January 10, 2019

A novel encoding & alignment of Histograms of referential integrity columns for scalable data generation

Suresh Soundararajan
Hewlett Packard Enterprise

A novel encoding & alignment of Histograms of referential integrity columns for scalable data generation

Abstract

Testing the performance of database management systems is often accomplished using synthetic data and workload generators such as TPCH and TPCC. However most synthetic benchmarks don't fully match customer database configurations. Customer database configuration data-sets are typically hard to obtain due to their sensitive nature and prohibitively very large sizes. As a result, oftentimes the data management systems are not thoroughly tested, and performance related bugs are commonly discovered after deployment, where the cost of fixing is very high. We propose a scalable data generator called **XGen**, an approach to generating data-sets out of customer metadata information, including integrity constraints and histogram statistics. Handling multiple referential integrity constraints is a very hard problem and we handle it in a very novel way by indirectly encoding the column dependencies so that we can still independently generate the column data for scalable data generation.

Problem statement

Referential integrity presents a very challenging problem for scalable data generation for very large data-sets. This is because of the dependency of the foreign key columns on the primary key column. We cannot independantly generate values for the foreign key(FK) columns as these values have to exist on the primary key(PK) column. This slows down the data generation if we have to check what we have generated for the PK and then generate a subset of them for the multiple FKs that are dependent on this PK.

Our solution

A novel method of encoding and aligning the histograms of referential integrity columns is presented here to enable independent and scalable data generation of very large columns involving multiple foreign keys(FKs) on the same primary key(PK), without compromising on the quality of data generation wrt to the histograms of the column.

We studied literature on how others have approached the problem. Quality of generated data (how closely it can match real world data) was one of the biggest issues. The reason for this is that there are *subtle correlations between different database columns* and it's difficult to factor this accurately during data generation. The word Correlation is made of Co- (meaning "together"), and Relation. When two sets of data are strongly linked together we say they have a High Correlation. Correlation is Positive when the values increase together, and Correlation is Negative when one value decreases as the other increases. The second issue was speed of generation and scalability when we need to handle very large data-sets. We have addressed both these issues to a very great extent.

There are multiple ways in which correlations are represented within a database schema. To start with the simple check constraints (Key/Unique column, not NULL, values within a range) which are properties of data within a single column and then referential integrity (RI) constraints, which are across columns and these are *explicitly identified* by the user while creating the metadata (read DDLs).

The second key issue of scalability can be addressed if we can *generate each column independently, even though there are dependencies among*

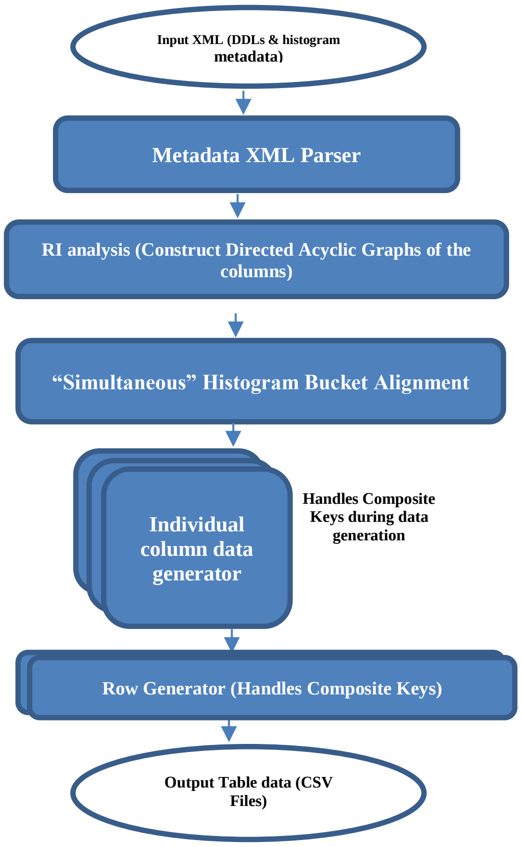
Input XML (DDLs & histogram metadata)
Metadata XML Parser
RI analysis (Construct Directed Acyclic Graphs of the columns)
"Simultaneous" Histogram Bucket Alignment
Individual column data generator
Handles Composite Keys during data generation
Row Generator (Handles Composite Keys)
Output Table data (CSV Files)

Figure 1: Xgen Architecture

them. This would enable two levels of parallelism during data generation – one at the column level and then one at the table level which would make the solution scalable for very large data-sets. We need to smartly *encode* the explicitly specified inter-column dependency (RI) between columns – *in other words each foreign key column should somehow apriori know what are the data values going to be chosen for the primary key column*. We found a very innovative way of doing this by Shen & Antova [1] by aligning histogram boundaries of related columns which *we leveraged and extended it in a novel way to handle multiple RIs on the same key*. Multiple RIs on the same key is very common in production data and handling it is important.

We process the input XML that has both DDL and histogram metadata of a given HPE's NonStop® SQL/MX® database and store in our internal data structure vectors. We then construct directed acyclic graphs (DAGs) whose nodes corresponds to columns and the edges represent the referential key constraints among the columns. Each DAG models one key (PK) column that is used as a foreign key (FK) in one or more tables. We could have a forest of DAGs depending on the number of different PKs that are used as FKs and we topologically sort each of the DAGs to get the primary key column on the top and then do *simultaneous* histogram bucket alignment of each column that is part of the DAG, which is a process of creating new *common boundaries (dotted lines) across all the columns in the DAG (Fig 2).* In this process we typically generate new intervals and need to push them in the correct place in the column's existing histogram and also need to re-distribute the interval distinct counts(UEC) & row-counts by looking at the immediate neighbors. By doing so we are ensuring that when we chose values from *some fixed points* in the aligned boundary as the column values for the PK, we can also chose the *same values for the column values of the FKs* as well, thereby ensuring *independence of column data* generation, which helps us to do parallel column data generation that aids the scalability for large data-sets. A crucial piece of information for doing this correctly across different FKs is to know the interval unique/distinct counts(UEC) of the PK which is stored along with the FK interval's metadata and that is used to sub-divide the histogram boundary into fixed chunks (sub-intervals). Out of the sub-intervals, we consider for each column, its *interval unique count* number of values as the values generated for that column. In the example here, for the boundary starting at value 190 and ending at 400, PK uniques count is 7, so the values chosen for the PK would be 400 – 190 = 210, divided by 7 would make it fixed points separated by 30, so we chose 190, 220, 250, 280, 310, 340, 370 for the PK. For the corresponding interval in the FK, the values chosen would be the number of uniques of each FK. So for FK1, it would be 190, 220, 250 as its uniques count is 3 & for FK3 it would be 190, 220, 250, 280. Thus we can independently generate values of each FK in this novel way by encoding the dependency. We do this for all integral data types and datatypes that can be converted to an integral value like DATE and fixed point Numeric. For strings and floating point types, a slight variation was needed to handle them.

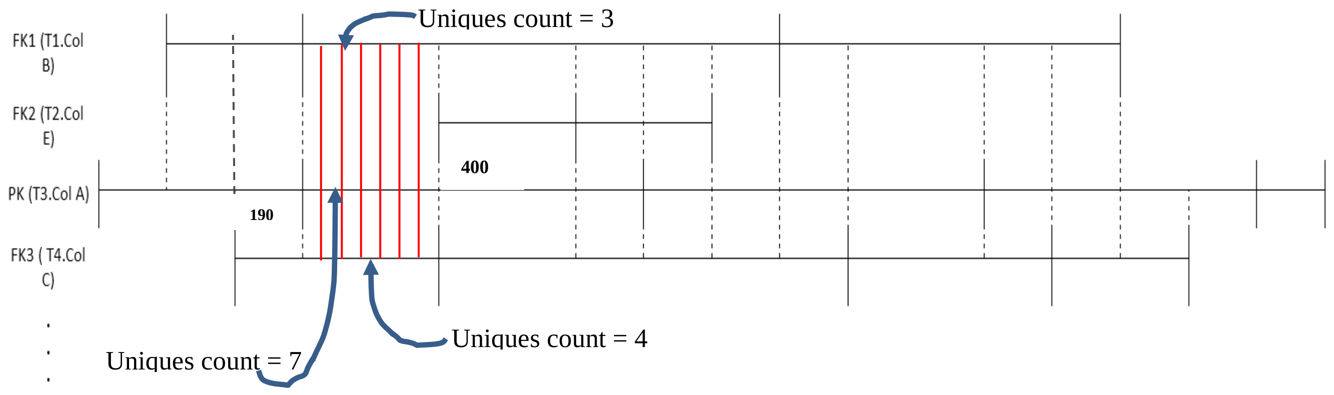

Figure 2: *Simultaneous Alignment* of Histogram interval boundaries for a DAG of 4 nodes

Finally while forming the tuples of the table using the individual columns that were generated in the previous phase, we can form them parallelly for different tables resulting in scaling the solution across multiple tables.

Evidence the solution works

>>Statistics for XGen generated TPCH table:

```
Histogram data for Table
SURESH_DATAGEN.XGEN_TPCH2X.SUPPLIER
Table ID: 3932353710815286816

   Hist ID # Ints    Rowcount        UEC Colname(s)
========= ====== =========== ===========
==========================
1923909252    48       20000       20000 S_SUPPKEY
1923909257    62       20000       19999 S_NAME
1923909262    62       20000       20000 S_ADDRESS
1923909267    25       20000          25 S_NATIONKEY
1923909272    62       20000       20000 S_PHONE
1923909277    56       20000       19803 S_ACCTBAL
1923909282    62       20000       19972 S_COMMENT
1923909287     1       20000       20000 S_SUPPKEY, S_NATIONKEY
```

>>Statistics for Base TPCH table:

```
Histogram data for Table SURESH_DATAGEN.TPCH2X.SUPPLIER
Table ID: 3932345418723699427

   Hist ID # Ints    Rowcount        UEC Colname(s)
========= ====== =========== ===========
==========================
2055324022    48       20000       20000 S_SUPPKEY
2055324029    62       20000       20000 S_NAME
2055324032    62       20000       20000 S_ADDRESS
2055324039    25       20000          25 S_NATIONKEY
2055324042    62       20000       20000 S_PHONE
2055324049    56       20000       19803 S_ACCTBAL
2055324052    62       20000       19972 S_COMMENT
2055324059     1       20000       20000 S_SUPPKEY, S_NATIONKEY
```

Figure 3: No of Distincts (aka UEC) between

We have taken the TPCH benchmark that has a built-in data generator and compared it with the database generated with our XGen for 2X scale factor database. We have done the following experiments to verify the quality of generation. Simple statistical measures like No of distincts in both data-sets are matching. For supplier table (Fig 3), the 4th column in the table shows the No of distincts count for both the XGen generated table & the original table. Also we have elaborate count (*) queries with different column selectivities (10 to 100 %) to see the column data distribution for the various table columns were matching. We also run the actual TPCH queries and see that the query plan shapes are comparable. More experiments are underway.

Competitive approaches

There have been two main lines of work in the literature for test database generation. The first one focuses on generation of large synthetic databases, subject to user-provided closed-form column distribution. A major issue here is that the generated database tends to give empty result over complex queries, as subtle correlations between attributes are often not captured. The second approach addresses the problem by considering a set of constraints, say for e.g., given a workload of queries & generating a database such that each intermediate result has a certain size and the use of constraint solvers is very common in this. Most of these data generators require excessive efforts from the user to learn the descriptive language and accurately specify data dependencies and distributions. The approach taken by Shen & Antova [1] comes very close to the approach we have taken. But we have improved significantly on their approach and added a novel way of handling multiple RI constraints, which we believe is not handled by [1] and which makes our approach more usable for real-world customer databases.

Current status

The first version of this tool exists and is currently being used by the SQL/MX engineering team to test and find performance issues. More experiments are underway and we are fine tuning the generation. Also experiments are being done on customer metadata to reproduce performance issues.

Next steps

A lot enhancements are being planned. Today NonStop SQL/MX has only very minimal multi-column histogram support (only full column summaries and no histogram interval level details) and there are plans by the team to extend it in the near future. We have plans to extend XGen to factor in the multi-column statistics to enhance the quality of our data generation even more.

References

[1] Shen & Antova - DBtest'13 - Reversing Statistics for Scalable Test Databases Generation
[2] J. Gray, P. Sundaresan, et al. Quickly generating billion-record synthetic databases. In proceedings of the 1994 ACM SIGMOD international conference on Management of data, pages 243–252, 1994.
[3] A. Arasu, R. Kaushik, and J. Li. Data generation using declarative constraints. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pages 685–696, 2011.