Design and Implementation of CODD 1.0

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Engineering

IN

COMPUTER SCIENCE AND ENGINEERING

by

Ashoke Sudakar

Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012 (INDIA)

JULY 2015

Dedicated

To

My Family

Acknowledgements

First and foremost I would like to express my sincere gratitude to my project advisor, Prof. Jayant R. Haritsa for guiding me throughout this project. I thank him for his valuable guidance and moral support. I would like to thank the Department of CSA for providing excellent study environment and infrastructure. It has been a truly wonderful learning experience. I thank the faculty and the staff for their regular support, help and encouragement.

I am very grateful to my family for their continuous support, especially my sister Roopa who encouraged me to pursue my Masters. I express my sincere gratitude to my lab mates; Anshuman Dutt and Srinivas Karthik in particular, who have always been very approachable and helpful. Last but not the least, I thank my classmates and friends in IISc who helped me throughout my stay here. They made the IISc experience truly wonderful, fun and enriching.

Abstract

In this era of Big Data, the volume of data is increasing in an unprecedented manner and this is likely to continue in the future [1]. Database engines should be robust in order to handle such a huge surge in the data volume. This requires that the database engines are tested on various futuristic Big data scenarios, but the resources required to generate and store such scenarios are constrained due to space and time. 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. This calls for a tool, that would allow users to simulate such environments virtually in low-end systems. CODD [2], a java based graphical tool developed in our lab, aims to achieve the aforementioned by creating *dataless databases* in database systems. In this work, we have engineered CODD for (a) simulating different metadata configurations on PostgreSQL for query compile time testing, (b) simulating different hardware configurations on ComOptA[1] for hardware based testing and (c) synthetic database generation for query execution time testing.

[1]a popular commercial database engine.

Contents

List of Figures

Chapter 1

Introduction

The last decade has seen a tremendous amount of increase in the volumes of data and this is likely to continue in the future [1]. In order to handle such a huge amount of data, the database engines should be robust. In order for them to be robust against futuristic databases, we need to be able to test them on various alternative, futuristic Big Data scenarios. However, traditional database performance testing do not reveal how database engines handle futuristic Big Data, as neither there is an infrastructure available to store the test data nor it is feasible to test them due to time constraints. This motivates the idea for a testing tool, that would allow users to simulate such environments *virtually*, even in low-end systems. In particular, our contributions are the following:

i. Query Compile Time Testing

Given a query to be executed on a database, the *query compile time* involves the use of the metadata configuration of the database, by the query optimizer, to create an optimal execution plan for the query. Thus, this stage doesn't require the actual data to be present in the system for the optimizer to make its decisions. Leveraging this advantage, CODD [2], an easy to use graphical tool, was developed by Trivedi et al. that aids in automated creation, verification, retention, scaling and porting of database metadata configurations. Users can generate different metadata scenarios and test for the query optimizer's efficiency. Using the *size-scaling mode* of CODD, users can also simulate futuristic Big Data scenarios for compile time testing in a few minutes. Earlier, CODD was functional for IBM DB2, MS SQL Server, Oracle and HP NonStop SQL. On this front, our work focuses on engineering CODD for PostgreSQL and adding additional functionality for ComOptB[1].

[1]Another popular commercial database engine

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. Thus, testing the functionality of query optimizer on different kinds of hardware configurations 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 ComOptA, a popular commercial optimizer, on futuristic hardware scenarios.

iii. Query Execution Time Testing

Unlike compile time testing, *query execution time testing* requires the actual data to be present in the system. In Big Data scenarios, this presents a challenge for testing, as original data may not be available owing to huge volume or due to privacy concerns. We need a way to generate data in order to perform the execution time testing. This part of our work focuses on generating a synthetic database having data distributions similar to the actual database.

Please note that while Big Data has many different facets, including *variety*, *velocity*, and *volume*, CODD currently addresses only the *volume* aspect.

1.1 Contributions

The following are the new features added to CODD as part of this work.

- Support for PostgreSQL database engine.
- Important enhancements to ComOptB database engine.
- Hardware Configuration for ComOptA database engine.
- Support for creation of classical distribution histograms.

Apart from the above, this work also comprises of the following.

- Verifying that plan bouquet's good robustness extends to Big Data scenarios.
- Synthetic data generation.

CODD version 1.0, released on Feb 2015, is freely downloadable [3], and is currently in use at industrial and academic institutions worldwide. For instance, CODD is part of the design workbench at HP (for SQL/MX platform), TCS (Tata Consultancy Services), a Top-ten IT solutions company and MarketShare, a Big Data marketing analytics startup.

A part of this work[1] got accepted in the demonstrations track of VLDB 2015.

1.2 Organization

The rest of the report is organized as follows. We start with the background of CODD Metadata Processor in Section 2. In Sections 3, 4 and 5 we discuss *metadata configuration for PostgreSQL*, *Verifying that plan bouquet's good robustness extends to Big Data scenarios.*, *hardware configuration for ComOptA* respectively, in detail. In section 6, we look at the different *Design Bugs at Big Data scale* that CODD unveiled. Section 7 talks about a new approach to synthetic data generation for the *execution time testing*. Section 8 gives the details of the coding and other efforts required in this work. Finally we conclude the report with future work in Section 9.

[1]Ashoke S, Jayant R. Haritsa. CODD: A Dataless Approach To Big Data Testing

Chapter 2

Background on CODD Metadata Processor

The following background material is taken from [4].

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 independent of the data size and takes very less time and space.

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 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 factor α.

Chapter 3

Metadata Configuration For Compile Time Testing

CODD was functional already on leading commercial databases. In this work, we have engineered CODD to support PostgreSQL. Support for PostgreSQL was not available earlier because, unlike other databases, PostgreSQL doesn't allow inserting or updating column metadata statistics from outside the system. This required us to modify the database engine code to allow column metadata construction. The following section gives the details about the approach taken to support CODD for PostgreSQL.

3.1 PostgreSQL

PostgreSQL is the most advanced open source object-relational database management system [5]. PostgreSQL optimizer's relation and column statistics are stored in the catalogs PG_CLASS and PG_STATISTIC respectively. The optimizer uses these statistics to create an optimal execution plan for a query. PG_STATISTIC is not readable by the public for privacy reasons; hence, PostgreSQL defined a PG_STATS *view*, which is a publicly readable view on PG_STATISTIC that only exposes information about those tables that the user has permission to read [6]. The rest of this section discusses about the implementation of various modes of CODD for PostgreSQL.

3.1.1 Metadata Construction

Metadata construction mode of CODD allows users to create/input metadata details for each relation without the need for the data to be explicitly present in the relation. PostgreSQL supports the following metadata statistics for each relation:

- *Relation Statistics*
- *Column Statistics*
- *Index Statistics*

The details about how these statistics are engineered in CODD for PostgreSQL are explained below.

1. Relation and Index Statistics PostgreSQL stores the *relation* and the *index* level statistics in PG_CLASS catalog. The two important components of this catalog are *reltuples*, the total number of entries in each table and index, and *relpages*, the number of disk blocks occupied by each table and index. The query optimizer uses these column values in order to estimate the number of rows retrieved by a query [6], which is essential for good query plan choices. In order to achieve this, we run the ANALYZE command which catalogs entries about the relation and index statistics. We then update the statistics with the user's inputs.

When the execution plan for a query is constructed, PostgreSQL does a physical storage test (a check on *relpages*) for each relation and index assuming that the metadata details in the PG_CLASS catalog are obsolete. This is done to check if there is a correspondence between the relation metadata and the actual storage that a relation/index takes. Thus, if the metadata suggests that there are certain numbers of tuples present in the relation/index but in reality the relation/index is empty, PostgreSQL assumes *zero* rows and doesn't use the CODD engineered metadata statistics. To counter this issue, we have introduced a new configuration parameter *skip_pagetest* in *postgresql.conf* file, which is the PostgreSQL database configuration file [6]. If this parameter is set to 'ON', PostgreSQL will skip the physical storage test and use the CODD engineered metadata statistics.

2. Column Statistics PostgreSQL stores the column level statistics in PG_STATISTIC catalog. In order to make an estimate of the fraction of rows that satisfy each condition in *WHERE* clause of a query, the information stored in PG_STATISTIC is used by the optimizer [6]. We faced the following two major complications in constructing column metadata statistics.

i. Anyarray Data type: The PG_STATISTIC catalog couldn't be inserted or updated from outside the database system. The major issue is the presence of the *anyarray* data type for the column *stavaluesN*. In PG_STATS view, this corresponds to the columns *most_common_vals*, which stores a list of most common values in a column of a relation and *histogram_bounds*, which stores a list of values that divide the column's values into groups of approximately equal population. *anyarray* is a pseudo-type in PostgreSQL [6], which indicates that a function accepts any array data type. *"The pseudo-type is used to declare functions that are meant* *only to be called internally by the database system, and not by direct invocation in an SQL query"* [6]. Hence, we cannot update or insert values into any of the pseudo-types from outside the database system. So, we had to come up with a new way to insert and update *anyarray* pseudo-type.

ii. TOAST Technique: The column statistics construction is further complicated by the data types that have a variable-length (*varlena*) representation like *varchar* and *text*. PostgreSQL doesn't allow tuples to span multiple pages. This means that it is not possible to store very large field values directly and hence, uses a technique called TOAST (The Oversized-Attribute Storage Technique) [6] to store very large field values by compressing or breaking them up into multiple physical rows. So, before storing such values into column statistics, appropriate *toasting* function needs to be applied. Similarly, while retrieving the statistics values, appropriate *de-toasting* function needs to be implemented.

In order to overcome these complications, we have introduced a new command DL_ANALYZE, which has a similar construct to the ANALYZE command with a few additional parameters. The syntax of the DL_ANALYZE command is given below:

DL_ANALYZE *relation_name column_name column_type column_metadata*

The path followed by DL_ANALYZE is similar to ANALYZE command, except that when DL_ANALYZE is run, instead of collecting statistics about the column, the metadata details about the desired column, given as *column_metadata*, is inserted into the PG_STATISTIC catalog. The format of the *column_metadata* is given below:

null_frac |avg_width |n_distinct |most_common_vals |
most_common_freqs |histogram_bounds |correlation
value |value |value |value \value \value |
value \value \value |value \value |value

The first row of the above figure lists the column names of the PG_STATS view, which is given for easy understanding. Each column value is separated by a pipe (|) symbol and for a column with multiple values, the values within the column are separated by a backslash (\) symbol. The columns *most_common_vals*, *most_common_freqs* and *histogram_bounds* can have multiple values. The user enters the values in the CODD GUI and after the validation of the values, the *column_metadata* is formatted as mentioned above.

When DL_ANALYZE command is run from CODD, the function *dl_update_attstats*, defined in PostgreSQL for our application, is executed. This function does the required conversion of different data types to the internal *anyarray* pseudo-type. This function also handles the *toasting* and *de-toasting* of the values. The following files of PostgreSQL have been modified to implement DL_ANALYZE functionality:

- *gram.y, analyze.c* (major changes)
- *guc.c, plancat.c, equalfuncs.c, copyfuncs.c, kwlist.h, pgstat.h, parsenodes.h* (minor changes)

PostgreSQL also supports the metadata construction from a baseline configuration. In this mode, users can use the metadata statistics of another database (this statistics can be obtained using the Metadata Transfer mode) as a baseline configuration. Users are also given the option either to use them directly or to edit and modify them as required.

Construct from baseline configuration was used to load the metadata statistics onto the IBM servers. Once the metadata details were saved as a file, which were of few MBs, this was easily transferred to the IBM servers. CODD is supported for PostgreSQL version 9.2 and above; it was also back ported to version 8.3.

All other modes described in the CODD background are also engineered for PostgreSQL.

3.2 ComOptB

ComOptB is an industrial strength distributed database system. CODD has been engineered for ComOptB in [4], but was missing some important functionality and modes that were available for other database systems. In this work, we worked closely with ComOptB developers and implemented the missing functionality so that it could be successfully deployed as production-level software. The following section discusses the details about the newly added modes and the improvements made to the existing functionality.

3.2.1 Metadata Construction

Earlier, metadata construction could only be done from scratch. The support for the Construct mode from baseline configuration was not available for ComOptB. This required that the users had to enter all the metadata details manually every time. We have added the functionality of using a baseline configuration for metadata construction. The users can also edit and modify the baseline configuration before writing it to the statistics' table.

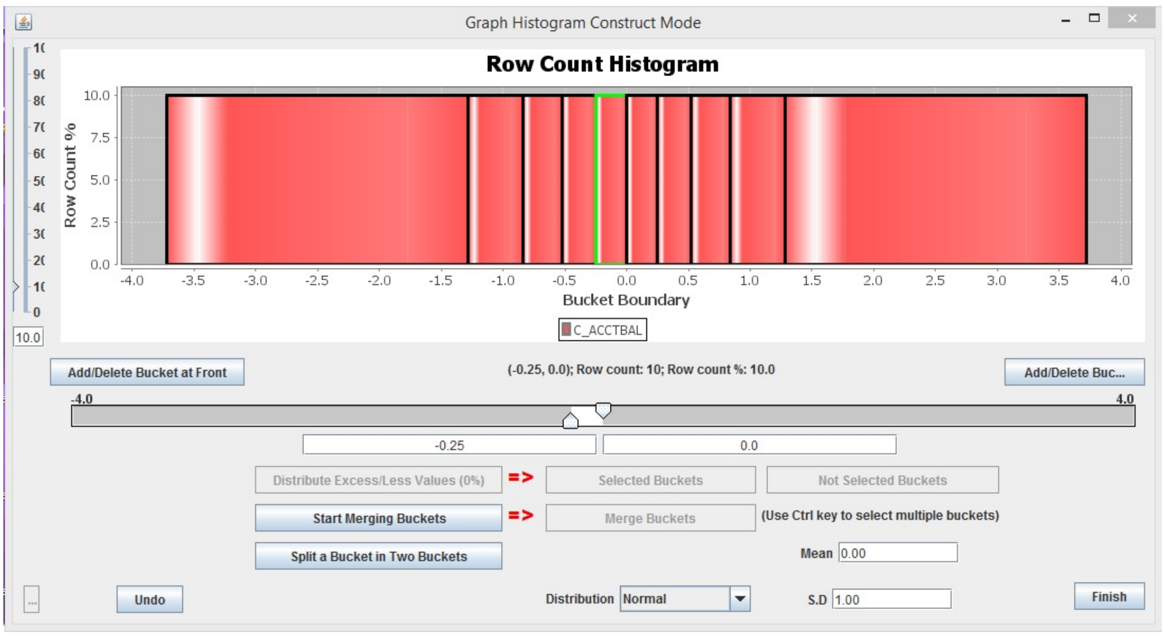
3.2.2 Metadata Transfer

This mode works in conjunction with the Construct mode. Users can store the metadata statistics in a file and use it as a baseline configuration in the Construct mode. This mode was not implemented earlier in ComOptB and was added as part of this work.

In addition, various bugs pertaining to scaling and overflow, and GUI issues pertaining to Graphical Histogram interface were also fixed.

3.3 Standard Distribution Generation

As we saw in Section 2.1, CODD allows a graphical histogram editing interface for the construction of metadata statistics. This could be used to easily create histograms corresponding to the attribute distributions. In addition, the individual bucket geometries could be modified using the mouse drag option. Graphical mode also checks for the validity of the histograms that could be created. For example, Figure 3.1 shows the histogram type permitted by the underlying engine - equi-depth histogram and users will not be able to create for example, equi-width histograms.



(-0.25, 0.0); Row count: 10; Row count %: 10.0

-0.25 0.0

Mean 0.00

Distribution Normal

S.D 1.00

Figure 3.1: GraphicalMode - Depicting Equi-depth and Normal Distribution

While interacting with HP developers, we realized that the support for standard distributions (e.g, Normal, Poisson, Skew) for creation of initial histograms would be an important addition. Thus, as a further aid, CODD also offers the user a pre-defined menu of classical distributions and choosing any of these results in the appropriate histogram being automatically created. Figure 3.1 shows one such histogram distribution, which is normally distributed. It then becomes easier for the end user to generate variations/construct different histograms on this base histogram.

Chapter 4

Plan bouquet robustness guarantees at Big Data Scale

Given an SQL query, current database systems execute it using a least cost plan which is largely based on estimates of predicate selectivity. Due to invalid statistics and assumptions, error in estimates can lead to highly sub-optimal plans. In [7], a novel strategy named "Plan bouquets" has been proposed which provides guarantees on the worst case execution performance which does not rely on the estimates of predicate selectivity. In this section, we verify if plan bouquets' attractive performance guarantees extends to Big Data environments.

4.1 Background

The explanation given in section 4.1 is taken from [8]

4.1.1 Plan Bouquet

Modern database systems estimate predicate selectivity, and search for a least cost plan among several other alternatives which is then executed. Often, the selectivity estimates are significantly erroneous with respect to the actual values subsequently encountered during query execution. Such errors lead to poor execution plan choices by the optimizer, resulting in substantially inflated query response times which can even be in orders of magnitude.

In the efforts to mitigate this problem, [7] proposed a new query processing strategy called "Plan bouquets", which provides upper bounds on worst case execution performance. The basic idea in the bouquet approach is to completely jettison the compile-time estimation process for error prone selectivity. Instead, these selectivities are discovered at run-time through a sequence of cost-limited executions from a small set of plans. Plan bouquet technique works in two phases, namely Bouquet Identification (Compile time) and Bouquet Execution (Execution time). We will focus only on the Bouquet Identification phase, as CODD currently allows us to analyze only the compile time performance guarantees and not the run time guarantees. The metric which is used to quantify the worst case execution performance in Plan bouquet is explained first.

Robustness Metric Given a user query Q with an associated error-prone selectivity space ESS, our notion of robustness is the maximum performance sub-optimality MSO that could occur due to selectivity estimation errors, as compared to an idealized system that magically knows the correct values of all selectivities. Specifically, let q_e denote the optimizer's estimated query location in the ESS, and q_a denote the actual run-time location. Also, denote the plan chosen by the optimizer at q_e by P_{oe}, and the optimal plan at q_a by P_{oa}. Finally let $cost(P_j, q_i)$ represent the execution cost incurred at an arbitrary ESS location q_i by plan P_j. Then, robustness is defined by the normalized metric:

$$\mathsf{MSO}(Q) = \max_{q_e, q_a \in ESS(Q)} [\frac{cost(P_{oe}, q_a)}{cost(P_{oa}, q_a)}] \tag{4.1}$$

which ranges over $[1,\infty)$.

4.1.2 Bouquet Identification

Consider EQ, the simple single predicate join query shown in Figure 4.1 – here, the optimizer needs to estimate the selectivities of one selection predicate (*p_retailprice*) and two join predicates (*part* ⋈ *lineitem*, *lineitem* ⋈ *orders*).

```
select * from lineitem, orders, part
where p_partkey = l_partkey and l_orderkey =
o_orderkey
      and p_retailprice < 1000
```

Figure 4.1: Example Query (EQ)

Let's assume that both the join predicates can be estimated correctly, which results in only one error-prone dimension.

First, through repeated invocations of the optimizer, the "Parametric Optimal Set of Plans" (POSP) that cover the entire selectivity range of the p_retailprice predicate is identified. The cost of these five plans, P1 through P5 are enumerated in Figure 4.2, using abstract plan costing over the range.

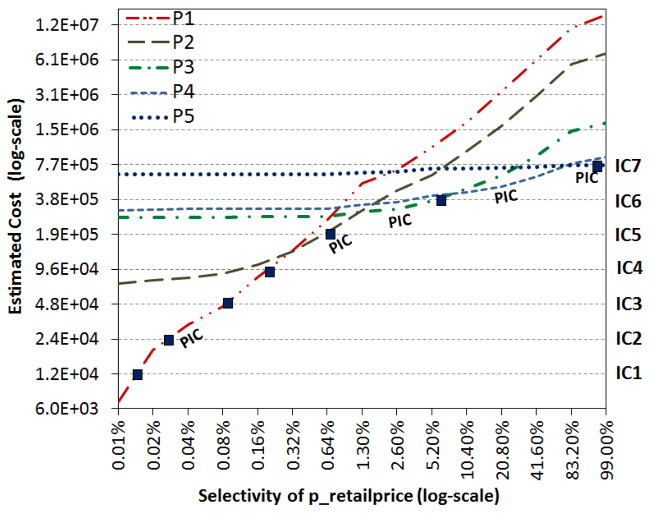


Figure 4.2: POSP performance (log-log scale)

From these plots, they construct the "POSP Infimum Curve" (*PIC*), defined as the trajectory of the minimum cost from among the POSP plans – this curve represents the ideal performance.

The next step, which is a distinctive feature of the bouquet approach, is to *discretize* the PIC by projecting a graded progression of *isocost* (IC) steps onto the curve. For example, in Figure 4.2, the dotted horizontal lines represent a geometric progression of isocost steps, IC1 through IC7, with each step being *double* the preceding value. The intersection of each IC with the PIC (indicated by ■) provides an associated selectivity, along with the identity of the best POSP plan for this selectivity. For example, in Figure 4.2, the intersection of IC5 with the PIC corresponds to a selectivity of 0.65% with associated POSP plan P2. The subset of POSP plans that are associated with the intersections forms the "plan bouquet" for the given query – in Figure 4.2, the bouquet consists of {P1, P2, P3, P5}.

This could easily be generalized to multi-dimensional ESS. In that case, the IC steps and PIC curve become surfaces, and their intersections represent selectivity surfaces on which multiple bouquet plans may be present. The basic mechanics of the bouquet algorithm remain very similar to the 1D case, but the primary difference is that they jump from one IC surface to the next only after it is determined (either explicitly or implicitly) that none of the bouquet plans present on the current IC surface can completely execute the given query within the associated cost budget. The complete algorithmic details are available in [7].

In [7], the bound on *MSO* is given by

$$MSO \leq 4 * \rho$$

where ρ is the maximum number of plans in a contour over all iso-cost contours in the ESS. To empirically reduce the value of ρ, a technique called "Anorexic reduction"[9] is used in the paper.

4.1.3 QUEST

QUEST (QUery Execution without Selectivity esTimation) is the prototype implementation of plan bouquet concept on the PostgreSQL engine. QUEST uses the Picasso plan and reduced plan diagrams to identify the bouquet of plans. Given a query point in the selectivity space, Picasso estimates the constants that would result in the desired selectivities of the base relations by essentially carrying out an "inverse-transform" of the statistical summaries corresponding to the Picasso columns.

The plan diagrams for PostgreSQL are generated by using the *histogram_bounds* metadata statistics. For each $\leq$ predicate, the *histogram_bounds* bucket boundaries are read from the statistics and linearly interpolated from the lowest histogram boundary. This gives the constants that would in turn help in estimating the cost at each query point in the selectivity space. Please note that the resolution and type of distribution (Uniform or Exponential) determines the number of query points. For example, in a 2-D plan diagram with a uniform grid resolution of 100, there are 10000 query points.

4.2 Guarantees at Big data scale

We focus on the pre-processing step of Plan bouquet and analyse how ρ value in the MSO varies with scale. Figure 4.3 shows an example 2-D Picasso plan diagram for parameterized SQL query template, QT5, based on Query 5 of TPC-H [10] benchmark.

We will consider the QT5 as the running example in this section. We were faced with the following technical issues while testing for scaled data.

The ESS is explored using a sequence of cost-limited partial executions, beginning with the cheapest isocost step (*c_min*). For example, *c_min* for the query in Figure 4.3 on 1GB data scale is 54. This corresponds to 0.1% selectivity at 100 resolution with Exponential distribution. This cost is based on the number of tuples, flowing through each node in a query plan. The number of tuples in this case was 30. Now if we scale it to $1TB$ data scale using CODD, the number of tuples becomes 30000 and the cost also significantly increases to around $5.0E4$. This is depicted in Figure 4.4.

```
select
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from customer, orders, lineitem,
        supplier, nation, region
where c_custkey = o_custkey and l_orderkey = o_orderkey
and l_suppkey = s_suppkey and c_nationkey = s_nationkey
and s_nationkey = n_nationkey and n_regionkey = r_regionkey
and r_name = 'ASIA' and o_orderdate >= '1994-01-01'
and o_orderdate < '1995-01-01' and c_acctbal <= 5400
and s_acctbal :varies and l_extendedprice :varies
group by n_name
order by revenue desc
```

Figure 4.3: QT5

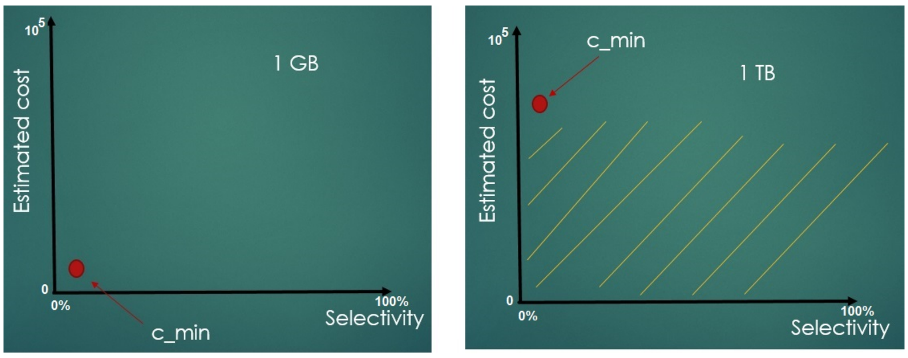


Figure 4.4: *c_min* at Big data scale

Now in the scaled version, *c_min* is too high that a significant portion of the ESS (the shaded region in Figure 4.4) is not being explored. This results in contours present in that portion not being considered for exploration by the QUEST implementation. Hence, we need a way to bring the *c_min* down so that the entire ESS is explored and we can analyze the performance guarantees at Big Data scale. We look at the different approaches considered to solve this problem.

4.2.1 Higher Resolution

One simple approach to reduce *c_min* is to generate the Picasso plan diagrams at a higher resolution (say 1000 resolution). With higher resolution, the selectivity is much lower (say 0.001%) and the constant estimate by Picasso corresponding to the selectivity should be much lower. This means that the number of tuples and correspondingly the cost(*c_min*) should be much less, so that we would be able to explore the entire space. But, this was not the case as there is a major difference between the Picasso estimated selectivity and the actual predicate selectivity estimated by the optimizer for PostgreSQL, at Big Data scale.

We will explain this with an example. The *histogram_bounds* for the column *s_acctbal* of PostgreSQL are : [922,1063, ... , 11553] - 100 histogram buckets, the values are assumed to be uniformly distributed within a bucket. These boundary values remain the same for both $1GB$ and $1TB$. Now, Picasso does linear interpolation from the lowest histogram boundary (922 in our example); the constant value predicted by Picasso for the resolution 100 (or selectivity 0.1%) and 1000 (or selectivity 0.001%) at $1TB$ scale are 922.68 and 922.02 respectively. This does not help in bringing the *c_min* down since the number of tuples in both the cases are almost the same ($\sim$ 30000). We will still end up with the inflated *c_min* for the scaled data. This means that generating plan diagrams at higher resolution does not help us in bringing the *c_min* down.

4.2.2 Add Histogram Bucket

We tried to address this issue by adding a new histogram bucket (typically one is enough) with a histogram boundary of say, 900 (This value should be less than or equal to the minimum value of the column for a significant reduction in the *c_min*. This requirement would be clear when we explain the next approach in section 4.2.3.) Now, this brought the *c_min* down considerably and we were able to explore the ESS. But there were inherent flaws in this approach.

- Adding histogram bucket destroys the equi-depth property of the PostgreSQL statistics.
- Optimizer plan choices may differ given that we have manufactured statistics by adding a new bucket.

4.2.3 Use Minimum

We looked into how PostgreSQL estimates selectivities for the predicate *s_acctbal* $\leq$ 922 or $\leq$ 910 with the lowest histogram boundary value being 922. Whenever PostgreSQL has to compare with the lowest histogram boundary or below that, it finds the minimum value of the column and uses that as the lowest histogram boundary. In our example, if 908 is the minimum value of the column, then the lowest bucket boundary is taken as 908 and the original boundary value, 922, is subsumed by this bucket[1]. The number of tuples that is returned for the minimum value is 1 and the predicate with the minimum value will result in the lowest cost for the query.

[1]Please note that the statistics are not changed by PostgreSQL; this is achieved only programmatically

We implemented the logic of fetching the minimum value when the Picasso selectivity differs from the predicate selectivity and then start the linear interpolation from that so that the c_min is low enough to allow us to explore ESS completely. Since we get to explore the entire ESS, we can now analyze the plan bouquet guarantees at Big Data scale.

Table 4.1 lists the results of the experiments run for incrementally scaled Big Data scenarios.

Database Size	No. of Contours	PostgreSQL MSO	Plan Bouquet MSO
1GB	7	12	2x10^3
10GB	10	12	4x10^3
100GB	12	16	5x10^3
1TB	16	16	2.5x10^4
10TB	18	16	3x10^5

Table 4.1: Plan Bouquet guarantees

As can be seen, plan bouquet approach continues to provide low MSO values at Big Data scales - for example, on a 10TB database, its MSO is only 16. In contrast, the native PostgreSQL engine's performance becomes progressively worse with increasing database size, reaching an MSO of 300000 at the 10 TB scale! This proves that the plan bouquets' approach is robust against Big Data. Thus, CODD enabled us to experimentally verify the plan bouquets' performance guarantees at Big Data scale.

4.3 Repeatable executions of QUEST

Apart from robustness, another unique benefit of the bouquet mechanism's choosing to abjure selectivity estimation is that query execution strategies are *repeatable* across different invocations, that is, even if the distribution histograms of the relation columns are highly erroneous with regard to the underlying data, the plan bouquet approach will provide exactly the same performance as that obtained when the system had accurate histograms, making it particularly attractive in industrial applications.

CODD can be used to prove the above claim by *radically altering* the distribution histograms of the attributes featured in the query, while keeping the underlying data *unchanged*. Subsequent to the alteration, when the bouquet algorithm is re-executed, it is verified to behave identically to its prior incarnation.

Chapter 5

Hardware Configuration For Hardware-based Testing

The query optimizer can choose a better execution plan if the system's hardware details, viz, I/O or CPU performance or utilization are taken into account. These statistics are known to be the hardware statistics. While choosing a plan, the query optimizer uses these hardware statistics to estimate the CPU resources or the I/O resources required by the query.

We have added the capability of simulating futuristic hardware for ComOptA. The following are the two different types of hardware statistics that are modeled for ComOptA.

- Gather System statistics: Gather system (or workload) statistics refers to the statistics that are collected when the system activity is high for a certain period of time. For example, if the system is I/O bound, it will be captured in the statistics and the optimizer will choose a less I/O intensive plan after the statistics are used. Table 5.1 gives parameters [11] that are modeled with respect to gather system statistics.

- Noworkload statistics: Noworkload statistics refers to the statistics that are gathered by submitting random reads to data. Table 5.2 gives parameters [11] that are modeled with respect to noworkload statistics.

In addition we also modeled the parameter *parallel_max_servers*, which specifies the number of parallel execution processes. This parameter would let the optimizer create a plan for *that* degree of parallelism.

Parameter	**Description**
cpuspeed (MIPS)	Represents workload CPU speed. CPU speed is the average number of CPU cycles in each second.
sreadtim (ms)	Single block read time is the average time to read a single block randomly.
mreadtim (ms)	Multiblock read is the average time to read a multiblock sequentially.
mbrc (blocks)	Multiblock count is the average multiblock read count sequentially.
maxthr (bytes/s)	Maximum I/O throughput is the maximum throughput that the I/O subsystem can deliver.
slavethr (bytes/s)	Slave I/O throughput is the average parallel slave I/O throughput.

Table 5.1: Gather System Statistics

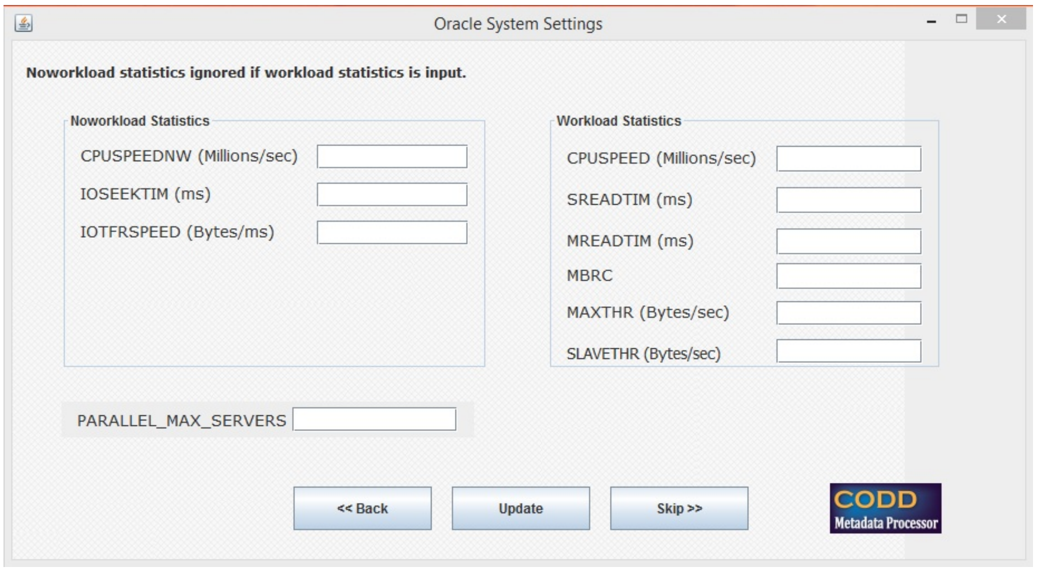
Parameter	**Description**
cpuspeedNW (MIPS)	Represents noworkload CPU speed. CPU speed is the average number of CPU cycles in each second.
ioseektim (ms)	I/O seek time equals seek time + latency time + operating system overhead time.
iotfrspeed (bytes/ms)	I/O transfer speed is the rate at which an ComOptA can read data in the single read request.

Table 5.2: Noworkload Statistics

5.1 Implementation Details

CODD is now extended for simulation of hardware requirements for ComOptA. Once the user enters the database connection details, he is redirected to a new page where he has the option to input the hardware parameters as shown in Figure 5.1.

Users have the option to enter both workload and noworkload statistics but ComOptA will ignore the noworkload statistics and use the workload statistics if it is present. Once he enters the data and clicks on the Update button, input validation is done and the user is directed to the usual flow of CODD. After inputting the other metadata statistics in Construct Mode GUI, the user clicks on the Construct button; the hardware statistics along with the other metadata details are then written to the database catalogs. Users also have the option of skipping the hardware statistics settings by pressing the Skip button in the Hardware Settings GUI.


Oracle System Settings

Noworkload statistics ignored if workload statistics is input.

Noworkload Statistics

CPUSPEEDNW (Millions/sec)	
IOSEEKTIM (ms)	
IOTFRSPEED (Bytes/ms)	

Workload Statistics

CPUSPEED (Millions/sec)	
SREADTIM (ms)	
MREADTIM (ms)	
MBRC	
MAXTHR (Bytes/sec)	
SLAVETHR (Bytes/sec)	

PARALLEL_MAX_SERVERS

Figure 5.1: ComOptA Hardware Statistics

5.2 Experimental Results

5.2.1 Testing Environment

- Database: 100-PetaByte (10^{17} Bytes) TPC-H dataless database, generated using CODD.
- Computational Platform: Windows 8 Operating System, Intel Core i5 CPU 2.60 GHz, 6GB RAM.

The default system settings used were:

- CPUSPEED = 2x10^3 MIPS
- MBRC = 64
- SREADTIM = 5ms
- MREADTIM = 10ms

The futuristic hardware settings used were:

- CPUSPEED = 2x10^5 MIPS

- MBRC = 64x10^2
- SREADTIM = 1ms
- MREADTIM = 2ms

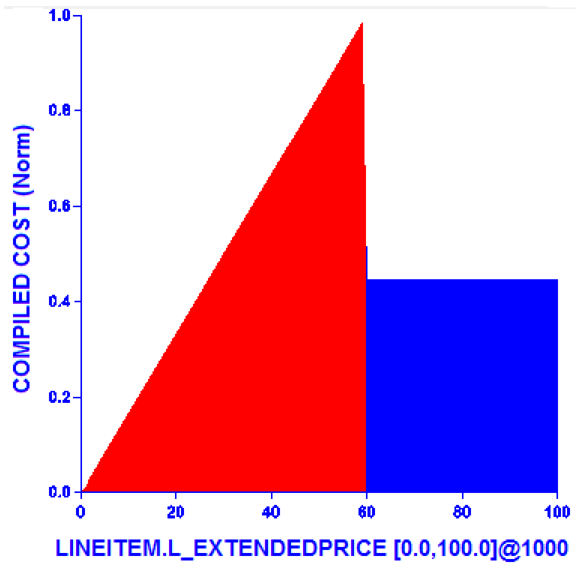
We ran an explain query for a full table scan of the *lineitem* table, with both the system default settings and the futuristic hardware setting and found the results in Table 5.3. The optimizer estimated only one-fifth of the time for the futuristic configuration.

Configuration	**CPU Time**
System Default	1000 Hours
Futuristic Hardware	200 Hours

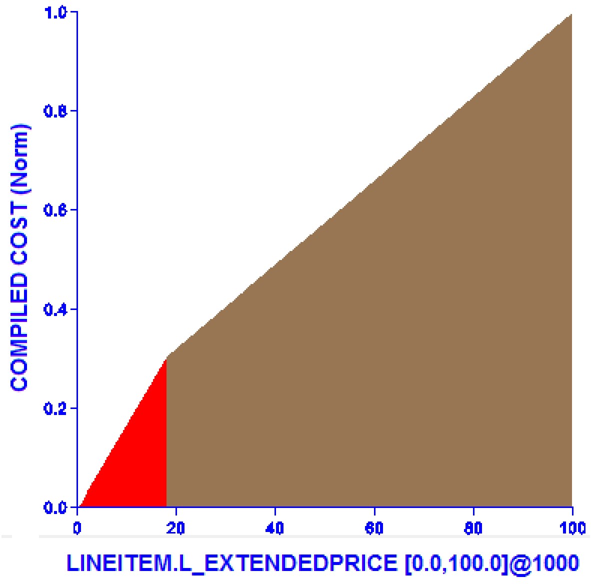
Table 5.3: Futuristic Hardware Configuration

We ran the benchmark QT5 template on a *100 petabyte shell* (only the cost variation on the *lineitem* selectivity is shown for better readability) with both the default settings and with the futuristic hardware settings.

Figure 5.2 shows the Picasso compilation cost diagram generated with the default system and futuristic hardware settings.



(a) 100 PB / 1 Server



(b) 100 PB / x100 Scaleup Server

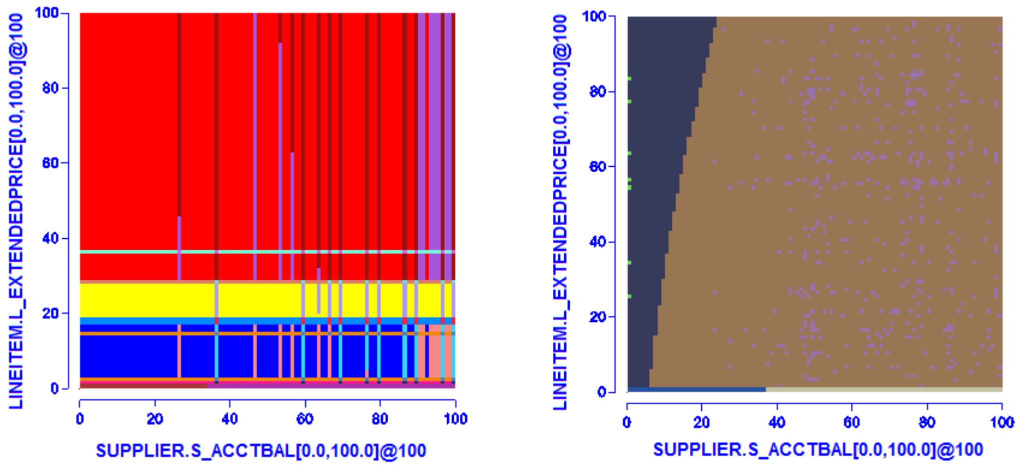
Figure 5.2: Cost Diagrams for QT5 (ComOptA)

The following are the findings of the experiments conducted by simulating the hardware environments:

- It can be seen from Figure 5.2(a) that the query execution costs initially rise with the selectivity, as expected. However, at a selectivity of 60 percent, there is a sudden dip followed by saturation of the query execution cost, violating the *cost-monotonic* behavior that typically holds for such queries. The primary change is a switch from a nested-loops join of *lineitem* and *supplier* in the red plan to a sort-merge join in the blue plan.

- Interestingly, this issue disappears when the processor speed is scaled up by a factor of 100, as shown in Figure 5.2(b). These results suggest the presence of underlying issues in the construction of ComOptA's cost estimation module.

We analyzed the effect of *parallel_max_servers* parameter by setting it to a futuristic value of 3000, indicating that the optimizer could generate a plan that requires 3000 parallel execution slaves. There is a significant difference in the number of plans while using parallelism (we used parallelism for *lineitem* table) and this difference becomes more prevalent at Big Data scale. Figure 5.3 shows the plan diagram generated for QT5 at 1PetaByte scale.



(a) 1 PB / 1 Server (b) 1 PB / 3000 Servers

Figure 5.3: Plan Diagrams for QT5 (ComOptA)

We observe that the number of plans has reduced drastically from 20 to 6, and the geometries of the plan optimality regions have undergone significant changes. Moreover, the plans themselves are quite different between the two figures. CODD could hence be used to simulate different futuristic scenarios easily to identify the different hardware configurations that would suit the end user.

Chapter 6

Design Bugs at Big Data scale

CODD has also proved to be effective at identifying bugs that surface at Big Data scale. We present two such instances here:

- By iteratively executing CODD with QT5 on ComOptA, with the database size increasing in each iteration, we were able to quickly discover that ComOptA's cardinality estimation module saturated when the input data size to an operator exceeded 20 exabytes - specifically, the output cardinality of the operator became frozen at approximately 18.4 exabytes, no matter how large the input was beyond this point! This is visually shown in the (partial) plan tree of Figure 5, where the circled regions highlight the erroneous behavior.

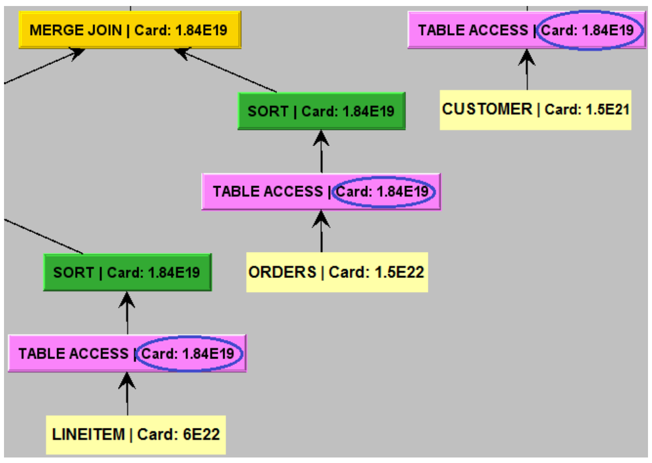

MERGE JOIN | Card: 1.84E19
TABLE ACCESS | Card: 1.84E19
SORT | Card: 1.84E19
CUSTOMER | Card: 1.5E21
TABLE ACCESS | Card: 1.84E19
SORT | Card: 1.84E19
ORDERS | Card: 1.5E22
TABLE ACCESS | Card: 1.84E19
LINEITEM | Card: 6E22

Figure 6.1: Cardinality Error in ComOptA

- In section 5.2, we saw that, in ComOptA, the query execution cost violates the *cost-monotonic* behavior at high selectivity. We verified that this anomaly only shows up at

100-PetaByte scale and not below that data scale.

- After scaling the benchmark database beyond 1TB using CODD tool, the minimum cardinality estimation by the Picasso tool always resulted in very high values. This resulted in the fact that the compilation time cardinality diagram for certain queries had a flat plane, i.e., the minimum and the maximum cardinality estimation were almost the same. When we investigated the issue, we found that Picasso's selectivity estimates were different from the actual predicate selectivity estimated by the optimizer. As a case in point, we saw in section 4 that we were not able to explore the ESS completely because of the way Picasso estimated selectivity. This shows that CODD could be effective in finding bugs not only in database engines, but also in other tools that interact with them.

Chapter 7

Execution Time Testing

While it is not necessary to have the data present in the system for doing the compilation time testing, it is required in the execution phase of the database engine. The issues with using real data for testing are the following:

- Transferring Big Data could be a laborious task.
- Clients may not be willing to provide the data, due to privacy concerns.

Thus it is required to be able to do execution time testing without access to the original data. In this part of work we focus on the generation of a synthetic database using ICA (Independent Component Analysis) based technique, such that the generated database has similar distributions to the original database.

7.1 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 $R_i \in D$.

7.2 Assumptions for the suggested solution:

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

- There exists no NULL values in the relation,
- The data present in relations is of numeric datatype,
- Underlying data distribution is Non-Gaussian.

7.3 Synthetic Database Generation

Synthetic data generation involves two steps:

- *Data Summarization:* consolidating or summarizing the actual data into a metadata form, which is very small.
- *Data Generation:* Method to generate data from the consolidated information in *data summarization* step.

7.3.1 Data Summarization

We propose the Independent Component Analysis (ICA) method as the *data summarization* step of the synthetic data generation. Before we take a look at ICA, we will look at why this method is necessary.

Data Correlation: For uncorrelated data, there have been traditional methods in the form of 1D histograms, to summarize the data. These 1D histograms could then be used in the *data generation* step. But we cannot use the 1D histogram technique with correlated data since, individually generating data for each column would result in inaccurate data generation. To explain this, consider the case of Salary and Tax where both are correlated. When Salary increases, Tax should also increase; but if we have 1D histograms for both Salary and Tax, and if we generate them individually, we may end up with lower Tax for a higher Salary.

So, if we can *de-correlate* the data, we can use the traditional 1D histogram methods to summarize the data. De-correlating typically involves seeking basis vectors that the data is projected against, such that you maximize *criteria* viz *variance*, *kurtosis*, etc [12]. We will look at which methods help us in de-correlating the data.

We will denote the database relation **R** as a data matrix, where rows denote the number of observations and each column is a random variable. Let's say the data matrix **D** is a **2xN** matrix, i.e., there are two random variables and N observations of each. We find the basis vector

$$\mathbf{s} = \begin{bmatrix} 0.4 \\ -2 \end{bmatrix} \tag{7.1}$$

Now, when the first component is extracted, say $\mathbf{y}$, it is given as: $\mathbf{y} = \mathbf{s}^T\mathbf{D}$. This means, "Multiply 0.4 by the first row of data, and subtract twice the second row of data". $\mathbf{y}$ is thus a **1xN** vector that has the property that it maximized specific *criteria*.

7.3.2 Different Criteria

PCA Criteria: *Principal Component Analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components* [13]. In PCA, we find the basis vectors that best explain the *variance* of the data. The first basis vector is chosen in such a way that it best fits all the variance from the data; the second vector also has this criteria, but must be orthogonal to the first vector and so on. Apparently, these vectors are the eigenvectors of the data's covariance matrix. In statistical terms, PCA is based on second-order statistics.

ICA Criteria: ICA also involves finding the basis vectors, but requires that the resulting vector is one of the *independent components* of the data. This involves maximization of the absolute value of normalized *kurtosis*, a fourth order statistic. This is actually done by projecting the data on to some basis vector and measuring the kurtosis; then basis vector is changed and the kurtosis is measured again and so on. Eventually, we would end up with a resultant vector that has the highest kurtosis, and that is our independent component.

Why use ICA? PCA finds the most uncorrelated components, i.e.,correlation close to *zero*, to explain the maximum variation in data. In case of multivariate Gaussian distributions, if the components are uncorrelated, then they are independent [14]. Thus PCA efficiently represents multivariate Gaussian distributed data. But for non-Gaussian distributions, uncorrelated components don't necessarily mean independence. Thus, when the underlying data is not Gaussian, higher order statistics beyond variance needs to be considered for finding independent components. ICA provides a transformation of data such that the columns of the relation are statistically independent of each other.

7.4 Independent Component Analysis

The details of this section are taken from [15]. We use the following ICA model

$$\mathbf{x} = \mathbf{A}\mathbf{s}^{1} \tag{7.2}$$

where $\mathbf{x}$ is the input/observed random vector, $\mathbf{A}$ is the mixing matrix and $\mathbf{s}$ is the required output random vector. Using the observed random vector $\mathbf{x}$, we must estimate both $\mathbf{A}$ and $\mathbf{s}$. We first estimate the matrix $\mathbf{A}$, compute its inverse, say $\mathbf{W}$, and obtain the independent component by:

$$\mathbf{s} = \mathbf{W}\mathbf{x} \tag{7.3}$$

We now define the concept of independence used by ICA estimator.

Independence: Two scalar valued random variables y_1 and y_2 are said to be independent if information on the value of y_1 does not reveal any information on the the value of y_2 and vice versa. Technically, this is defined by the joint and marginal probability densities. If $p(y_1,y_2)$ denote the joint probability density function (pdf) of y_1 and y_2 and $p_1(y_1)$ and $p_2(y_2)$ denote the marginal probability densities of y_1 and y_2 respectively, then y_1 and y_2 are independent iff the joint pdf is factorizable as:

$$p(y_1,y_2) = p_1(y_1)p_2(y_2) \tag{7.4}$$

7.4.1 FastICA

The key to estimating the ICA model is non-Gaussianity. Maximizing the non-Gaussianity of $\mathbf{w}^T\mathbf{x}$ in (4) gives us one of the independent components. We refer the reader to [15] for the detailed explanation and the proof of this statement. We will be using the FastICA algorithm to find the independent components of our ICA model.

FastICA is an efficient and popular algorithm for ICA based on a fixed-point iteration scheme maximizing non-Gaussianity as a measure of statistical independence. See Algorithm 1 [16].

FastICA algorithm requires that the data is pre-processed by *centering* and *whitening* the data. Centering the data involves making the data have zero mean. Whitening the data involves transforming the data so that the new components are uncorrelated and have variance *one*.

The iterative algorithm then finds the direction for the weight vector $\mathbf{w}$ maximizing the non-

[1]Notation: bold lower-case indicate vectors and bold upper-case letters denote matrices.

Algorithm 1 FastICA

Require:

Input:

- Number of desired components C.
- Matrix $\mathbf{X} \in \mathbb{R}^{N \times M}$, where each column represents an N-dimensional sample, where $C < N$.

Output:

- Un-mixing matrix $\mathbf{W} \in \mathbb{R}^{C \times N}$, where each row projects X onto independent components.
- Independent components matrix $\mathbf{S} \in \mathbb{R}^{C \times M}$, with M columns representing a sample with C dimensions.

```
1: for $p$ in 1 to $C$ do
2:     $\mathbf{w_p} \leftarrow$ Random vector of length $N$
3:     while $\mathbf{w_p}$ changes do
4:         $\mathbf{w_p} \leftarrow \frac{1}{M}\mathbf{X}g(\mathbf{w_p}^T\mathbf{X})^T - \frac{1}{M}g'(\mathbf{w_p}^T\mathbf{X})\mathbf{1}\mathbf{w_p}$
5:         $\mathbf{w_p} \leftarrow \mathbf{w_p} - \sum_{j=1}^{p-1} \mathbf{w_p}^T\mathbf{w_j}\mathbf{w_j}$
6:         $\mathbf{w_p} \leftarrow \frac{\mathbf{w_p}}{\|\mathbf{w_p}\|}$
7:     end while
8: end for
```

Gaussianity of the projection $\mathbf{w}^T\mathbf{x}$ for the data $\mathbf{x}$. This is repeated until we find C independent components as given in Algorithm 1.

Once we have the output matrix S with all the independent components, we can generate the 1D histograms from S. This completes the *data summarization* step. We can then generate the synthetic relation $\bar{R}_{ICA}$ (in transformed space), by generating data for each column independently using those 1D histograms.

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

$$\bar{R} = \bar{R}_{ICA}.W^{-1}$$

Here we take the pseudo-inverse of W. This completes the *data generation* step.

7.5 Experimental Results

To test the performance of our algorithm, we took the *store_returns* relation of TPC-DS [17] database as input, call it R, and then generated synthetic relation R^{ICA} using our ICA based algorithm. We also generated synthetic relation R^{PCA} using previously proposed PCA based

Algorithm 2 Generating the synthetic relation:

```
• Generate the synthetic relation $\bar{R}_{ICA}$ (in transformed space), using 1D histograms.
• Calculate the synthetic relation $\bar{R}$ (in actual space), using the following inverse transformation:
      $\bar{R} = \bar{R}_{ICA}.W^{-1}$
• Do the inverse process of whitening and centering.
• Return $\bar{R}$.
```

algorithm [4]. We compared the efficiency of the algorithm by generating Picasso plan diagrams for the following self-join query with selectivity variation on *sr_return_amt* and *sr_return_tax* attributes of *store_returns* relation, which are highly correlated. We provide a self-join query example because we are more interested in knowing how far the distribution characteristics are retained within a relation.

```
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 100 and the distribution of the query was set to uniform. These diagrams were generated on the PostgreSQL database optimizer. Figure 7.1 gives the comparison between the original relation, R, synthetic relation using ICA, R^{ICA} and synthetic relation using PCA, R^{PCA}.

Plan Diagram: R had 7 plans, R^{ICA} had 10 plans and R^{PCA} had 6 plans. From the figure, it is clear that the synthetic relation generated using ICA had similar geometries to the original relation. Synthetic relation generated using PCA differed a lot with respect to the geometries of the original relation.

Compilation Time Cardinality Diagram: The estimated cardinality for R, R^{ICA} and R^{PCA} were almost the same.

Compilation Time Cost Diagram: The cost estimates of R and R^{ICA} were comparable except for the low selectivities. The difference in cost estimates for low selectivities could be because ICA is sensitive to outliers and hence the bucket boundary difference for low values could be less than for R.

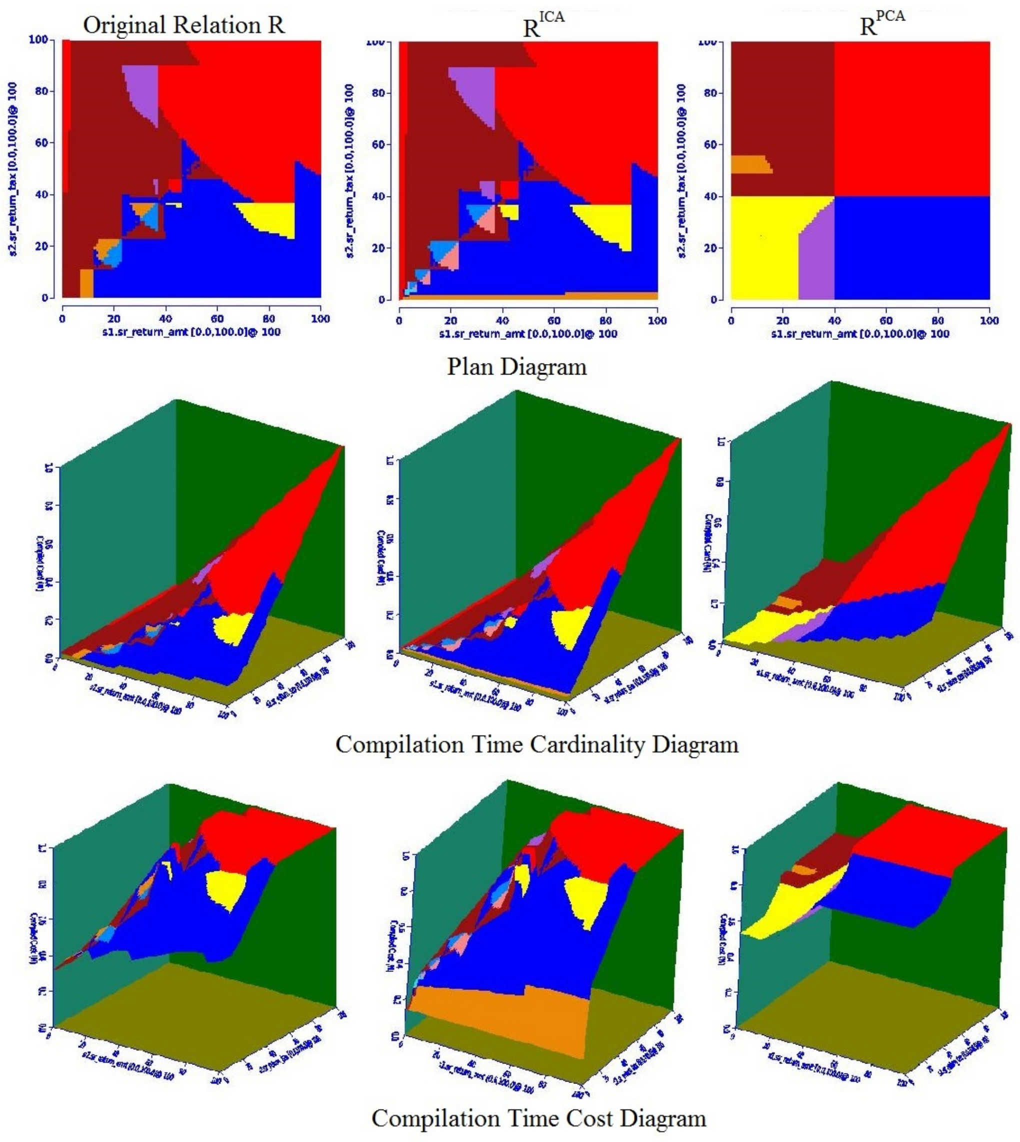


Figure 7.1: Picasso Diagrams

We also generated plan diagrams for comparing join across relations. Figure 7.2 gives the plan diagram generated for the following join query. The relations generated are *catalog_sales* and *store_sales* of TPC-DS benchmark database.

```
SELECT * FROM store_sales, catalog_sales
WHERE sr_item_sk = cs_item_sk
AND ss_list_price :varies
AND cs_list_price :varies
```

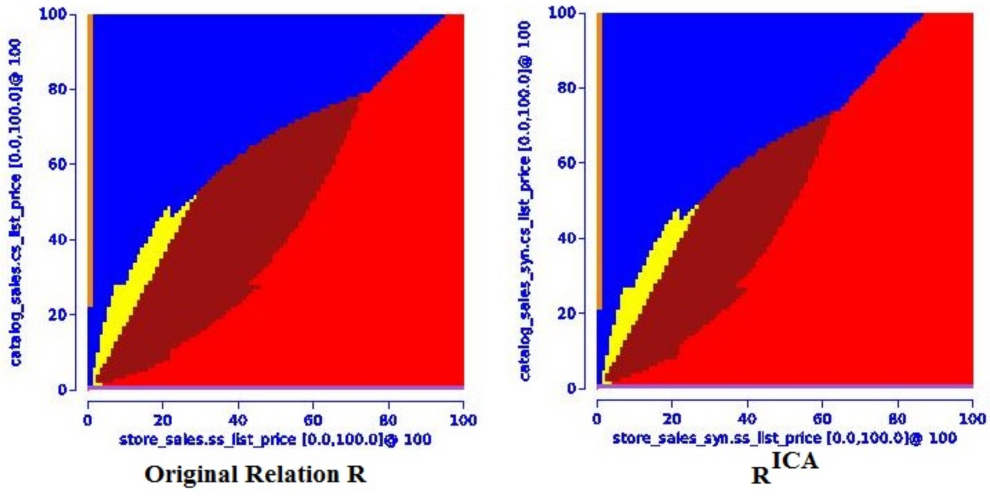


Figure 7.2: Plan Diagram for Join across relation

As can be seen, the synthetic relation generated using ICA has similar geometries to the original relation. We have omitted the cardinality and cost diagram because they were almost identical to the original relation.

These experiments show that ICA based algorithm prove to be a potential way to generate the data synthetically that models the original data very closely.

Chapter 8

Coding Efforts in New Features

- **Picasso and CODD:** Around 2K lines of new code was added and 2K lines of existing code modified.
- **PostgreSQL:** Around 1K lines of new code was added to PostgreSQL code base.
- **Synthetic Data Generation:** Around 1K lines of code.

Chapter 9

Conclusions and Future Work

In this work, we engineered CODD for PostgreSQL and verified that the Plan Bouquet's robustness guarantees hold at Big Data scale. We also showcased simulating futuristic hardware configuration for ComOptA. Our experimental results showed that hardware configuration could significantly impact the plan choices by the optimizer. We then proposed ICA based approach to solve the problem of synthetic database generation for execution time testing. Our experiments show that ICA algorithm performs well with skewed database as evident from the data generation of relations belonging to TPC-DS benchmark data.

Our future work involves implementing full scale execution time testing by generating the data at query execution time instead of persistently storing the data.

Bibliography

[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. ii, 1

[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. ii, 1

[3] http://dsl.serc.iisc.ernet.in/projects/codd/. 2

[4] Ankur Gupta. Big data testing environments http://dsl.serc.iisc.ernet.in/publications/thesis/ankur.pdf. 4, 9, 30

[5] PostgreSQL. http://www.postgresql.org/. 6

[6] http://www.postgresql.org/docs/9.3/static/index.html. 6, 7, 8

[7] Anshuman Dutt and Jayant R. Haritsa. Plan bouquets: Query processing without selectivity estimation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, SIGMOD '14. 11, 13

[8] Anshuman Dutt, Sumit Neelam, and Jayant R. Haritsa. Quest: An exploratory approach to robust query processing. In *Proceedings of the 40th International Conference on Very Large Data Bases*, PVLDB 7(13) '14. 11

[9] 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. 14

[10] TPC Benchmark H. http://www.tpc.org/tpch. 14

[11] http://docs.oracle.com/cd/b19306_01/server.102/b14211/stats.htm. 18

[12] http://stats.stackexchange.com/questions/35319/what-is-the-relationship-between-independent-component- analysis-and-factor-analy. 26

[13] http://en.wikipedia.org/wiki/principal_component_analysis. 27

[14] http://en.wikipedia.org/wiki/normally_distribut ed_and_uncorrelated_does_not_imply_independent. 27

[15] Aapo Hyvarinen and Erkki Oja. Independent component analysis: Algorithms and applications. In *Neural Networks'00*. 28

[16] http://en.wikipedia.org/wiki/fastica. 28

[17] TPC Benchmark DS. http://www.tpc.org/tpcds. 29