Big Data Testing Environments

Ankur Gupta
ankur.gupta@csa.iisc.ernet.in
SR No: 04-04-00-10-41-12-1-09278
Department of Computer Science and Automation
Indian Institute of Science, Bangalore
Final Report of ME dissertation (2013-2014)

Abstract

It is the era of big data. Over the last decade, the quantity of data has been exploding in an unprecedented manner. This trend is likely to continue even in the future[1]. To handle this massive amount of data, which is ever increasing, the database engines should be robust. An inadequately tested database engine can wreak havoc for the organization using it. In order for them to be robust for futuristic databases, we need to test them on various alternative futuristic big data scenarios. However, to generate and store such a humongous database is infeasible due to time and space constraints. The performance of a database engine depends not only on the alternative database scenarios, but also on the hardware available to execute the queries on those scenarios. Therefore, we also need some mechanism to test the database engine on alternative hardware configurations, to make the engine robust for futuristic hardware configurations. All this calls for a testing tool which can simulate such environments virtually on a low end machine by generating the data at query execution time, instead of physically storing it. In this work, we worked on engineering CODD[2] to simulate such environments for COM_OPT - an industrial strength distributed database system, which supports big data. Specifically, we have engineered CODD for (a) simulating different metadata configurations on COM_OPT for query compilation time testing, (b) simulating different hardware configurations on COM_OPT for hardware based testing and (c) synthetic database generation for query execution time testing.

1 Introduction

Database engine testing is a crucial activity during the development of database systems. The performance of database systems usually depends on various factors, like generation of good query execution plans, availability of different kinds of access paths and resources available at the server. There are various benchmark databases, like TPC-H[3] and TPC-DS[4], which are used for the performance testing of database systems. In the era of big data, the quantity of data has been exploding in an unprecedented manner. This trend is likely to continue even in the future[1]. To handle this massive amount of data, which is ever increasing, the database engines should be robust. An inadequately tested database engine can wreak havoc for the organization using it. Today, the testing of database engines is constrained by the resources available to generate and store the test databases with desired scenarios, which has increased the need of finding alternative ways to test the database engine for big data environments manifold. In this work, we have focused on three different aspects of big data testing for HP NonStop SQL database, which is a relational database engine, designed for a clustered and massively-parallel system architecture. Now onwards, we will refer this database engine as COM_OPT.

i. Query Compilation Time Testing. There are various modules in a database system, such as query optimizer, which depend only on the metadata of actual data at query compilation time. The testing of these modules does not require the presence of actual data in the database system. To test these modules, a software CODD[2], along with the software Picasso[5] is used, both of which are developed in our lab. CODD is a tool, which supports the construction of alternative database scenarios without storing the actual database contents, while Picasso is a tool, which is used to analyze the behavior of query optimizers, using various kind of diagrams, at query compilation time as well as query execution time. Earlier, CODD was functional for IBM DB2, MS SQL Server, Oracle and PostgreSQL database systems. This part of our work focuses on engineering CODD for COM_OPT to al-

low query compilation time testing of a futuristic big data system using the metadata of actual data. In this work, we were able to simulate 1-Zettabyte(10^{21}Bytes) TPC-H database on COM_OPT, which is currently infeasible even on high-end systems, and we found that 1-Zettabyte is the maximum limit supported by COM_OPT for a scaled version of TPC-H database. We also found a bug in the cardinality estimation module of COM_OPT, where it had exaggerated the value of cardinality to orders of magnitude of the maximum possible cardinality.

ii. Hardware Based Testing. CODD supports creation of alternative database scenarios, but it currently does not support the creation of alternative hardware scenarios. The choice of the optimal query execution plans depends not only on the metadata available in the system, but also on the hardware availability. Creating alternative hardware configuration scenarios is useful for testing the functionality of query optimizer on different kinds of hardware configurations, which allows us to make the database engine robust for futuristic hardware scenarios. In this part of our work, we added a new feature in CODD, which allows us to simulate different kinds of hardware configurations to analyze the behavior of COM_OPT on futuristic hardware scenarios.

iii. Query Execution Time Testing. The compile time testing of database engines can be done using metadata statistics, but for execution time testing, the actual data needs to be present in the system. Since generating and storing alternative big data scenarios is infeasible even on high end systems, because of time and space constraints, therefore we need a way to do execution time testing without storing the database contents. This can be made possible, if we do on-the-fly generation of data whenever a query gets executed. The first step in on-the-fly data generation is to generate a synthetic database, which has similar data distribution as the actual database. This part of our work focuses on generating a synthetic database having data distributions similar to the actual database, for execution time testing using the metadata statistics calculated from the actual database.

The rest of this report is organized as follows. We start with the background of CODD Metadata Processor in Section 2. Section 3 tells about the new features added to CODD, which are *metadata configuration for COM_OPT*, *hardware configuration for COM_OPT* and *execution time testing*. In Section 4, 5 and 6, we discuss each one of these in detail. Section 7 gives the details of the coding and other efforts required in this work. Section 8 contains the literature survey related to the *synthetic database generation* for *execution time testing*. Finally we conclude the report with future work in Section 9.

2 Background on CODD Metadata Processor

CODD[2] is a Java based graphical tool, developed in our lab, which is used to create *dataless databases* in database systems. The database engines are usually tested on alternative database scenarios to make them robust with respect to various kinds of data distributions as well as different database sizes. Creation of alternative database scenarios involves a lot of time and space overheads in generating and storing those databases, which makes it difficult to test the database engine on those database scenarios. There are various components in a database system, such as query optimizer, which do not depend directly on data, but on the metadata. CODD makes the testing of those components possible, by providing a graphical user interface through which databases with the desired metadata characteristics can be simulated by constructing the metadata of those databases. For construction of different kinds of metadata scenarios, CODD provides the following modes of operation:

2.1 Metadata Construction

Metadata Construction Mode of CODD allows the users to simulate desired database scenarios by providing a graphical user interface, which is used to construct a *dataless database* using the metadata of that database. CODD uses the existing techniques provided by the commercial database engines for manually updating the metadata statistics. To create the *dataless database*, first we update the relation level metadata statistics, followed by the attribute and index level metadata statistics. Attribute level metadata statistics also includes the histograms for which CODD supports two kinds of interfaces, *manual* and *graphical*. In manual interface, the user is supposed to write all the histogram information manually, while the graphical interface allows the users to create the desired histogram using mouse.

Construct Mode uses a directed acyclic graph based validation algorithm to ensure that the input metadata values are both *legal* (valid range, correct type) and *consistent* (compatible with other metadata values). After validation of input metadata statistics at relation, attribute, and index level, the metadata statistics are written in database catalogs to complete

the construction of metadata. Since this mode works only on metadata, the construction process is very fast and it takes very less time and space even for a big database.

2.2 Metadata Scaling

The testing of database engines is also done on different scaled versions of the original database, to analyze the behavior of query optimizer on different sizes of same database as well as the futuristic big data version of the database. CODD supports two kinds of scaling methods, which scale the metadata of original database to simulate the scaled database.

2.2.1 Size-based Scaling

Given an initial metadata shell and a scaling factor α, as input, it produces a scaled metadata shell such that the size of the database represented by the scaled metadata shell, is α times of the size of the database represented by the initial metadata shell.

2.2.2 Cost-based Scaling

Given a query workload, an initial metadata shell, and a scaling factor α, as input, it produces a scaled metadata shell such that the optimizer's estimated cost of executing the query workload on scaled metadata shell is scaled by the scale factor α.

3 New Features Added To CODD Metadata Processor

Earlier, CODD had the support to simulate *dataless databases* for compile time testing on various *commercial centralized database engines*, but it was not supported on COM_OPT. It was also lacking the support for *hardware based testing* and *execution time testing*, which is useful for the testing of database engines on futuristic big data scenarios. In this work, CODD has been engineered to make the big data testing possible on COM_OPT, and the following three features have been added in CODD for that purpose:

- **Metadata Configuration**
- **Hardware Configuration**
- **Execution Time Testing**

Sections 4, 5 and 6 discuss each one of these in details, along with the experimental results.

4 Metadata Configuration For Compile Time Testing

In [6], CODD was engineered for COM_OPT and it was just a *pilot project*, developed on the older NT version of COM_OPT. This *pilot project* had various functionalities, which were either missing or not correctly implemented. In this work, we worked closely with COM_OPT developers to resolve all those issues, made lots of improvements in the functionality, and tested it on the latest version of COM_OPT. Now this software has been brought from a pilot version to a production level software, and currently it has been deployed in industry for the big data testing of COM_OPT. The rest of this section discusses about the various modes of CODD for COM_OPT, and the improvements made in them.

4.1 Metadata Construction

Metadata Construction mode of CODD allows the users to create a metadata shell for the given input relations, without requiring the presence of data in those relations. The input to the *construct mode* are the empty relations, which are grouped under a schema. Construct Mode GUI for COM_OPT requires the initial metadata statistics to be present for all the relations selected by user for metadata construction, and then it updates those initial metadata values with user provided values. The initial metadata statistics for empty relations are created using the following command for each input relation:

UPDATE STATISTICS FOR TABLE <TABLENAME> ON EVERY COLUMN

The above command creates an entry for the input relation and all of its attributes in *histograms* and *histogram_intervals* tables, shown in Figures 1 and 2. The user is supposed to update these statistics manually before starting the metadata construction process. If these statistics are missing, the user is informed about it by the construct mode interface.

COM_OPT supports the following three kinds of metadata statistics for each relation:

- *Relation Statistics*,
- *Attribute Statistics*,
- *Index Statistics*.

The details about all these three kinds of statistics are given below.

4.1.1 Relation Statistics

Relation level metadata statistics for COM_OPT include the following:

- *cardinality* (Number of records in the relation),
- *record_size* (Number of bytes in each record),
- *block_size* (Number of bytes in each disk block), etc.

Apart from the *cardinality* of the relation, all other values are stored in system metadata tables, which can't be modified by the user. Therefore, metadata construction mode allows the user to update only the value of relation cardinality. Once the value of relation cardinality has been updated in construct mode user interface, it can't be modified again (to keep it consistent with the metadata of all attributes of the given relation). The *rowcount* attribute of the *histograms* table, see Figure 1, is used to store the value of relation cardinality.

Column Number	Column Name	Data Type	Description
1	TABLE_UID	LARGEINT	The UID of the table for which this histogram is generated
2	HISTOGRAM_ID	INT (UNSIGNED)	System-generated ID for the histogram. Each HISTOGRAM_ID has a corresponding ID in the HISTOGRAM_INTERVALS table
3	COL_POSITION	INT	Column position in a column group for which the histogram is generated
4	COLUMN_NUMBER	INT	Table column number for which this histogram is generated.
5	COLCOUNT	INT	Number of columns in the column group
6	INTERVAL_COUNT	SMALLINT	Number of intervals in the histogram. If the value is *n*, then there are *n+1* corresponding rows in the HISTOGRAM_INTERVALS table with the same HISTOGRAM_ID
7	ROWCOUNT	LARGEINT	Total number of rows in the table
8	TOTAL_UEC	LARGEINT	Total number of unique entries in the table
9	LOW_VALUE	VARCHAR (250)	Low value of column distribution (for the entire table)
10	HIGH_VALUE	VARCHAR (250)	High value of column distribution (for the entire table)

Figure 1: HISTOGRAMS Table

4.1.2 Attribute Statistics

Once the relation level metadata has been updated, the attribute level metadata statistics are updated for each attribute of that relation. Attribute level metadata statistics are created for each column as well as for the group of columns (in case of multi-column attributes). Attribute level metadata statistics for COM_OPT are stored in *histograms* table, see Figure 1, and include the following:

- *total_uec* (Number of distinct values),
- *high_value* (Highest value in the column for single column attributes / Highest value from each column separated by comma for multi-column attributes),
- *low_value* (Lowest value in the column for single column attributes / Lowest value from each column separated by comma for multi-column attributes).

Column Number	Column Name	Data Type	Description
1	TABLE_UID	LARGEINT	The UID of the table for which this histogram is generated
2	HISTOGRAM_ID	INT (UNSIGNED)	System-generated ID for the histogram. Each HISTOGRAM_ID has a corresponding ID in the HISTOGRAMS
3	INTERVAL_NUMBER	SMALLINT	Sequence number for this interval
4	INTERVAL_ROWCOUNT	LARGEINT	Number of rows in this interval
5	INTERVAL_UEC	LARGEINT	Number of unique entries in this interval
6	INTERVAL_BOUNDARY	VARCHAR (125)	The value of the upper boundary for this interval. The boundary for this interval is defined to be the INTERVAL_BOUNDARY of the previous row (except for the first interval).

Figure 2: HISTOGRAM_INTERVALS Table

After updating the attribute statistics, attribute histogram related metadata statistics are updated. COM_OPT uses equi-width histograms to represent the data distributions of each column. The number of buckets present in the histogram is stored in *interval_count* column of *histograms* table, see Figure 1, while individual bucket related information is stored in *histogram_intervals* table, see Figure 2. In COM_OPT, if the histogram contains N number of buckets, then N+1 buckets are stored in *histogram_intervals* tables, where first bucket is the default bucket, which contains *low_value* in place of *interval_boundary*, and zero in place of *interval_rowcount* and *interval_uec*. In CODD, the default bucket is created internally without the knowledge of the user, since it don't need

any extra input from the user. For multi-column attributes, COM_OPT creates only two buckets, where *interval_boundary* contains *low_value* and *high_value* for first and second buckets respectively. Since it contains only two buckets and there is no need of any extra input from the user, therefore the histogram editing is disabled in user interface for multi-column attributes, and two buckets, as discussed above, are created internally without the knowledge of the user.

Construct mode interface allows the user to input histogram statistics either manually or from a file. Whenever user updates the metadata for an attribute, the attribute level as well as histogram level metadata statistics are written in a file, which can be used later to read this information while editing the metadata for the same attribute again. In this work, the functionality to create new buckets and delete existing buckets in the manual histogram editing interface has been improved significantly. The format of histogram file has also been improved, and now it contains the additional information about the histogram, which was missing earlier.

CODD also provides graphical user interface to edit the histograms so that the user can create the desired histogram using mouse. Earlier, the user was supposed to make few entries in the manual histogram editing interface, before using the graphical histogram editing interface. Now this restriction has been removed and the user can start using graphical histogram editing interface, by just entering the information about the *total_uec*, *low_value* and *high_value*.

To finally write all the metadata information in database catalogs, COM_OPT supports direct INSERT and UPDATE commands over *histograms* and *histogram_intervals* table. In current release, the functionality to write the metadata information in the database catalogs, has been rewritten, since earlier functionality was not correct. The following are the steps to update the metadata information in *histograms* and *histogram_intervals* tables, which are now part of the current release:

- First we clear all the statistics for the input relation using the following command:

  UPDATE STATISTICS FOR TABLE <TABLENAME> ON EVERY COLUMN CLEAR;

- Then, we create the default statistics for the input relation using the following command:

  UPDATE STATISTICS FOR TABLE <TABLENAME> ON EVERY COLUMN;

- Once the default statistics are created for the input relation, the relation level and attribute level statistics are updated in *histograms* table using the SQL UPDATE command.

- The UPDATE STATISTICS command creates two default buckets in *histogram_intervals* table for each attribute histogram, where the *interval_boundary* value is empty, while other values are zero. To write the histogram related information, these two buckets are updated using the SQL UPDATE commands, while the extra buckets are added using SQL INSERT commands.

One more major change has been made in the functionality of CODD for COM_OPT. Earlier, after the metadata statistics were updated for an attribute, that attribute used to be removed from the drop down menu. In this functionality, it was not possible for the user to update the metadata for that attribute again. In case of an error, the user used to be redirected to the database connection interface and all the information written in Construct Mode interface were lost. This functionality has been improved now, so that the user can update the metadata multiple times before writing it in the database catalogs.

4.1.3 Index Statistics

COM_OPT also supports the index level metadata statistics, which are stored in system metadata tables and cannot be modified by users. Therefore, index level metadata statistics were excluded from the construct mode interface for COM_OPT.

4.2 Metadata Validation

CODD incorporates metadata validation mechanism in Construct Mode to ensure that the input metadata values are both *legal* (valid range, correct type) and *consistent* (compatible with other metadata values). Whenever the user updates the relation level or attribute level metadata, the input metadata values are validated, and if there is any validation error, an appropriate error message is shown to the user. The metadata values are validated for two kinds of constraints:

i. Structural Constraints or Legality Constraints: The input metadata value should be of *correct type* and it must be in the *valid range* of values for that type. For example, *cardinality* for a relation must be of positive integer type and the value must be less than or equal to $2^{63} - 1$.

ii. Consistency Constraints: Input metadata value should be compatible with other metadata values. For example, *total_uec* in a column must be less than or equal to the *cardinality* of the relation.

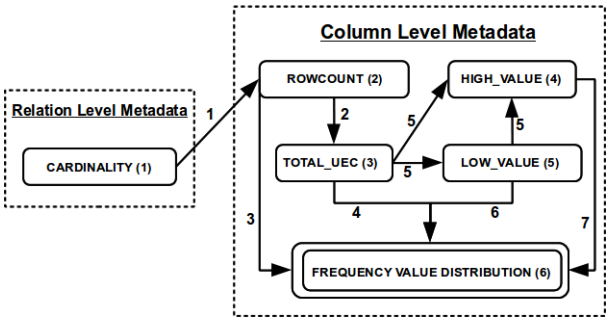

Figure 3: Metadata Constraint Graph

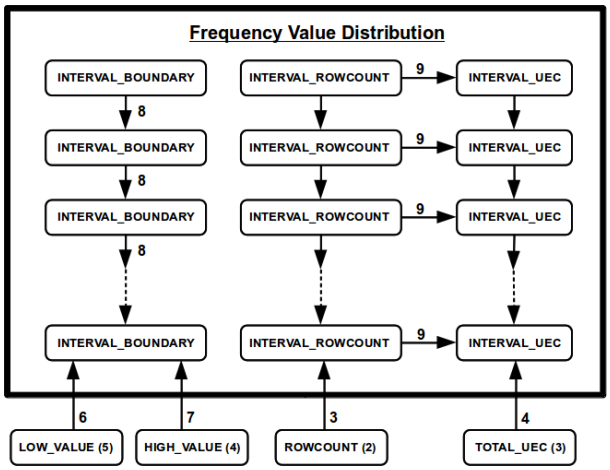

Figure 4: Constraint Graph For Distributions

To validate the input metadata in CODD, we first construct a directed acyclic constraint graph G(V,E) from all the metadata entities. In this graph G, the set of nodes V represents the structural/legality constraints, while the set of edges E represents the consistency constraints, as discussed in [2]. The directed acyclic constraint graph G for COM_OPT is shown in Figures 3 and 4. The numbers present on the nodes of acyclic constraint graph G represent the structural constraints, see Figure 5, while the numbers present on the edges represent the consistency constraints, see Figure 6. Once we have the directed acyclic constraint graph G, we run a topological sort on G, which provides a linear ordering of the nodes. This linear ordering of nodes is used to validate the input metadata for structural and consistency constraints. If any of the above type of constraint is not satisfied, then the user gets an error message containing the type of constraint and the details about it. When the user inputs all the valid metadata, the metadata validation becomes successful and the metadata is updated into the database catalogs to complete the metadata construction process.

Constraint	Description
1	The cardinality of the relation (CARDINALITY) should be of integer type and it's value should be greater than Zero and less than or equal to 2^{63}-1.
2	The cardinality of the column (ROWCOUNT) should be of integer type and it's value should be greater than Zero and less than or equal to 2^{63}-1.
3	The number of distinct values in a column (TOTAL_UEC) should be of integer type and it's value should be greater than Zero and less than or equal to 2^{63}-1.
4	The data type of the value present in HIGH_VALUE should be same as the data type of the corresponding column and the value should be in the range of the data type of the column.
5	The data type of the value present in LOW_VALUE should be same as the data type of the corresponding column and the value should be in the range of the data type of the column.
6	The values of INTERVAL_ROWCOUNT and INTERVAL_UEC should be of integer type and their values should be greater than or equal to Zero and less than or equal to 2^{63}-1. The data type of the value present in INTERVAL_BOUNDARY should be same as the data type of the corresponding column.

Figure 5: Structural Constraints

Constraint	Description
1	The cardinality of column (ROWCOUNT) should be equal to the cardinality of its corresponding table (CARDINALITY).
2	The number of distinct values in a column (TOTAL_UEC) cannot be greater than the cardinality of the column (ROWCOUNT). TOTAL_UEC should be equal to ROWCOUNT, if there is a UNIQUE constraint on the column.
3	The sum of the values in INTERVAL_ROWCOUNT should be less than or equal to the cardinality of the column (ROWCOUNT)
4	The sum of the values in INTERVAL_UEC should be equal to the number of distinct values in the column (TOTAL_UEC)
5	HIGH_VALUE should be greater than LOW_VALUE, whenever there are more than one distinct values in the corresponding column (TOTAL_UEC), otherwise HIGH_VALUE should be equal to LOW_VALUE. In case of multicolumn histograms, the High value and Low Value of each columns must match with the High Value and Low Value of corresponding single column histograms over those columns
6, 7	The values in INTERVAL_BOUNDARY should lie between LOW_VALUE and HIGH_VALUE and the last value in INTERVAL_BOUNDARY should be equal to HIGH_VALUE.
8	The values in INTERVAL_BOUNDARY should be in increasing order.
9	The value of INTERVAL_ROWCOUNT should be greater than or equal to the value of INTERVAL_UEC, and these values should be equal if there exists a UNIQUE constraint on the column. If the value of INTERVAL_UEC is Zero, then the value of INTERVAL_ROWCOUNT should also be Zero.

Figure 6: Consistency Constraints

In earlier version of CODD, some of the structural constraints as well as the consistency constraints were missing, because of which the metadata written was inconsistent in some of the cases. To ensure that the metadata is legal and consistent, we discussed all the constraints with COM_OPT developers and then came up with some additional constraints, which have been added in current version of CODD for COM_OPT. The final set of constraints are shown in Figures 5 and 6.

4.3 Metadata Scaling

CODD supports two kinds of metadata scaling for COM_OPT, *Size-based scaling* and *Cost-based Scaling*. The details of both are given below.

4.3.1 Size-based Scaling

Given an initial metadata shell and a scaling factor α, as input, it produces a scaled metadata shell such that the size of the database represented by the scaled metadata shell, is α times of the size (in Bytes) of the database represented by the initial metadata shell.

For *size-based scaling*, the metadata statistics for the relations and their attributes are scaled separately. To scale the size of a relation, the *cardinality* of the relation is scaled by a factor of α. The size scaling is implemented differently for different kinds of columns. If the columns are either part of primary key, or are part of foreign key, or have unique constraint on them, then the domain scaling is implemented and the value of *total_uec* is scaled by scale factor α. For the histograms of these columns, the values of *interval_rowcount*, *interval_uec* are scaled by α for each bucket, and the domain scaling is implemented for *interval_boundary*. For domain scaling, if the data type of the columns are either *integer* or *largeint*, then *low_value* and *high_value* are scaled by scale factor α for each column, and the value of *interval_boundary* is scaled by scale factor α for each bucket, otherwise these values remain unchanged. For other columns, the values of *total_uec*, *low_value* and *high_value* remain unchanged, and in the histograms of these columns, the value of *interval_rowcount* is scaled by α, while the values of *interval_uec* and *interval_boundary* remain unchanged in each bucket.

One limitation of the current implementation of *size scaling* is that, the *string* type columns should not have a comma(,) present in *low_value* or *high_value*, which also separates the values of different columns for multi-column attributes.

4.3.2 Cost-based Scaling

Given a query workload Q, an initial metadata shell M, and a scaling factor α, as input, it produces a scaled metadata shell M^α, such that the optimizer's estimated cost of executing the query workload Q on scaled metadata shell M^α is scaled by the scale factor α. The details of the implementation of *cost-based scaling* algorithm are given in [2].

For example, consider a 1GB TPC-H workload consisting of queries Q3, Q4 and Q12, with full selectivity, which operates on *lineitem*, *customer* and *orders* relations. Lets suppose the probabilities of the appearance of these queries are 0.4, 0.3 and 0.3 respectively, and we want to scale the execution cost of these queries by a scale factor of 3. The *cost scaling*, on this workload gives us a unique solution with the scale factors 3, 1 and 3 for the three relations respectively, and the overall scaling factor obtained is 2.37. The cost of queries before scaling and after scaling, along with scale factors obtained for each query is shown in Figure 7.

Cost / Query	Probability	Cost before scaling	Cost after scaling	Scaling factor obtained
Q3	0.4	406.89	772.52	1.89
Q4	0.3	97.43	354.98	3.64
Q12	0.3	203.54	610.57	2.99

Figure 7: Cost of queries after cost-based scaling

4.4 Brief details of the additional modifications done for current version:

- Improved the functionality to create and delete the histogram buckets in construct mode interface
- Improved Graphical Histogram Editing Interface to make it intuitive.
- Added additional validation tests for histogram validity in Construct Mode Interface and added support to allow a bucket to have zero records.
- Earlier graphical histogram editing interface was supporting only $2^{32} - 1$ cardinality. Now it has been to updated to allow $2^{64} - 1$ cardinality for columns.
- Improved functionality to give proper error messages, whenever there is any issue.

- Added the support to trim all the strings read from construct mode user interface.
- Removed the SYSKEY attribute from the attribute drop-down, since it is created only when we don't have the primary key on the relation.
- For string type columns, only first 30 characters are written while writing the histogram in database catalog tables. For this, we added the functionality to write only first 30 characters of the string in metadata tables.
- Added the support for *date* and *timestamp* data types.
- Modified scale mode to work properly for multi-column attributes, etc.

4.5 Experimental Results

For doing our experiments, we tried to construct a 1-Yottabyte($10^{24} Bytes$) TPC-H dataless database on COM_OPT using CODD, and during this construction, we found that it is not possible for COM_OPT to support 1-Yottabyte TPC-H database. When we explored more, we found that the maximum TPC-H database size supported on COM_OPT is 1-Zettabyte ($10^{21} Bytes$), and if it has to support a database bigger than this, then it will need significant changes in its architecture. We informed about this issue to COM_OPT developers, and then we started our experiments by constructing a 1-Zettabyte TPC-H dataless database. To analyze the behavior of COM_OPT optimizer, we generated different kinds of Picasso Diagrams[5], and then analyzed them. Figure 8 shows a 300x300 resolution Picasso plan diagram for TPC-H Query Template 21, see Figure 12, on 1-Zettabyte dataless database which contains 158 different plans.

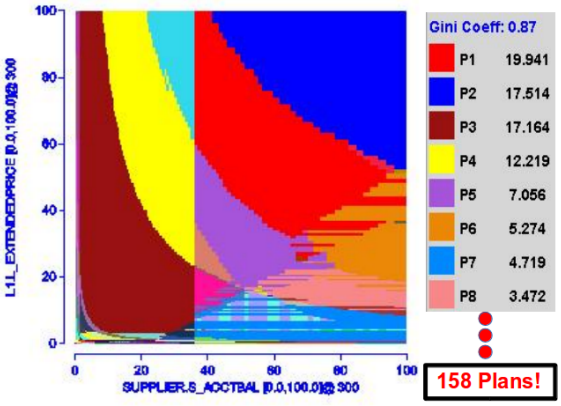

Figure 8: Plan Diagram for Query Template 21

4.5.1 Analysis of diagrams for TPC-H Query Template 8

Testing Environment

- Database: 1-Zettabyte ($10^{21} Bytes$) TPC-H dataless database, generated using CODD, with indexes on all attributes
- Computational Platform: COM_OPT Operating System, Intel Itanium Quad Core CPU 1.6 GHz, 16 GB RAM
- Picasso Settings: Diagram resolution was set to 100 and the distribution of query points was set to exponential. Optimization level was set to the highest level.

Figure 9 shows Query Template 8 of TPC-H Benchmark with selectivity variation on S_ACCTBAL and L_EXTENDEDPRICE attributes of SUPPLIER and LINEITEM relations, respectively. For testing of COM_OPT on big data, we created 1-Zettabyte TPC-H dataless database by scaling the metadata of 1GB TPC-H database by scale factor of 10^{12}, and then we generated Plan Diagram, Compilation Cardinality Diagram and Compilation Cost Diagram for Query Template 8 using Picasso and analyzed the behavior of COM_OPT on big data system.

```
SELECT
    O_YEAR,
    SUM(CASE
        WHEN NATION = 'BRAZIL' THEN VOLUME
        ELSE 0
    END) / SUM(VOLUME)
FROM(
    SELECT
        YEAR(O_ORDERDATE) AS O_YEAR,
        L_EXTENDEDPRICE * (1 - L_DISCOUNT) AS VOLUME,
        N2.N_NAME AS NATION
    FROM
        PART, SUPPLIER,     LINEITEM, ORDERS,
        CUSTOMER, NATION N1, NATION N2, REGION
    WHERE
        P_PARTKEY = L_PARTKEY
        AND S_SUPPKEY = L_SUPPKEY
        AND L_ORDERKEY = O_ORDERKEY
        AND O_CUSTKEY = C_CUSTKEY
        AND C_NATIONKEY = N1.N_NATIONKEY
        AND N1.N_REGIONKEY = R_REGIONKEY
        AND R_NAME = 'AMERICA'
        AND S_NATIONKEY = N2.N_NATIONKEY
        AND O_ORDERDATE BETWEEN
        DATE'1995-01-01' AND DATE '1996-12-31'
        AND P_TYPE = 'ECONOMY ANODIZED STEEL'
        AND S_ACCTBAL :VARIES
        AND L_EXTENDEDPRICE :VARIES
    ) AS ALL_NATIONS
GROUP BY O_YEAR
ORDER BY O_YEAR
```

Figure 9: Query Template 8 of TPC-H Benchmark

i. Plan Diagram

Figure 10 shows the plan diagram for Query Template 8. A set of 47 different optimal plans, P1 through P47, covers the entire selectivity space. The legend shows that plan P1 is covering 68.48% of the space, whereas plan P5 onwards cover less than 1% area each, which shows that the optimizer has made extremely fine grain choices near the low selectivity region. There are plans which are not convex and unique. For example, plan P3 is occurring inside P2 at multiple places. The plans here violates the assumptions of PQO[7] that the plan regions must be convex and unique.

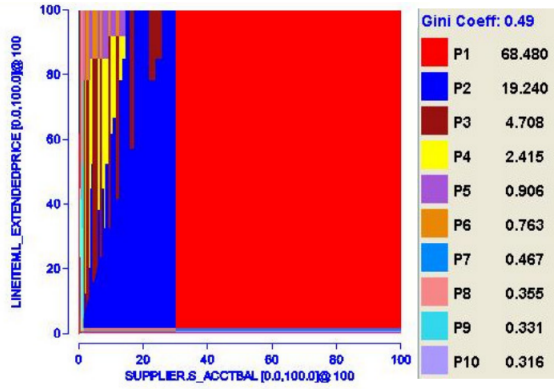

Figure 10: Plan Diagram for Query Template 8

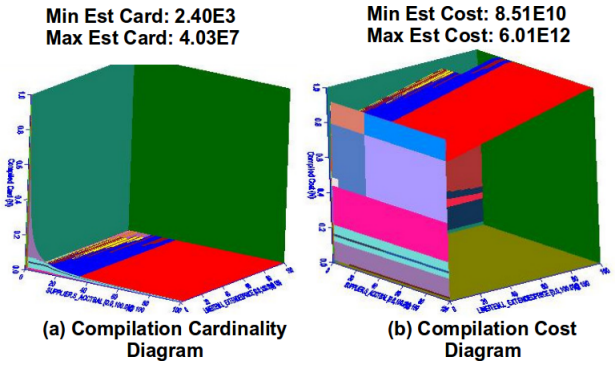

Figure 11: Compilation Cardinality and Compilation Cost Diagrams for Query Template 8

ii. Compilation Cardinality Diagram

Figure 11(a) shows the compilation cardinality diagram for Query Template 8, which looks very interesting. In this diagram, most of the area has same cardinality except the area near to origin. For Query Template 8, maximum cardinality can reach only upto 2.4E3, since in Query Template 8, there is an *order by* clause on *o_year* attribute of *orders* table which had only 2.4E3 distinct values in 1-Zettabyte dataless database, but what we see in the diagram is that at origin, the cardinality reaches upto 4.03E7 which is wrong. It shows the possibility of bugs in cardinality estimation module of COM_OPT for a big data system.

iii. Compilation Cost Diagram

Figure 11(b) shows the compilation cost diagram for Query Template 8. The compilation cost diagram shows that the cost of query execution increases along with the increase in selectivities in most of the regions. However, there are some regions where we see a sudden decrease in the query execution cost with the increase in the selectivities, which is a clear violation of Plan Cost Monotonicity.

5 Hardware Configuration For Hardware-based Testing

In Section 4, we engineered CODD Metadata Processor for COM_OPT, which is used for testing of big data environments using the metadata of that big data. Since it uses only the metadata, creating alternative big data scenarios is highly efficient in terms of time and space. Whenever the database engine executes a query, it tries to determine the most efficient plan to execute the query by considering the possible query plans. The choice of optimal query execution plan depends on the metadata as well as the hardware available in the system. As the size of data is growing day by day, we will need more and more hardware to process that data in future. Earlier CODD had the support only for the simulation of futuristic big data, not the futuristic hardware, hence it was not possible to fully simulate the futuristic big data environments. In this work, we worked in collaboration with COM_OPT developers and added the functionality which allows us to simulate alternative hardware scenarios for COM_OPT. Now the COM_OPT database engine can be tested on futuristic big data environments, without having the futuristic big data as well as futuristic hardware actually present in the system, which was not possible earlier.

5.1 Implementation Details

To simulate different hardware configurations, COM_OPT provides OSIM (Optimizer Simulation)

environment, which has several configuration parameters related to system configuration that can be modified by the user. The following are the steps to simulate different hardware configurations:

- First generate the configuration files for OSIM using the following command:

  CONTROL QUERY DEFAULT OSIM_CAPTURE_LOC 'C:\osim.test'

  This command generates various configuration files, under the directory 'C:\osim.test', which are used for the simulation of virtual hardware environments.

- Update the CPU related information in following configuration file:

  C:\osim.test\NAClusterInfo.txt

- Update the memory related information in following configuration file:

  C:\osim.test\DEFAULT_DEFAULTSFILE.txt

- Make sure that all the tables present in database have atleast one record present, so that metadata statistics are not empty. Otherwise the simulation mode will not work and will give errors.

- Generate metadata statistics on all tables present in the database using the following command:

  UPDATE STATISTICS FOR TABLE <TABLENAME> ON EVERY COLUMN;

- The next command is used to run the COM_OPT in simulation mode:

  CONTROL QUERY DEFAULT OSIM_SIMULATION_LOCATION 'C:\osim.test'

- The setup for hardware simulation is done and the query optimizer will now work in simulated hardware environment.

- To exit the simulation mode, use the following command:

  CONTROL QUERY DEFAULT OSIM_SIMULATION_LOCATION ' '

5.2 Experimental Results

5.2.1 Testing Environment

- Database: 1-ZettaByte ($10^{21} Bytes$) TPC-H dataless database, generated using CODD, without any index on attributes

- Computational Platform: Windows XP Operating System, Intel Core i3 CPU 2.53 GHz, 4 GB RAM

- Picasso Settings: Diagram resolution was set to 30 and the distribution of query points was set to uniform. Optimization level was set to the highest level.

```
SELECT
    S_NAME, COUNT(*) AS NUMWAIT
FROM
    SUPPLIER, LINEITEM L1, ORDERS, NATION
WHERE
    S_SUPPKEY = L1.L_SUPPKEY
    AND O_ORDERKEY = L1.L_ORDERKEY
    AND O_ORDERSTATUS = 'F'
    AND EXISTS (
        SELECT * FROM LINEITEM L2
        WHERE
        L2.L_ORDERKEY = L1.L_ORDERKEY
        AND L2.L_SUPPKEY <> L1.L_SUPPKEY
    ) AND NOT EXISTS (
        SELECT * FROM LINEITEM L3
        WHERE
        L3.L_ORDERKEY = L1.L_ORDERKEY
        AND L3.L_SUPPKEY <> L1.L_SUPPKEY
        AND L3.L_RECEIPTDATE > L3.L_COMMITDATE
    )
    AND S_NATIONKEY = N_NATIONKEY
    AND S_ACCTBAL :VARIES
    AND L1.L_EXTENDEDPRICE :VARIES
    AND N_NAME = 'SAUDI ARABIA'
GROUP BY
    S_NAME
ORDER BY
    NUMWAIT DESC, S_NAME
```

Figure 12: Query Template 21 of TPC-H Benchmark

5.2.2 Analysis of diagrams for TPC-H Query Template 21

To do the experiments for hardware simulation, different hardware settings were configured and then plan diagrams were generated using Picasso. Query Template 21 of TPC-H Benchmark, see Figure 12, with selectivity variation on S_ACCTBAL and L_EXTENDEDPRICE attributes of SUPPLIER and LINEITEM relations respectively, was used for generation of plan diagrams. These experiments were done on the 32-bit version of COM_OPT. Figure 13 shows the Picasso plan diagrams, generated for 4 different combinations of Memory and CPUs. It also shows the number of plans generated along with the minimum and maximum cost to execute the query.

- Top Left diagram shows the plan diagram generated for 64MB Memory and 1 CPU.

- Top Right diagram shows the plan diagram generated for 64MB Memory and 16 CPUs.
- Bottom Left diagram shows the plan diagram generated for 4GB Memory and 1 CPU.
- Bottom Right diagram shows the plan diagram generated for 4GB Memory and 16 CPUs.

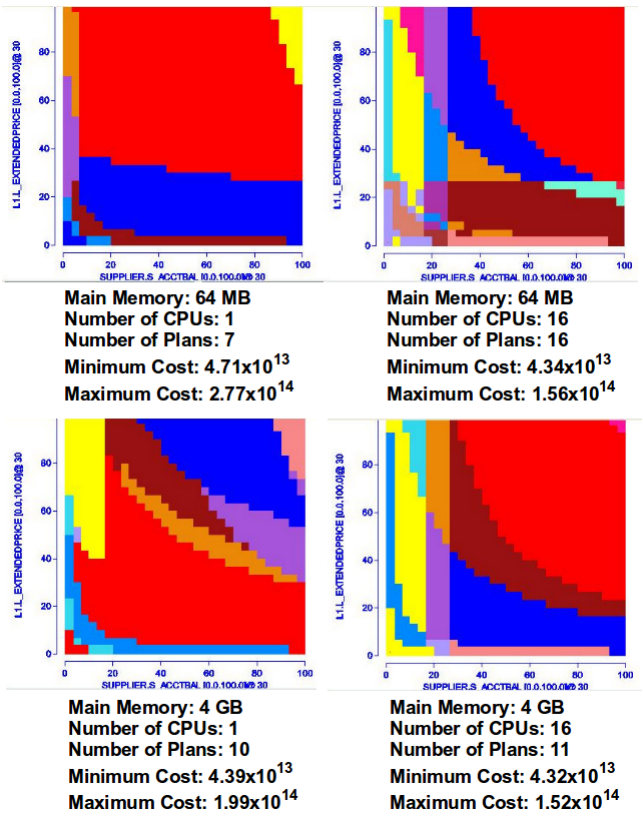

Figure 13: Picasso Diagrams for Hardware Simulation

From Figure 13, it's clear that the cost of executing the queries decreases when we increase the memory or the number of CPUs, but the improvement is not significant. These experiments were done on 32-bit version of COM_OPT, which supports only upto 4GB of memory. Therefore we don't see much improvement in costs after increasing the memory, because the database size is very large compared to the memory supported by the database engine. The same approach can be used on 64-bit version of COM_OPT to simulate systems with higher memory, which can give better insights about the improvement in cost after increasing the memory. The most important thing here is that increasing the number of CPUs is not giving much improvement in cost. COM_OPT is designed for highly parallel machine architectures, and it is expected that increasing the number of CPUs should give significant improvement in the execution cost of queries on big data.

6 Execution Time Testing

The compile time testing of database engines can be done using metadata statistics, but for execution time testing, the actual data needs to be present in the system. Generating and storing alternative big data scenarios for testing is infeasible even on high end systems, because of time and space constraints. Therefore we need a way to do execution time testing without storing the database contents. This can be made possible, if we do on-the-fly generation of data on the operators of the query execution plan whenever a query gets executed. The first step in on-the-fly data generation is to generate a synthetic database, which has similar distributions as the actual database. In Section 8, we have discussed different kinds of techniques available to generate synthetic databases. The technique which is closely related to this work is by Shen and Antova[8], which uses the single dimensional histograms present in database catalog to generate the synthetic data. The drawback of this technique is that it focuses only on single dimensional histograms for data generation, because of which the correlations present in the database are lost. This section focuses on the generation of a synthetic database using PCA based techniques, such that the generated database has similar distributions to the actual database, as well as, all the correlations are also maintained.

6.1 Problem Statement

- Given the following information about a database instance D:
  - Logical Schema S, and
  - Set of schematic constraints C, such as primary key, foreign key, not null, unique etc.
- Generate a synthetic database instance $\bar{D}$, which:
  - has the same logical schema as S,
  - satisfies all the schematic constraints in C,
  - has the same size (Number of records in each relation) as D, and
  - each relation $\bar{R}_i \in \bar{D}$ has data distributions, similar to $R_i \in D$.

6.2 Assumptions for the suggested solution:

The suggested solution is only a partial solution and following are the assumptions made for it:

- Relations are in 3-NF,
- There exists no NULL values in the relations,
- The data present in relations is numeric type,
- There is no foreign key relationship between relations.

6.3 Synthetic Database Generation

Let relation R contains 5 attributes: A_1, A_2, A_3, A_4, A_5, and n rows, as shown in left side of Figure 14. Consider each attribute A_i of R as a Random Variable and let $\bar{A}_i$ be the mean of A_i. Calculate Covariance Matrix of R, using the following formula:

$$Cov(A_i, A_j) = \frac{1}{n-1} \sum_{k=1}^{n} \Big(A_{i,k} - \bar{A}_i\Big)\Big(A_{j,k} - \bar{A}_j\Big)$$

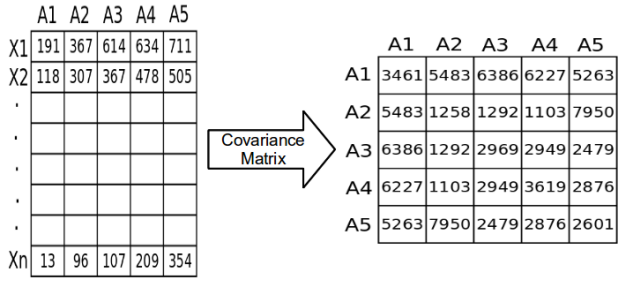

Figure 14: Covariance Matrix calculated from Relation R

The Covariance Matrix calculated from Relation R will be a square matrix, as shown in right side of Figure 14. If the covariance matrix is diagonal, then it represents that all attributes of the relation are uncorrelated and we can generate the data for each attribute using 1-dimensional histograms, as discussed in [8]. The main issue comes when some or all the attributes of the relation are correlated, in which case the covariance matrix will not be diagonal. For such a case, we can use Principal Component Analysis to decorrelate all attributes of the relation. PCA provides a linear transformation of the data such that after the transformation, the attributes of the relation become uncorrelated with each other. This transformation is given by multiplication with the matrix P, which contains the principal components of the relation in each column.

Let us suppose that we apply PCA on relation R to get the uncorrelated relation R_{PCA}, using the following transformation:

$$\mathbf{R_{PCA}} = \mathbf{R}.\mathbf{P},$$

where P is an orthonormal matrix, which contains the principal components of R in each column. Here the Covariance Matrix of relation R_{PCA} will be diagonal, as shown in Figure 15, since the relation R_{PCA} is uncorrelated.

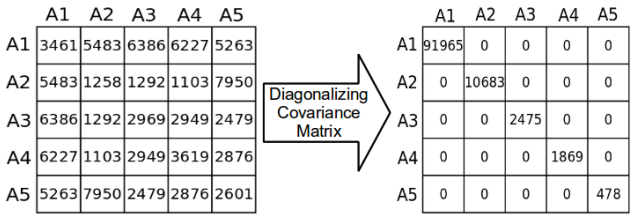

Figure 15: Diagonal Covariance Matrix calculated from Covariance Matrix

Since all the attributes of R_{PCA} are uncorrelated, we can generate the 1-dimensional histograms from R_{PCA} and generate the synthetic relation $\bar{R}_{PCA}$ (in transformed space), by generating data for each column independently using those 1-dimensional histograms, as shown in Algorithm 1.

Algorithm 1 Calculating the histograms:

1: Take a relation R as input.
2: Calculate the mean $\bar{A}_i$ of each column A_i of R, and store it.
3: From each column A_i of R, subtract its mean $\bar{A}_i$.
4: Calculate the eigen vectors of $R^T R$, which are called the principal components of R.
5: Arrange the eigen vectors of $R^T R$ in a matrix P, where each column of P is one of the eigen vectors.
6: Calculate $R_{PCA} = R.P$
7: Calculate 1-dimensional histograms for each column of R_{PCA}.

From synthetic relation $\bar{R}_{PCA}$ (in transformed space), we can generate the synthetic relation $\bar{R}$ (in actual space), as shown in Algorithm 2, using the following inverse transformation:

$$\bar{R} = \bar{R}_{PCA}.P^{-1} = \bar{R}_{PCA}.P^T$$

Here $P^{-1} = P^T$, since P is an orthonormal matrix.

Algorithm 2 Generating the synthetic relation:

1: Generate the synthetic relation $\bar{R}_{PCA}$ (in transformed space), using 1-dimensional histograms.
2: Calculate the synthetic relation $\bar{R}$ (in actual space), using the following inverse transformation:

$$\bar{R} = \bar{R}_{PCA}.P^T$$

3: In each column A_i of R, add the mean $\bar{A}_i$, which was calculated earlier.
4: Return $\bar{R}$.

6.4 Overview of the synthetic database generation algorithm

Figure 16 gives us the overview of the algorithm to generate a Synthetic Relation $\bar{R}$, corresponding to the actual Relation R, using Principal Component Analysis and 1-dimensional histograms. The dotted arrow represents the synthetic relations corresponding to the actual relations.

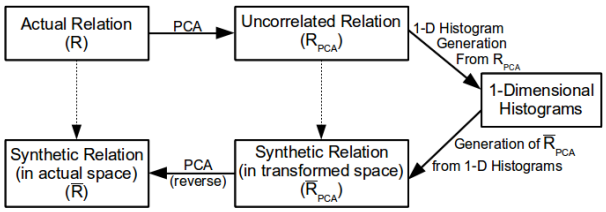

Figure 16: Original Relation to Synthetic Relation

6.5 Experimental Results

To test the performance of our algorithm, we took first 1.5 million records from *store_returns* relation of TPC-DS database as input, call it R, and then generated one *synthetic relation* $\bar{R}$ using our PCA based algorithm and another *synthetic relation* R^{1D} using 1-dimensional histogram based algorithm given by Shen and Antova[8]. Then we did two kinds of comparison between our algorithm and the 1-D histogram based algorithm.

6.5.1 Comparison based on correlation coefficients and attribute distributions

For comparing the correlation coefficients, we took all the attribute pairs of three correlated attributes of *store_returns* relation: *sr_return_quantity*, *sr_return_amt* and *sr_return_tax*, and then compared the correlation coefficients for those attribute pairs. The table shown in Figure 17 contains the correlation coefficients between attribute pairs, shown in first column, for original relation R as well as synthetic relations $\bar{R}$ and R^{1D} generated using both algorithms. The second, third and fourth column contains the values of correlation coefficients for the attribute pairs in R, $\bar{R}$ and R^{1D} respectively. From this table, it's clear that our algorithm retains most of the correlations because we are generating the data after making all the attributes uncorrelated using PCA and then again converting those uncorrelated attributes into correlated, while the 1-D histogram based algorithm loses all these correlations because the data is generated independently even for correlated attributes.

Correlation Coefficient for: / Attribute Pairs	Original Relation (R)	Synthetic Relation ($\bar{R}$)	Synthetic Relation (R^{1D})
(sr_return_quantity, sr_return_amt)	0.58	0.57	0.00
(sr_return_quantity, sr_return_tax)	0.46	0.49	0.00
(sr_return_amt, sr_return_tax)	0.79	0.85	0.00

Figure 17: Correlation Coefficients between attribute pairs for Original Relation R, and Synthetic Relations $\bar{R}$ and R^{1D}

Attribute Name / KL-divergence From Original Relation (R) to:	Synthetic Relation ($\bar{R}$)	Synthetic Relation (R^{1D})
sr_return_quantity	4.0E-2	3.0E-2
sr_return_amt	3.9E-3	4.8E-5
sr_return_tax	7.6E-2	3.4E-3

Figure 18: KL-divergence from attributes of Original Relation R to the attributes of: (a) Synthetic Relation $\bar{R}$, (b) Synthetic Relation R^{1D}

For comparing the attribute distributions, we again took the three attributes *sr_return_quantity*, *sr_return_amt* and *sr_return_tax*, approximated their probability distributions using histograms, and then compared the distance between those distributions using KL-divergence[9]. The KL-divergence gives us the difference between two probability distributions, and the value of KL-divergence is near to zero if two distributions are similar to each other. From Figure 18, it's clear that both algorithms gives the attribute distribution similar to the attribute of original relation, but relatively 1-D histogram based algorithm gives better results then our algorithm, because it is directly

generating the distribution for each attribute without changing the basis.

6.5.2 Comparison based on Picasso diagrams

For comparing the Picasso diagrams, we formed the following query with selectivity variation on *sr_return_amt* and *sr_return_tax* attributes of *store_returns* relation, which are highly correlated, see Figure 17.

```
select * from
    store_returns s1, store_returns s2
where
    s1.sr_item_sk = s2.sr_item_sk
and s1.sr_ticket_number = s2.sr_ticket_number
and s1.sr_return_amt :varies
and s2.sr_return_tax :varies
```

In Picasso, we generated the diagrams with resolution 30 on both axis and the distribution of query points was set to uniform. Optimization level was set to the highest level. Because of some technical issues related to loading of data in 32-bit test version of COM_OPT, these diagrams were generated on another widely used commercial database optimizer.

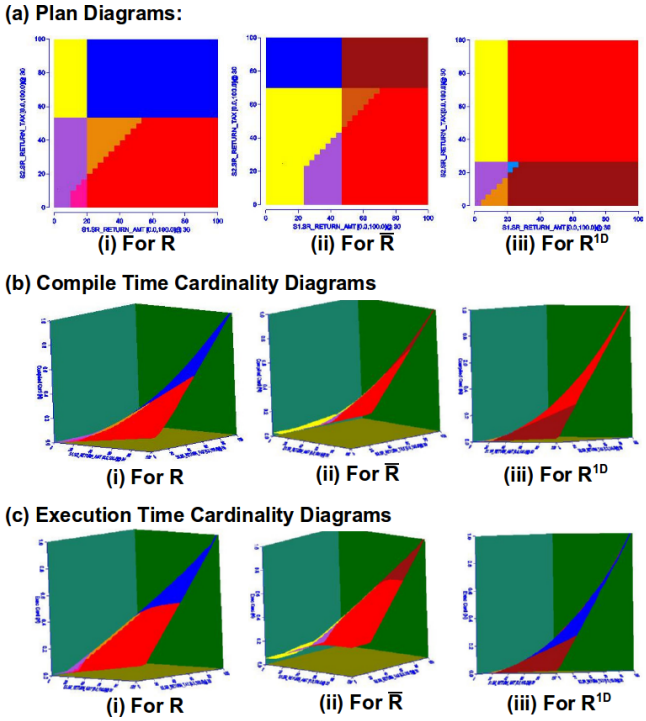

Figure 19: Picasso Diagrams for: (i) Original Relation R, (ii) Synthetic Relation $\bar{R}$, (iii) Synthetic Relation R^{1D}

Figure 19(a) contains the plan diagrams generated on relations R, $\bar{R}$ and R^{1D}. From this figure, we can see that all the plan diagrams have same number of plans, which is 6, and the area and position of each plan is different in different diagrams. Here it's clear that except the low selectivity region, the plan diagram for $\bar{R}$ is more similar to the plan diagram R, which shows that our algorithm works better than 1-D histogram based algorithm.

Figure 19(b) contains the compile time cardinality diagrams generated on relations R, $\bar{R}$ and R^{1D}. From this figure, we can see that the shape of the diagram on relation R^{1D} is more close to the shape of diagram on original relation R. The diagram generated on our relation $\bar{R}$ is also similar to the diagram on original relation R except for the low selectivity region.

Figure 19(c) contains the execution time cardinality diagrams for all three relations. From this figure, we can see that the shape of the diagram for relation $\bar{R}$ is more close to the shape of diagram for original relation R, except for the low selectivity region. The shapes of compilation and execution cardinality diagrams, see Figures 19(b)(iii) and 19(c)(iii), generated on relation R^{1D} look very similar, because the optimizer at compile time assumes the attribute value independence property, which is true for the relation R^{1D} at execution time, since all its attributes are generated using 1-D histograms independently. While the shapes of compile and execution time cardinality diagrams generated on our relation $\bar{R}$, see Figures 19(b)(ii) and 19(c)(ii), are dissimilar to each other, because we capture the correlations, which is clear from the difference between the slope of both diagrams.

These experiments show that our PCA based algorithm works far better than 1-D histogram based algorithm, but it still needs more improvements, because it does not work well for low selectivity region.

7 Coding Efforts in New Features

In this work, we did all our implementations using the Java language. For metadata configuration and hardware configuration part, no additional library was used, except the libraries which had been used earlier for CODD. For synthetic data generation, we used three mathematics libraries. The Apache Commons Mathematics Library[10] was used for matrix multiplications and generating random numbers. The EJML[11] library was used for Principal Components Analysis. The BEAST2[12] library was used initially

for generating a covariance matrix using the Wishart distribution, which was used to generate the correlated data for our experiments. Later we used the store_returns relation of TPC-DS database, which had the correlated data. In this work, we improved about 8K lines of code and added about 2K lines of new code.

8 Literature Survey

Synthetic databases are widely used for testing of database system components and performance testing of database engines. There has been a wide variety of research done on generation of synthetic databases, which cover different aspects of synthetic database generation, as discussed below:

i. Synthetic databases with large amount of data:

As database sizes grow, the generation of databases starts taking longer than the testing of database system components. To solve this problem, Gray et al presented a paper[13] which focuses on generation of databases having large amount of data quickly using parallelism. It first discusses the techniques to generate the data sequentially and then generalizes those techniques to generate the data in parallel. It also discusses various techniques for generation of *Dense Unique Random Data*, and then shows that the technique which generates the numbers using a *generator of the cyclic group of integers under multiplication* is best for this purpose. The algorithm presented in this paper is as follows. Pick a prime P larger than N and a generator G for the multiplicative group modulo P. Then the series is:

$$< G^i \bmod P \,|\, i = 1, \ldots, P \;\; and \;\; (G^i \bmod P) \leq N >$$

ii. Synthetic databases with various kinds of data distributions:

The paper by Bruno and Chaudhury[14] focuses on generating databases with different kinds of inter and intra table dependencies. For this, they presented a special purpose language called DGL - Data Generation Language, for generation of synthetic databases with different kinds of distributions in each column. DGL supports various kinds of data types and it has a big collection of primitive iterators, which can generate different kinds of distributions, store the generated data into database tables etc.

iii. Synthetic databases which are query aware:

The paper by Binnig et al[15] presented a unique concept of reverse query processing, which is opposite to traditional query processing. In reverse query processing, we are given a query, its output and a database schema, as input, and by using *reverse query processing*, we create a database instance which satisfies all the consistency constraints and gives the same output result. Reverse query processing has applications in generating test databases and functional testing of database applications.

In [16], Lo et al presented a framework for testing DBMS features, which is an extended version of their earlier work[17]. In this paper, they presented the concept of *symbolic query processing*, and a database generator called QAGen, which is based on *symbolic query processing*. QAGen is a query aware database generator which takes a query and the set of constraints, like cardinality at each query operator, defined on the query as input and generates a query aware database as output.

iv. Synthetic databases with similar data distributions as original databases:

Execution time testing part of our work falls in this category, since we are generating a synthetic database, which has data distributions similar to a given real world database. It is different from [15] and [16], in the sense that they were generating databases which were query dependent and had nothing to do with the data present in the original database, while we are generating a database which has data distributions similar to the real world database. The work which is closely related to our work on synthetic database generation is by Shen and Antova[8]. In [8], Shen and Antova presented a tool called RSGen, which reverses the metadata statistics for scalable test database generation. This tool uses the histograms available in database catalogs of database systems to generate the data for each attribute independently. The drawback of this tool is that it focuses only on single dimensional histograms for data generation, because of which the correlations present in the database are lost, see Section 6.5.

9 Conclusions and Future Work

In this work, we engineered CODD for COM_OPT, which is a distributed relational database system, designed for highly parallel system architectures. We also added two additional features in CODD,

Hardware-based testing and *Synthetic database generation for execution time testing*. During our experiments, we found some serious issues in COM_OPT, related to the maximum database size supported, and the cardinality estimation module of the query optimizer. The COM_OPT currently supports only 10^{18} cardinality in a relation, which may not be enough for a futuristic big database. These issues have also been communicated to the COM_OPT developers, and they are looking into these issues with high priority.

In Section 6, we proposed a PCA based algorithm to generate a synthetic database, similar to the original database, for execution time testing. This algorithm assumes that there are no null values in the relations and there are no foreign-key constraints. This algorithm needs to be improved to take care of null values and foreign-key constraints. Currently this algorithm works only for numeric type data, and it needs to be extended to work for other kinds of data types. In future, our plan is to extend this synthetic database generation algorithm to mimic full scale execution time environments for big data testing, by generating the data at query execution time instead of persistently storing the data.

References

[1] As big data explodes, are you ready for yottabytes? http://www.forbes.com/sites/oracle/2013/06/21/as-big-data-explodes-are-you-ready-for-yottabytes.

[2] Rakshit S. Trivedi, I. Nilavalagan, and Jayant R. Haritsa. Codd: Constructing dataless databases. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, DBTest '12, pages 4:1–4:6. ACM, 2012.

[3] TPC Benchmark H. http://www.tpc.org/tpch.

[4] TPC Benchmark DS. http://www.tpc.org/tpcds.

[5] Harish D, Pooja N. Darera, and Jayant R. Haritsa. On the production of anorexic plan diagrams. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1081–1092. VLDB Endowment, 2007.

[6] Deepali Nemade. Analyzing the behavior of a distributed database query optimizer - http://dsl.serc.iisc.ernet.in/publications/thesis/deepali.pdf.

[7] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. *The VLDB Journal*, 6(2):132–151, May 1997.

[8] Entong Shen and Lyublena Antova. Reversing statistics for scalable test databases generation. In *Proceedings of the Sixth International Workshop on Testing Database Systems*, DBTest '13, pages 7:1–7:6. ACM, 2013.

[9] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, pages 79–86, 1951.

[10] Commons Math: The Apache Commons Mathematics Library. http://commons.apache.org/proper/commons-math/.

[11] EJML: Efficient Java Matrix Library. http://code.google.com/p/efficient-java-matrix-library/.

[12] BEAST2: Bayesian Evolutionary Analysis by Sampling Trees. http://www.beast2.org/.

[13] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 243–252. ACM, 1994.

[14] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1097–1107. VLDB Endowment, 2005.

[15] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *Proceedings of the 23rd International Conference on Data Engineering*, ICDE '07, pages 506–515. IEEE, April 2007.

[16] Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. A framework for testing dbms features. *The VLDB Journal*, 19(2):203–230, April 2010.

[17] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: Generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 341–352.