigma_{S.r>100}(S)| = 800 \tag{2.6}$$

$$|\sigma_{S.r>100\ \wedge\ R.b\leq 80}(R \bowtie S)| = 350 \tag{2.7}$$

$$|\sigma_{S.r>100\ \wedge\ R.b\leq 80\ \wedge\ T.x\leq 30}(R \bowtie S \bowtie T)| = 50 \tag{2.8}$$

3 Problem

3.1 Statement

We shall now formally state the problem:

> Given a database schema $\mathcal{S}$ and a set of ALQPs $\mathcal{W}$, generate a database instance that conforms to $\mathcal{S}$ and satisfies all the cardinality constraints $(\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_m)$ generated from $\mathcal{W}$.

3.2 Assumptions

The assumptions made in our work are:

- All the joins appearing in the constraints are *primary key-foreign key* joins.
- The dependency (due to joins) between relations should be *non-cyclic.*
- Selection predicates include only non-key attributes.

4 Architecture

Figure 2 provides an overview of the architecture of the data generator. Its various components are:

- **Parser:** The purpose of parser is to take $\mathcal{W}$ as input and give the set of cardinality constraints as described in Section 2.2. Note that for now we are not considering projection operator. Therefore, all the constraints will be of the form:

$$|\sigma_{\mathcal{P}}(\mathcal{R}_{i_1} \bowtie ... \bowtie \mathcal{R}_{i_p})| = k \tag{4.1}$$

- **View Generator:** This component takes the database schema as input and creates a view $\mathcal{V}_i$ corresponding to every relation $\mathcal{R}_i$. Creation of views help us to get rid of the join expressions in the constraints, i.e., once the views are created, each constraint can be re-written as a selection predicate over a single view only. We shall see this component in detail in Section 5.1.
- **LP Formulator:** This component uses the views given by the view generator to re-write the cardinality constraints given by the parser. Further, an LP is created for each view. This is done by converting each constraint (on that view) to a corresponding equation. Once this is done, the system of equations is passed on to the LP solver. We shall discuss the details of constructing equations in detail in Section 5.2.

- **LP Solver:** This component takes the system of equations and gives one of the feasible solutions[4]. We use the GNU Linear Programming Kit (GLPK) [15] for solving the LP.
- **Relations Generator:** This component takes the solution given by the LP solver and (i) makes it consistent across the views, (ii) constructs compressed relations from it, which is sufficient to generate the entire relation. Section 5.3 discusses this component in detail.
- **Tuple Generator:** The compressed relations that we obtain from the relations generator serve as the seeds to the tuple generator. The tuple generator has the capability of generating a tuple(s) on demand for any relation $\mathcal{R}_i$. The detailed description of this component is mentioned in section 5.4.

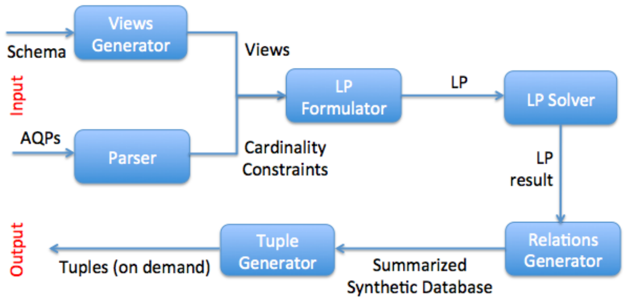

Figure 2: Architecture

5 Details

5.1 View Generator

As discussed earlier, the purpose of this component is to simplify the constraints such that we can replace all the join expressions by a single view. For this we need to construct a view $\mathcal{V}_i$ corresponding to each relation $\mathcal{R}_i$. A view $\mathcal{V}_i$ can be considered as a set of non-key attributes that are present in either $\mathcal{R}_i$ or in any other relation on which $\mathcal{R}_i$ depends. The dependencies between relation can be seen from the *dependency graph*.

Dependency Graph: In a dependency graph, we create a node for every relation. A directed edge from a node $\mathcal{R}_i$ to $\mathcal{R}_j$ is added, if $\mathcal{R}_i$ contains a *foreign-key* referencing $\mathcal{R}_j$.

Now, a relation $\mathcal{R}_i$ is said to be dependent on relation $\mathcal{R}_j$ if there exists a path from $\mathcal{R}_i$ to $\mathcal{R}_j$ in the dependency graph.

Let us see the following example: Consider a database having following four relations:

$$Catalog\_sales(\underline{PK_1}, FK_C, FK_D, cs\_sales\_price)$$

$$Date\_dim(\underline{PK_3}, d\_qoy, d\_year)$$

$$Customer(\underline{PK_2}, FK_{CA})$$

$$Customer\_address(\underline{PK_4}, ca\_state)$$

Here, PK_1, PK_2, PK_3, PK_4 are the primary keys of the respective relations. FK_C references to $Customer$, FK_D references to $Date\_dim$ and FK_{CA} references to $Customer\_address$. Therefore, the dependency graph would be as shown in Figure 3.

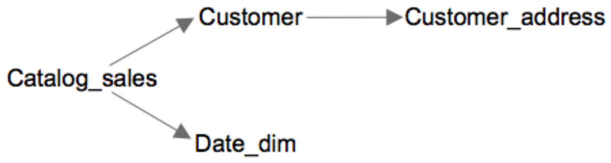

Figure 3: Dependency Graph

After executing the View Generation Algorithm (shown in Algorithm 1), the views thus obtained would be:

$$Catalog\_sales'(ca\_state, d\_qoy, d\_year, cs\_sales\_price)$$

$$Date\_dim'(d\_qoy, d\_year)$$

$$Customer'(ca\_state)$$

$$Customer\_address'(ca\_state)$$

[4]The system of equations can have infinite solutions. The solution corresponding to the original database instance might differ from the solution we get here. But, both the solutions would satisfy all the constraints.

Algorithm 1 View Generation Algorithm

```
Input: $\{\mathcal{R}_i\}_{i\in[n]}$                ▷ Set of relations $\mathcal{R}_1$,..,$\mathcal{R}_n$
Output: $\{\mathcal{V}_i\}_{i\in[n]}$               ▷ Set of views $\mathcal{V}_1$,..,$\mathcal{V}_n$
    Functionalities Used:
      • getFK($\mathcal{R}_i$)    ▷ Returns set of $< attr, ref >$ pairs corresponding to the FKs in $\mathcal{R}_i$
      • getNonKey($\mathcal{R}_i$)    ▷ Returns set of non-key attributes in $\mathcal{R}_i$
 1: procedure GETVIEW($\mathcal{R}_i$,$\{\mathcal{V}_i\}_{i\in[n]}$, $\{flag_i\}_{i\in[n]}$)
 2:     if !($flag_i$) then
 3:         $\mathcal{V}_i \leftarrow$ getNonKey($\mathcal{R}_i$)
 4:         for $< attr, ref >$ in getFK($\mathcal{R}_i$) do
 5:             $\mathcal{V}_i \leftarrow \mathcal{V}_i \cup$ GETVIEW($ref$, $\{\mathcal{V}_i\}_{i\in[n]}$, $\{flag_i\}_{i\in[n]}$)
 6:         end for
 7:         $flag_i \leftarrow true$
 8:     end if
 9:     return $\mathcal{V}_i$
10: end procedure
11: Init: $\mathcal{V}_i \leftarrow \emptyset$, $flag_i \leftarrow false\ \forall i \in [n]$
12: for $i \leftarrow 1$ to $n$ do
13:     $\mathcal{V}_i \leftarrow$ GETVIEW($\mathcal{R}_i$, $\{\mathcal{V}_i\}_{i\in[n]}$, $\{flag_i\}_{i\in[n]}$)
14: end for
```

5.2 LP Formulator

The LP Formulator receives the set of constraints from the parser and the set of views from the view generator. The first task it does is to re-write the constraints by replacing the join expressions with appropriate views. Say we have a constraint having join: $\mathcal{R}_i \bowtie \mathcal{R}_j$. Since, we have assumed that all joins are of primary key-foreign key type, one of these is a dependent relation. Say $\mathcal{R}_i$ depends on $\mathcal{R}_j$. In such a case, we shall replace the expression $\mathcal{R}_i \bowtie \mathcal{R}_j$ with $\mathcal{V}_i$. Because of the way we constructed the views, it is easy to see that we can apply the predicate that was there on $\mathcal{R}_i \bowtie \mathcal{R}_j$ to $\mathcal{V}_i$. Likewise, all the join expressions can be expressed in terms of a single view respectively.

So now we can solve for each view separately. For each view $\mathcal{V}_i$, we will find the set of constraints imposed on $\mathcal{V}_i$. A constraint $\mathcal{C}_j$ on $\mathcal{V}_i$ will be of the following form:

$$|\sigma_{\mathcal{P}_j}(\mathcal{V}_i)| = k_j \tag{5.1}$$

Now, let us assume we have a single view $\mathcal{V}$ having a set of attributes A_1, A_2,...,A_n. We need to formulate an LP for the view $\mathcal{V}$. Let the domain of an attribute A_i be represented by $Dom(A_i)$. We assume that the domain of all the attributes are positive integers bounded by an integer D. For attributes with non-integral domain, we can map the values to integers. This assumption is to simplify the analysis and can be removed easily.

Say we are given a set of m constraints that $\mathcal{V}$ satisfies. Each constraint C_j $(1 \leq j \leq m)$ for simplicity can be expressed as: $< \mathcal{P}_j, k_j >$, which means that the number of tuples (rows) satisfying the condition $\mathcal{P}_j$ is equal to k_j.

For every tuple $t \in Dom(A_1) \times Dom(A_2) \times ... \times Dom(A_n)$, we create a variable x_t representing the number of copies of t in $\mathcal{V}$. Now, for each of the m constraints C_j $(1 \leq j \leq m)$, we create a linear equation:

$$\sum_{t:P_j(t)=true} x_t = k_j$$

In addition, we also require that $x_t > 0\ \forall t$. Here, since the number of variables in the LP is proportional to the domain size, which can be huge, there are some optimizations that can be done to reduce the size of the LP. We will now discuss these optimizations.

5.2.1 Domain Decomposition

A set v^i is created for each attribute A_i. Values in v^i are added according to the following: We iterate over the constraints and if a constraint has

- $A_i >= a$ or $A_i < a$, we add a in v^i.
- $A_i > a$ or $A_i <= a$, we add $a + 1$ in v^i.
- $A_i = a$, we add a and $a + 1$ both in v^i.

All the other constraints can be expressed as combinations of the above. In addition, we also add 1 (minimum value in domain) in v^i if not already present. Let $v^i_1, v^i_2, ..., v^i_{l_i}$ represent the constants (in increasing order) in the set v^i. Now we can divide the domain of an attribute A_i into a set of l_i intervals $I^i : [v^i_q, v^i_{q+1})$ $(1 \leq q < l_i) \cup [v^i_{l_i},)$. The semantics of the variables can now be modified such that we introduce a variable $x_{t'}$ for each interval combination $t' \in I^1 \times I^2 \times ... \times I^n$, representing the number of tuples lying in the interval combination t'.

Therefore, now for each constraint C_j $(1 \leq j \leq m)$, the linear equation would be:

$$\sum_{t':P_j(t')=true} x_{t'} = k_j$$

Here as well we will have the additional constraint of $x_{t'} > 0\ \forall t'$. *Example:* Let us see the LP formulation for relation *Catalog_sales* having following constraints:

$$|CS| = 14401261$$

$$|\sigma_{cs\_sales\_price>150}(CS)| = 734606$$

$$|\sigma_{cs\_sales\_price>150 \wedge ca\_state=1}(CS)| = 13806$$

These constraints can be converted into the corresponding LP equations as follows:

$$x_{[1,151)[1,2)} + x_{[1,151)[2,)} + x_{[151,)[1,2)} + x_{[151,)[2,)} = 14401261 \quad (5.2)$$

$$x_{[151,)[1,2)} + x_{[151,)[2,)} = 734606 \quad (5.3)$$

$$x_{[151,)[1,2)} = 13806 \quad (5.4)$$

Therefore, we can see that by applying this optimization, the number of variables are a function of the size of the sets $v^i s$ instead of the domain size. But, even after applying this optimization, the number of variables are exponential in the number of attributes. To further reduce the LP's complexity, we will next look at another optimization that tries to decompose the view into smaller views.

5.2.2 View Decomposition

In this optimization, we will decompose the view into small components having fewer attributes. The constraints will then be applied to these small components. Since, the number of attributes are fewer, the size of the LP would also reduce.

The algorithm consists of the following steps:

- Constructing a graph from the constraints.
- Identifying the smaller components.
- Applying constraints on the smaller components.

Graph Construction: Construct a graph $G = (V, E)$, where vertices corresponds to the attributes in the view and we add an edge (A_i, A_j) in G if there exists a constraint C in which attributes A_i and A_j co-appear. We also add more edges to the graph in order to make it *chordal*. A graph is chordal if each cycle of length 4 or more has a *chord*; a chord is an edge joining two non-adjacent nodes of a cycle. It is easy to see that a chord joining vertices corresponding to attributes A_i and A_j can always be added by assuming that the original set of constraints had the trivial constraint:

$$|\sigma_{A_i \geq 1 \wedge A_j \geq 1}| = |R|$$

We converted the graph to chordal because the chordal graphs have a special property [16] that allows us to construct the original view from the decomposed components.

Identifying Smaller Components: These smaller components are nothing but the maximal cliques obtained from the graph. Let $\mathcal{A}_{c_i}$ represent the set of attributes that clique c_i contains.

Adding Constraints: Instead of applying constraints to the complete view, we shall now apply the constraints to the cliques. For each clique, we will add all the constraints that are within their scope. In addition, since cliques can have common attributes as well, we need to add more constraints to ensure consistency across cliques. Consider two set of attributes $\mathcal{A}_{c_i}$ and $\mathcal{A}_{c_j}$ such that $\mathcal{A}_{c_i} \cap \mathcal{A}_{c_j} \neq \emptyset$. Further for any $p \in Dom(\mathcal{A}_{c_i} \cap \mathcal{A}_{c_j})$, let $Ext_{\mathcal{A}_{c_i}}(p)$ denote the set of assignment to $\mathcal{A}_{c_i}$ that is consistent with the assignment p. We include the following equation for each $p \in Dom(\mathcal{A}_{c_i} \cap \mathcal{A}_{c_j})$ in the LP:

$$\sum_{y \in Ext_{\mathcal{A}_{c_i}}(p)} x_y = \sum_{z \in Ext_{\mathcal{A}_{c_j}}(p)} x_z$$

5.3 Relations Generator

The LP solver gives the result for each clique c_i in the form of the set:

$$\{(x, k_x^i)\}_{x \in Dom(\mathcal{A}_{c_i})}$$

where k_x^i represents the number of tuples in the view that have x value combination in the attributes present in $\mathcal{A}_{c_i}$. Lets call this as the clique-solution set. We might not explicitly mention it always, but when we say $x \in Dom(\mathcal{A}_{c_i})$, we consider only those domain values for which the corresponding k_x^i is non-zero since we do not need to store the domain values that do not occur from our LP solution.

The relations generator does the following:

- First, it merges the clique-solution sets to obtain the solution for the complete view.
- Then, it makes the views consistent.
- Finally, it constructs relations summaries.

5.3.1 Obtaining View Solution

The solution for the complete view is obtained by *merging the cliques* (Algorithm 2). This is done by first ordering the cliques using the *Order Cliques Procedure* (Algorithm 3). Thereafter, we merge the cliques-solution sets one by one. For merging, we first sort the clique-solution sets based on their common attributes and then use the *Align Procedure* (Algorithm 4) and *Merge Procedure.* The Align Procedure bucketize the two clique-solution sets simultaneously so that the bucket sizes can become the same in the two sets.

The Merge Procedure simply merges the solutions in the two sets one-to-one.

Algorithm 2 Cliques Merging Algorithm

```
Input: ${\{(x, k_x^i)\}_{x \in Dom(\mathcal{A}_{c_i})}}_{i\in[l]}$, $\{c_i\}_{i\in[l]}$   ▷ Set of value combinations with cardinalities for each clique and the set of cliques
Output: $\mathbb{X}_\mathcal{V}$   ▷ Set of value combinations with cardinalities for the view $\mathcal{V}$
    Functionalities Used:
      • $sort(\mathbb{X}, S)$   ▷ Sort $\mathbb{X}$ on value combinations corresponding to the attributes present in the set $S$
1: Init: $\mathcal{V}' \leftarrow \emptyset$
2: $\mathbb{C} \leftarrow$ OrderCliques($\{c_i\}_{i\in[l]}$)
3: $\mathcal{V}' \leftarrow \mathbb{C}[0]$
4: $\mathbb{X}_{\mathcal{V}'} \leftarrow \{(t, k_t)\}_{t\in Dom(\mathcal{V}')}$
5: for $j \leftarrow 1$ to $l - 1$ do
6:     $\mathbb{X}_{\mathbb{C}_j} \leftarrow \{(x, k_x)\}_{x\in Dom(\mathbb{C}[j])}$
7:     $\mathbb{X}_{\mathbb{C}_j} \leftarrow sort(\mathbb{X}_{\mathbb{C}_j}, \mathcal{V}' \cap \mathbb{C}[j])$
8:     $\mathbb{X}_{\mathcal{V}'} \leftarrow sort(\mathbb{X}_{\mathcal{V}'}, \mathcal{V}' \cap \mathbb{C}[j])$
9:     $\mathbb{X}_{\mathcal{V}'}, \mathbb{X}_{\mathbb{C}_j} \leftarrow$ Align($\mathbb{X}_{\mathcal{V}'}$, $\mathbb{X}_{\mathbb{C}_j}$)
10:    $\mathcal{V}' \leftarrow \mathcal{V}' \cup \mathbb{C}[j]$
11:    $\mathbb{X}_{\mathcal{V}'} \leftarrow$ Merge($\mathbb{X}_{\mathcal{V}'}$, $\mathbb{X}_{\mathbb{C}_j}$)
12: end for
13: $\mathbb{X}_\mathcal{V} \leftarrow \mathbb{X}_{\mathcal{V}'}$
```

5.3.2 Making Views Consistent

So, after using the Cliques Merging Algorithm, we obtain the solutions for every view. A solution for a view $\mathcal{V}_i$ is of the form:

$$\{(x, k_x)\}_{x\in Dom(\mathcal{V}_i)}$$

These solutions might be inconsistent as these independently solved views might not be able to give a valid solution for the relations. The relations have referential constraints that might get violated. In order to ensure that the final relations have referential integrity, we need to make the view solutions consistent. The technique used for this induces some error in satisfying the cardinality constraints, but in our experiments we show that these errors are minor and can be tolerated easily in realistic scenarios.

To make the views consistent, we use the *View Consistency Algorithm* (Algorithm 6). We first run a topological sort on the *dependency graph* (like the one shown in Figure 3). Since we assumed that the dependencies are non-cyclic, we are guaranteed to get an ordering of views. Now, for each view $\mathcal{V}_i$ in the order, if there exists a value combination x_{i-1} in the solution set of $\mathcal{V}_{i-1}$, for which if we project the value combination corresponding to the $\mathcal{V}_i's$ attributes, this projected solution (say x'_{i-1}) does not exist in solution set of $\mathcal{V}_i$. If so, we add another entry $(x'_{i-1}, 1)$ in the solution set of $\mathcal{V}_i$.

Algorithm 3 Order Cliques Procedure

```
Input: $\{c_i\}_{i\in[l]}$, $G$   ▷ Set of cliques and the graph $G$
Output: $\mathbb{C}$   ▷ Ordered set of cliques
    Functionalities Used:
      • $findAllPaths(P, Q)$   ▷ Returns all paths between two sets of vertices $P$ and $Q$
      • $getAdjacentCliques(\mathbb{C})$   ▷ Returns the set of cliques that share a vertex with any clique in the set of cliques $\mathbb{C}$
1: Init: $\mathbb{C} \leftarrow c_1$, $visisted \leftarrow c_1$, $k \leftarrow 1$
2: while $k \neq l$ do
3:     for $c_j$ in $getAdjacentCliques(\mathbb{C})$ do
4:         $flag \leftarrow true$
5:         $commonVertices \leftarrow visited \cap c_j$
6:         for $path$ in $findAllPaths(visited, c_j)$ do
7:             if $path \cap commonVertices = \emptyset$ then
8:                 $flag \leftarrow false$
9:                 break
10:            end if
11:        end for
12:        if $flag$ then
13:            $visited \leftarrow visited \cup c_j$
14:            $C[k] \leftarrow c_j$
15:            $k \leftarrow k + 1$
16:            break
17:        end if
18:    end for
19: end while
20: return $\mathbb{C}$
```

5.3.3 Constructing Relation Summary

Once we have the consistent views solutions, we next need to get the relation summaries from it. For this, we create a summarized relation set $\widetilde{\mathcal{R}_i}$ for each relation $\mathcal{R}_i$. This set consists of the attributes in $\mathcal{R}_i$ except the primary key attribute. In addition, we maintain the corresponding number of tuples for each entry in $\widetilde{\mathcal{R}_i}$ (like we have it in view solutions). For the attributes that are common between the summarized relation set and the corresponding view solution set, the value combinations and corresponding cardinalities are directly borrowed. What remains are the foreign key attributes. For filling a foreign key attribute fk, we first need to see the view corresponding to the relation that the foreign key refers to. Say the view thus obtained is $\mathcal{V}_j$. Now, to fill the fk value in r^{th} row of $\widetilde{\mathcal{R}_i}$, we see the value combination in the r^{th} row of view solution set of $\mathcal{V}_i$. From this value combination, we project the attributes of $\mathcal{V}_j$. Say the combination thus obtained is v. Now, we iterate over the solution set of $\mathcal{V}_j$ and keep summing up all the cardinality entries seen until we find v. This summed value gives us the fk value corresponding to r^{th} row of $\widetilde{\mathcal{R}_i}$.

We thus obtain the set $\widetilde{\mathcal{R}_i}$ for each relation $\mathcal{R}_i$. These are the relation summaries that are sufficient to generate tuples on demand.

Algorithm 4 Align Procedure

Input: $\{\mathbb{X}_i : (x_i, k_i)\}_{i \in m}$, $\{\mathbb{Y}_j : (y_j, l_j)\}_{j \in n}$ ▷ Set of value combinations with cardinalities for two sets of size m and n respectively
Output: $\mathbb{X}, \mathbb{Y}$ ▷ Returns the two sets after alignment
Functionalities Used:

- *split*$(\mathbb{Z}, r, value)$ ▷ Returns the set after splitting its r^{th} entry $(z_r, count_r)$ into $\mathbb{Z}_r : (z_r, value)$ and $\mathbb{Z}_{r+1} : (z_r, count_r - value)$

```
1: Init: $i \leftarrow 0$, $j \leftarrow 0$
2: while $\mathbb{X}_i$ exists do
3:     if $k_i < l_j$ then
4:         $split(\mathbb{Y}, j, k_i)$
5:     else
6:         if $k_i > l_j$ then
7:             $split(\mathbb{X}, i, k_j)$
8:         end if
9:     end if
10:    $i \leftarrow i + 1$
11:    $j \leftarrow j + 1$
12: end while
13: return $\mathbb{X}$, $\mathbb{Y}$
```

Algorithm 5 Merge Procedure

Input: $\{\mathbb{X}_i : (x_i, k_i)\}$, $\{\mathbb{Y}_i : (y_i, k_i)\} i \in [n]$ ▷ Set of value combinations with cardinalities for two sets of same sizes
Output: $\mathbb{Z}$ ▷ Merged Set

```
1: Init: $\mathbb{Z} \leftarrow \emptyset$
2: for $i \leftarrow 1$ to $n$ do
3:     $z_i \leftarrow x_i \cup y_i$
4:     $\mathbb{Z}_i \leftarrow (z_i, k_i)$
5: end for
6: return $\mathbb{Z}$
```

Algorithm 6 View Consistency Algorithm

Input: $\{\mathbb{X}^i : \{(x, k_x)\}_{x \in Dom(\mathcal{V}_i)}\}_{i \in [n]}$, Dependency Graph G ▷ Solution set for each view
Output: $\mathbb{Y}$ ▷ Consistent solution set for each view
Functionalities Used:

- *topologicalsort*(G) ▷ Returns a view tranveral order by taking the dependency graph as the input
- *project*$(\mathbb{X}^i, \mathcal{V}_j)$ ▷ Returns the value combinations in $\mathbb{X}^i$ corresponding to the attributes present in $\mathcal{V}_j$

```
1: Init: $\mathbb{Y}^1 \leftarrow \mathbb{X}^1$
2: for $i \leftarrow 2$ to $n$ do
3:     $tempset \leftarrow project(\mathbb{X}^{i-1}, \mathcal{V}_i)$ - $\mathbb{X}^i(x)$
4:     if $tempset \neq \emptyset$ then
5:         for $z$ in $tempset$ do
6:             $\mathbb{Y}^i \leftarrow \mathbb{Y}^i \cup (z, 1)$
7:         end for
8:     end if
9: end for
10: return $\mathbb{Y}$
```

5.4 Tuple Generator

The relation generator gives us the summaries for each relation. An entry in this set is of the form (x, k_x). We consider the PK values to be the row numbers of the relation. Therefore, to get the r^{th} tuple of relation $\mathcal{R}_i$, the PK is chosen as r and the rest of the attributes come from the summarized relation. We iterate over the rows of $\widetilde{\mathcal{R}_i}$ and keep summing the cardinalities until the summation becomes greater than r. Say the summation crossed the value r in j^{th} row of $\widetilde{\mathcal{R}_i}$. So the rest of the values of the r^{th} tuple are precisely the one that are present in the j^{th} row of $\widetilde{\mathcal{R}_i}$.

6 Projections

In this section, we have proposed an algorithm for handling projections. The solution proposed needs to be optimized in order to make it practical. Let us assume that the constraints that include projections are of the form:

$$|\pi_{\mathbb{A}}(\mathcal{R}_{i_1} \bowtie ... \bowtie \mathcal{R}_{i_p})| = k$$

The joins can be removed by converting them to views as we did before. So now we get constraints of the form:

$$|\pi_{\mathbb{A}}(\mathcal{V}_i)| = k \tag{6.1}$$

Note that since filter predicate will only limit the domain of the attributes and can easily fit in our model, we have omitted them for simplicity.

The algorithm has the following steps:

- Construct a graph G with nodes corresponding to the attributes of the view. We add an edge between nodes of the attributes (A_i, A_j) if there exists a constraint $\mathcal{C}$ of the form (6.1) where $A_i, A_j \in \mathbb{A}$.
- Find all connected components in G.
- For each connected component, we solve an ILP that is formulated according to the algorithm that we present next.

6.1 ILP Formulation For Projections

Let the connected component have the attributes $A_1, A_2, .., A_N$ and r be the number of rows that are to be generated.

For now, we are considering domain for all attribute to be $\{1, 2, ...D\}$. We can reduce the size of the domain of an attribute A_i to the number of distinct values in A_i. Obtaining this value is not difficult. For simplicity we will stick the domain of size D for now.

We first define some notations. Let $\mathcal{R}$ be the relation to be generated and $\mathcal{R}_j$ denotes the j^{th} row in $\mathcal{R}$. Now, let us define an indicator variable $y^j_{k_1,...,k_N}$ as

$$y^j_{k_1,...,k_N} = \begin{cases} 1 & \mathcal{R}_j = (k_1, ..., k_N) \\ 0 & otherwise \end{cases}$$

signifying if j^{th} row is $(k_1, ..., k_N)$. Here $(k_1, ..., k_N) \in \mathbb{D} = Dom(A_1) \times Dom(A_2) ... \times Dom(A_N)$. Further, let us define another indicator variable $\widetilde{Y}_{p_1,...,p_N}$ as

$$\widetilde{Y}_{p_1,...,p_N} = \begin{cases} 1 & \sum_{j=1}^{r} y^j_{p_1,...,p_N} > 0 \\ 0 & otherwise \end{cases}$$

Here, $p_i \in Dom(A_i) \cup \{*\}$. Here $*$ indicates any value from the domain.

Now, for a constraint $\mathcal{C}$ of the form as mentioned in (6.1), let $\mathbb{P}_{\mathcal{C}} = \{p_i, ..., p_N\}$ where p_i = '$*$', if $A_i \notin \mathbb{A}$. The constraint can then be expressed as:

$$\sum_{p_i \forall A_i \in \mathbb{A}} \widetilde{Y}_{\mathbb{P}_{\mathcal{C}}} = k$$

The ILP can be formulated as:

$$max \sum_{\mathcal{C}} \widetilde{Y}_{\mathbb{P}_{\mathcal{C}}}$$

such that, $\forall \mathcal{C}$, we add

$$\sum_{p_i \forall A_i \in \mathbb{A}} \widetilde{Y}_{\mathbb{P}_{\mathcal{C}}} = k \tag{6.2}$$

and

$$0 \leq \widetilde{Y}_{\mathbb{P}_{\mathcal{C}}} \leq 1 \tag{6.3}$$

Also, we add the following two set of equations $\forall j \in [r]$:

$$\sum_{k_i,...,k_n \in \mathbb{D}} y^j_{k_1,...,k_N} = 1 \tag{6.4}$$

and

$$0 \leq y^j_{k_1,...,k_N} \leq 1 \tag{6.5}$$

Finally we add the condition:

$$\widetilde{Y}_{p_1,...,p_N} \leq \sum_{j=1}^{r} y^j_{p_1,...,p_N} \tag{6.6}$$

7 Experiments

For experiments, we took an instance of TPC-DS benchmark database of size 10 GB and a workload of 14 TPC-DS queries that were modified to suit our assumptions. The resultant workload had simple select - join queries that can be written easily in the form as expressed in equation (4.1).

Since the algorithm does not depend on the content of the database, working on TPC-DS is reasonable. Working on realistic databases would add no extra complexity to the algorithm.

The workload of 14 queries was first executed on the original TPC-DS database. We took the corresponding ALQP for each of the queries. We gave these 14 ALQPs along with the database schema as the input to the data generation algorithm. And we got a corresponding synthetic database. We then again executed this workload, this time on synthetic database. We found that the synthetic data satisfied the cardinality constraints except for the small additive errors that were present due to the additional tuples that were added while executing the View Consistency Algorithm. We report these errors in Table 1.

Further, we wanted to check the impact of the optimizations in reducing the number of variables. The effect of domain decomposition is straightforward as it reduced the effective domain sizes of each attribute exponentially. To check the impact of View Decomposition Algorithm, we ran our LP Formulator with and without the optimization. Table 2 shows the number of variables with and without the optimization for the views where we found improvement.

Table 1. Cardinalities of Base Tables

Table Name	Original DB	Synthetic DB
Item	102000	102006
Store	102	103
Catalog_sales	14401261	14401263
Customer_address	250000	250001
Customer_demographics	1920800	1920802
Inventory	133110000	133110000
Warehouse	10	10
Customer	500000	500001
Promotion	500	501
Date_dim	73049	73066
Store_sales	28800991	28800992
Web_sales	7197566	7197559

Table 2. Number of variables

View Name	Original	Optimized
Item	105	15
Store	9	6
Catalog_sales	324	66
Date_dim	864	49
Store_sales	437400	538
Web_sales	12	8

8 Conclusions and Future Work

We considered the problem of *execution-time* testing for database systems. We developed a data generator that is capable of mimicking real customer scenarios for a predefined workload. Our generation ensures that the data stored on the disk is independent of the size of the database. This feature enables us to support big data scenarios as well. Further, our generation algorithm is capable of supplying tuples on demand. Due to this, we adhere to CODD's philosophy of creating a database with no static data.

Currently, our implementation supports only simplified queries that contain only selection and join operators. Our next target is to optimize our algorithm for handling projections and implement it. Operators like *Group By* and *Union* also depend on projection. If we can handle projections efficiently, handling these operators is also straightforward. Also, we propose handling generic schemas and joins as a future work.

9 References

[1] R. S. Trivedi, I. Nilavalagan, and J. R. Haritsa, "CODD: COnstructing dataless databases," DBTest, 2012.

[2] `http://www.tpc.org/tpch/`.

[3] `http://www.tpc.org/tpcds/`.

[4] E. Shen and L. Antova, "Reversing statistics for scalable test databases generation," DBTest, 2013.

[5] N. Reddy and J. R. Haritsa, "Analyzing plan diagrams of database query optimizers," VLDB, 2005.

[6] A. S. and J. R. Haritsa, "CODD: A dataless approach to big data testing," *VLDB Endow., 2015.*

[7] N. Bruno and S. Chaudhuri, "Flexible database generators," VLDB, 2005.

[8] J. E. Hoag and C. W. Thompson, "A parallel general-purpose synthetic data generator," *SIGMOD Rec., 2007.*

[9] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "QAGen: Generating query-aware test databases," SIGMOD, 2007.

[10] E. Lo, C. Binnig, D. Kossmann, M. Tamer Özsu, and W.-K. Hon, "A framework for testing dbms features," *The VLDB Journal, 2010.*

[11] E. Lo, N. Cheng, and W.-K. Hon, "Generating databases for query workloads," *VLDB Endow., 2010.*

[12] E. Lo, N. Cheng, W. W. K. Lin, W.-K. Hon, and B. Choi, "Mybenchmark: generating databases for query workloads," *The VLDB Journal, 2014.*

[13] A. Arasu, R. Kaushik, and J. Li, "Data generation using declarative constraints," SIGMOD, 2011.

[14] A. Arasu, R. Kaushik, and J. Li, "DataSynth: Generating Synthetic Data using Declarative Constraints," *PVLDB, 2011.*

[15] `https://www.gnu.org/software/glpk/`.

[16] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers Inc., 2014.