# On Demand Data Generation at Big Data Scale

Raghav Sood
Computer Science and Automation
Indian Institute of Science Bangalore
raghav@dsl.serc.iisc.ernet.in

July 18, 2016

Mid-term ME Project Report

**Abstract**

OLAP engines are used widely in large data warehousing environments for Business Intelligence and Data Mining. To evaluate and test these engines, we need to have synthetic data with specific characteristics. TPC-DS [3] is an industry standard benchmark for Big Data systems but it represents only a particular data scenario. This limits its ability to capture various realistic customer scenarios. Other challenge in large scale synthetic data generation is the space and time overhead involved in storing and indexing generated data. This makes the traditional techniques of data generation simply impractical at Big Data scale. We have developed an approach to *generate synthetic data* having *custom characteristics*. Also the generation is *on demand* thereby mitigating the space and time overheads.

Our approach takes database schema and data characteristics expressed as set of *cardinality constraints* [4] as input. These constraints can be user specified or can also be inferred from output cardinality of all intermediate operators appearing in a given set of Query Plans. Given these cardinality constraints, our approach can then synthetically generate a *virtual* database instance, which for workload queries, behaves "exactly" like the client's database. The behaviour here is in volumetric terms i.e., getting the same number of output tuples at each intermediate operator in the query plan. This ability to mimick a client's data scenario at the development site will be very useful in database engine testing in particular and data masking in general. The approach also gives the ability to mimick scaledup versions of the client's database making it highly helpful in analyzing futuristic scenarios.

## 1 Introduction

A database engine has broadly two components- the *query optimizer* aka. planner and the *executor*. A query is processed in two phases. Compile-time processing is where given a query the optimizer comes up with an optimal plan. Thereafter, at execution-time, the executor runs the given plan and produces result. Therefore, testing of database engine can be categorized into *compile-time testing* and *execution-time testing*. And effective testing is predicated on the ability to easily construct alternative scenarios with regard to the database contents [6]. Our work is concerned with data generation which can be used for performance testing of the executor.

The already available synthetic benchmarks like TPC-H [2], TPC-DS [3] are commonly used to carry out performance tests in various domains. But a benchmark like these represents only a particular data scenario. The corresponding generators provided by TPC offer an option to generate various scaled versions of the benchmarks, say- 1, 3, 10, 30 or 100 TB for TPC-DS. Whereas extremely limited or no option is offered to change the distribution of the data or manipulate correlation accross columns or relations. This is because of the fundamental reason that allowing such an option defeats the benchmark's purpose of objective and verifiable evaluation of systems. Unfortunately, it makes these benchmarks far from various realistic customer scenarios to which a database vendor serves. However, often these benchmarks are the only possibility, as customer data and workloads are hard to obtain, due to their sensitive nature and moreover they are usually very large which makes transmission over the network hard if not impossible [5]. This may endup with unidentified bugs getting deployed.

Recent work [4] has proposed generating *workload-aware* datasets with the help of constraint solvers. However, this does not scale well to the amounts of data typically present in a customer dataset [5]. Another recent work RSGen [5] is on generating data by *reversing metadata statistics*. This techniques works well in terms of recovering the metadata statistics and in terms of volumetric similarity of COUNT queries on single columns. However, data generated by this technique shows discrepencies for queries which go beyond the simple independent range queries. Thus data generated by RSGen works for analyzing the behavior of query optimizer since it has fundamentally the same metadata characteristics. But this data cannot be used to analyze the performance of executor because execution of a query depends highly on the *fine-grained data characteristics* which are the operator level volumetric outputs at each intermediate stage of a query plan. Refer section 6 for a detailed discussion on related work.

We present a scalable approach to synthetically generate a database instance which has given fine-grained data characteristics. As an input to our approach, the desired data characteristics are specified in a natural, expressive and declarative manner called *cardinality constraints* [4]. These constraints can be user specified or can also be inferred from a given set of Query Plans. The workload for any database client application usually consists of a fixed set of queries. And on an existing client database, running the *explain analyze* command (in case of PostgreSQL or its equivalent in case of other engines) for workload queries gives the output cardinality of all intermediate operators as witnessed during the course of execution of the query. Starting PostgreSQL version 9.0, this command outputs in program readable formats (xml/json/yml) which makes it very easy to capture the fine-grained data characteristics as cardinality constraints. Our approach solves these cardinality constraints along with the referential integrity constraints imposed from the schema. The solution to these constraints is represented as *database summary* which is in order of megabytes while the original database could be in terabytes to petabytes. This database summary is next used as a seed to our on-demand synthetic tuple generator.

We are hence able to synthetically generate a *virtual* database instance, which for workload queries, behaves "exactly" like the client's database. The behaviour here is in volumetric terms i.e., getting the same number of output tuples at each intermediate operator in the query plan.

This ability to mimick a client's data scenario at the database vendor's development site will be very useful for database engine testing. Also the approach serves as a Data Masking technique. Organisations which outsource the testing of their database applications may now share the database summary. The approach also gives the ability to mimick scaledup versions of the client's database making it highly helpful in analyzing performance in futuristic scenarios.

# 2 Preliminaries

## 2.1 Annotated Query Plan (AQP)

A logical query plan is an extended relational algebra tree. It gives the set of operations involved in executing a query as per their sequence. For example, consider the following sql query on TPC-DS schema:

```
select * from
web_sales W, date_dim D, item I
where D.d_month_seq >= 1211
and W.ws_sold_date_sk = D.d_date_sk
and W.ws_item_sk = I.i_item_sk
```

An AQP is a logical query plan which in addition also gives input and output cardinalities for each operator in the tree. The input cardinality specifies the number of tuples that reach the operator and the output cardinality gives the number of tuples that leave the operator.

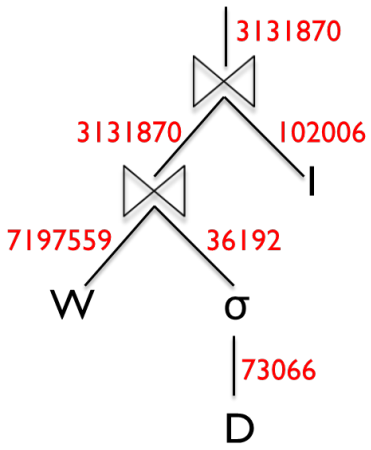


Figure 1: Annotated Query Plan (AQP)

## 2.2 Cardinality Constraints

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

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

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 AQP shown in Figure 2.2 are:

$$|W| = 7197559$$

$$|D| = 73066$$

$$|I| = 102006$$

$$|\sigma_{d\_month\_seq>=1211}(D)| = 36192$$

$$|\sigma_{d\_month\_seq>=1211}(W \bowtie D)| = 3131870$$

$$|\sigma_{d\_month\_seq>=1211}(W \bowtie D \bowtie I)| = 3131870$$

# 3 Problem

## 3.1 Statement

We shall now formally state the problem:

> Given a database schema $\mathcal{S}$ and a set of AQPs $\mathcal{W}$, 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 $\mathcal{W}$.

In the decision version [1] of this problem is *NEXP-complete* [4].

## 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 Data Generation

Our data generator takes the schema of the desired database as the input. Along with it, the set of ALQPs are also given as the input. The AQPs are generated after executing the queries on the original database. These AQPs can be easily fetched from the execution-plan information that the database engines provide. The generator uses these inputs to give the synthetic database (on-the-fly) as the output. We next describe the architecture of the generator.

## 4.1 Architecture

Figure 3.1 provides an overview of the architecture of the data generator. Its various components are:

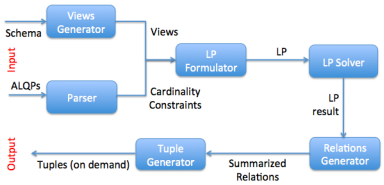


Figure 2: Architecture

- **Parser:** The purpose of parser is to take $\mathcal{W}$ as input and give the set of cardinality constraints as described in Section 2.1.3. Note that for now we are not considering projection operator. Therefore, all the constraints will be of the form:

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

- **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 3.2.1.
- **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 3.2.2.
- **LP Solver:** This component takes the system of equations and gives one of the feasible solutions [2]. We use the GNU Linear Programming Kit (GLPK) [1] for solving the LP.
- **Relation 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 3.2.3 discusses this component in detail.

[1] In the decision version of the problem, the output is YES if there exists a database instance that satisfies all the constraints and NO, otherwise

[2] 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.

- **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 3.2.4.

## 4.2 Details

### 4.2.1 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 3.2.

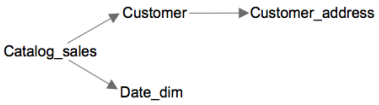


Figure 3: Dependency Graph

By traversing the dependency graph in reverse topological sort order and a view corresponding to each relation is generated.

$$Catalog\_sales'(ca\_state, d\_qoy, d\_year, cs\_sales\_price)$$
$$Date\_dim'(d\_qoy, d\_year)$$
$$Customer'(ca\_state)$$
$$Customer\_address'(ca\_state)$$

### 4.2.2 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: $\mathcal{R}_i \bowtie \mathcal{R}_k$. 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 $\mathcal{R}_k$. In such a case, we shall replace the expression $\mathcal{R}_i \bowtie \mathcal{R}_k$ 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 $\mathcal{R}_i \bowtie \mathcal{R}_k$ to $\mathcal{V}_i$. Likewise, all the join expressions can be expressed in terms of a single view respectively.

Continuing on the example as in Figure 2.2, considering that the dependency graph is rooted at node $W$ having two children $D$ and $I$, the cardinality constraints 2.5 and 2.6 can be rewritten as:

$$|\sigma_{d\_month\_seq>=1211}(W')| = 3131870$$
$$|\sigma_{d\_month\_seq>=1211}(W')| = 3131870$$

where $W'$ represents the view corresponding to $W$.

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:

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

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 domains, 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 $x_t \geq 0 \; \forall t$ as the number of tuples are always non-negative integers. Since, the solution of the LP need not be integral, we shall use simple *rounding technique*.

Further, 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.

#### 4.2.3 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:

$$x_{[1,151)[1,2)} + x_{[1,151)[2,)} + x_{[151,)[1,2)} + x_{[151,)[2,)} = 14401261$$

$$x_{[151,)[1,2)} + x_{[151,)[2,)} = 734606$$

$$x_{[151,)[1,2)} = 13806$$

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.

### 4.3 Relation Generator

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

$$\{(x, k^i_x)\}_{x \in Dom(\mathcal{A}_{c_i})}$$

where $x$ represents an interval combination and $k^i_x$ represents the number of tuples in the view that lie in the interval combination $x$. Note that $x$ constitutes of intervals of those attributes that are present in $\mathcal{A}_{c_i}$.

Since, the above solution has intervals for a set of attributes, to get a value combination from this, we need to pick a value from each interval. We pick the minimum value in every interval to get the value combination. We do this to minimize the error that is incurred in the view consistency algorithm (we shall discuss this in Section 3.2.3.2) [3]. Now on we shall use $x$ to represent the value combination thus obtained.

Let us call these solution sets as the *clique-solution sets*. 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 $k^i_x$ is non-zero since we do not need to store the domain values that do not occur from our LP solution.

The relation 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 relation summaries.

[3] any deterministic method of picking a value in an interval can be adopted to reduce the error

#### 4.3.1 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). This merge ordering is necessary in order to retain all the constraints that each clique solution implies. In every iteration of ordering algorithm, we look for a clique $c_i$ that is independent of the observed cliques so far except for the attributes that are common between $c_i$ and the observed cliques. Since the graph is chordal, we are guaranteed to get at least one such ordering.

Once we get the ordering, we then merge the cliques-solution sets one by one in that order. Merging is a three step process. We first sort the clique-solution sets based on their common attributes, then use the *Align Procedure* (Algorithm 4) to split the rows in the two clique-solution sets in such a way that the corresponding rows in the two sets have same values for the number of tuples entry. Finally we join the two sets using the *Merge Procedure* (Algorithm 5).

*Example:* Consider a case where we need to merge two clique-solution sets to obtain a view solution set. Let the two clique-solution sets be as follows:

| Clique-solution 1 | | |
|---|---|---|
| A | B | number of tuples |
| 10 | 20 | 100 |
| 30 | 50 | 60 |
| 10 | 50 | 40 |
| 30 | 20 | 200 |

| Clique-solution 2 | | |
|---|---|---|
| B | C | number of tuples |
| 20 | 5 | 150 |
| 50 | 15 | 100 |
| 20 | 15 | 150 |

After sorting on common attribute attribute B, we would obtain:

| Clique-solution 1 | | |
|---|---|---|
| A | B | number of tuples |
| 10 | 20 | 100 |
| 30 | 20 | 200 |
| 10 | 50 | 40 |
| 30 | 50 | 60 |

| Clique-solution 2 | | |
|---|---|---|
| B | C | number of tuples |
| 20 | 5 | 150 |
| 20 | 15 | 150 |
| 50 | 15 | 100 |

After aligning the two sets, we obtain:

| Clique-solution 1 | | |
|---|---|---|
| A | B | number of tuples |
| 10 | 20 | 100 |
| 30 | 20 | 50 |
| 30 | 20 | 150 |
| 10 | 50 | 40 |
| 30 | 50 | 60 |

| Clique-solution 2 | | |
|---|---|---|
| B | C | number of tuples |
| 20 | 5 | 100 |
| 20 | 5 | 50 |
| 20 | 15 | 150 |
| 50 | 15 | 40 |
| 50 | 15 | 60 |

Finally, after merging, the view solution set thus obtained is:

| View solution | | | |
|---|---|---|---|
| A | B | C | number of tuples |
| 10 | 20 | 5 | 100 |
| 30 | 20 | 5 | 50 |
| 30 | 20 | 15 | 150 |
| 10 | 50 | 15 | 40 |
| 30 | 50 | 15 | 60 |

# 5 2nd SECTION TITLE

# 6 Conclusions and Future Work

Proin id elit nec erat pellentesque gravida ut a lectus. Proin rhoncus eu justo et aliquet. Praesent auctor augue quis magna dapibus imperdiet. Fusce vehicula vehicula aliquet. Fusce ac dolor in lorem tristique bibendum eget finibus turpis. In odio dui, mattis sed commodo non, bibendum et augue. Nam dignissim id metus ut bibendum. Nulla eu leo sed dui venenatis viverra. Donec et enim rutrum lorem aliquam bibendum ac ac arcu. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In gravida enim vitae nisi tincidunt pulvinar. Maecenas hendrerit lacinia dui, et egestas turpis suscipit sed. Vestibulum eros orci, bibendum in diam in, suscipit aliquet ipsum. Nulla facilisi. Vestibulum vulputate, nulla ut ornare varius, eros leo elementum sem, at tempor ante massa vitae dolor.

# References


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

[2] TPC-H v2.17.1. `http://www.tpc.org/tpch`, 2014.

[3] TPC-DS v2.3.0. `http://www.tpc.org/tpcds`, 2016.

[4] Arvind Arasu, Raghav Kaushik, and Jian Li. Data generation using declarative constraints. SIGMOD, 2011.

[5] Entong Shen and Lyublena Antova. Reversing statistics for scalable test databases generation. DBTest, 2013.

[6] Rakshit S. Trivedi, I. Nilavalagan, and Jayant R. Haritsa. CODD: COnstructing dataless databases. DBTest, 2012.