Engineering Dataless Databases

Rakshit S. Trivedi I. Nilavalagan Jayant R. Haritsa

Technical Report
TR-2012-02

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

Abstract

Effective design and testing of database engines and applications is predicated on the ability to easily construct alternative scenarios with regard to the database contents. A limiting factor, however, is that the time and/or space overheads incurred in creating and maintaining these databases may render it infeasible to model the desired scenarios. In this paper, we present **CODD**, a lucid graphical tool that attempts to alleviate these difficulties through the construction of "dataless databases". Specifically, CODD implements a unified visual interface through which databases with the desired meta-data characteristics can be efficiently simulated without persistently generating and/or storing their contents. Metadata validation is incorporated to ensure that the simulated database is both legal and consistent. CODD is currently operational on a rich suite of popular database engines, and introduces two additional facets of relevance to test teams: First, it supports a cost-based database scaling model, in addition to the size-based scaling models that have long been in vogue. Second, it provides for largely automated meta-data transfer across different engines, thereby facilitating the comparative study of systems. We showcase here the ability of CODD to elegantly simulate a variety of testing scenarios ranging from legacy applications to BigData environments.

1 Introduction

The effective design and testing of database engines and applications is predicated on the ability to easily evaluate a variety of alternative scenarios that exercise different segments of the codebase or profile module behavior over a range of parameters [13, 9, 14]. A limiting factor is that the time and space overheads incurred in creating or maintaining these databases may render it infeasible to model the desired scenarios. However, there exists a class of important functionalities, such as query plan generators, system monitoring tools, and schema advisory modules (e.g. StatAdvisor [4]), for which the inputs are comprised solely of the *meta-data*, derived from the underlying database system. For such functionalities, developing software that creates meta-data without either having to generate, or maintain, the associated raw data would be extremely useful. In this paper, we present a graphical tool, called **CODD**, that supports both (a) the *ab initio* creation of metadata, and (b) the reclamation of the space occupied by an existing database *without* impacting its meta-data, inclusive of indexes.

To make the above concrete, consider the situation where a query optimizer developer wishes to evaluate a futuristic Big-Data setup featuring *yottabyte* (10^{24}) sized relational tables. Obviously, just generating this data, let alone storing it, is practically infeasible even on high-end systems. However, with CODD, the associated metadata can be easily constructed within a matter of minutes, including defining the desired attribute-value distributions through visual histogram constructions. Further, CODD incorporates a graph-based model of the structures and dependencies of metadata values, implementing a topological-sort based checking algorithm to ensure that the metadata values are both *legal* (valid range, correct type) and *consistent* (compatible with the other meta-data values).

CODD's ability to model essentially arbitrary database scenarios also comes in handy for debugging legacy applications, or identifying hidden constraints in database engine code. As a case in point, by iteratively executing CODD on a popular commercial query optimizer, with the database size increasing in each iteration, we quickly discovered that the cardinality estimation module "saturated" when the input data size exceeded 10^{19} bytes – no mention of this threshold was found in the publicly available documentation of the system.

Even for environments in which meta-data information is required to be sourced only from the original data itself, CODD makes it feasible to subsequently *drop* the raw data in a manner that is totally *opaque* to the meta-data, including information related to physical schema constructs such as indexes. This facilitates testers to temporarily load real-world database scenarios without having to incur the storage and maintenance overheads of retaining the data during the ensuing testing process. The ability to retain the physical schema as-is, in spite of the data removal, is an important semantic difference as compared to the data truncation facilities natively provided by database engines.

Another special feature of CODD is its support for *automated scaling* of meta-data instances, obviating the need for loading fresh data or meta-data to test variants of datasets. For example, assume that the meta-data for the baseline 100 GB TPC-DS benchmark is available, and we now wish to boost it to the benchmark's maximum size, namely 100 TB. This objective is easily achieved in CODD through the incorporation of size-based scaling models that mimic the TPC-H [19] and TPC-DS [20] data generators.

As a novel addition to the above, CODD provides *time-based* scaling models – here, the objective is to scale the meta-data such that the overall estimated execution time of a test query workload is scaled by a user-specified factor. Our approach first models the optimizer's plan costs for the query workload as functions of the scaling factors of the relations featuring in the queries. Then, we compute an inverse minimization function to determine a suitable choice of relation scaling factors oriented towards producing the desired time scaling. We expect that this feature could serve as a potent complement to the prevalent space-based scaling techniques since it ensures compliance of testing overheads with time budget constraints.

Finally, to facilitate comparative studies of different systems, CODD supports, to the extent possible, the *automated porting* of meta-data across database engines. Specifically, a tester can export most of the meta-data of a given database engine in a format that is compatible with the import interface of another engine, and explicitly input only the engine-specific idiosyncratic information. Another useful application of this feature is that it can be employed to assess, in advance, the potential impact of a *data migration* exercise without having to load the data on the target engine.

In a nutshell, CODD is an easy-to-use graphical tool for the automated creation, verification, retention, scaling and porting of database meta-data configurations. It is completely written in Java, running to over 10K lines of code, and is operational on a rich suite of industrial-strength database systems including DB2, Oracle, SQLServer, Sybase and PostgreSQL. For the commercial engines, it functions solely through the database APIs in a non-invasive manner, while for PostgreSQL, a few extra functions have been incorporated in the engine. Further, a conscious attempt has been made to design the interface such that the user can focus only on the logical meta-data semantics, and not have to contend with understanding the implementation specifics of individual engines. The tool is freely downloadable at [16].

While the current implementation focuses on meta-data based testing, our long-term goals extend to the evaluation of *execution-based* modules also. In particular, we have begun investigating mechanisms for coupling CODD with on-the-fly synthetic data generation approaches (e.g. [10, 1]) such that statistically consistent data is output from plan operators without persistently storing their inputs. Once this coupling is achieved, CODD will be able to holistically mimic, in a dataless manner, testing setups ranging from legacy applications to futuristic Big-Data environments.

1.1 Related Work

The whole process of environment setup includes physical design creation, data generation, data loading and statistics collection. Recent works have concentrated towards improving the efficiency of the data generation tools. Furthermore, the native data loading utilities (e.g. [23, 24]) that ship with commercial database engines have evolved to provide efficient data loading techniques. There are several other external utilities [22, 25] that aim to provide these features in a unified way for all the database engines. But even with the presence of these tools and utilities, the time complexity for setting up Big data environments is very high(in fact, bulk-loading even the comparatively minuscule 100 GB TPC-DS benchmark takes a few days with current database engines on vanilla hardware).

We acknowledge that such setups are currently necessary for execution tasks (this limitation may be overcome with a potential extension to the the tool as described later), but the resource investment seems of very little use for the class of functionalities (e.g. query generators) we have mentioned above. Also, such environments are never physically realizable when space constraints are involved. CODD attempts to alleviate these limitations by enabling a user to create dataless environments in a very short amount of time.

Another related class of utilities that needs attention are the native metadata scripting features (e.g. [7, 30] available with commercial database engines. These utilities provide the ability to *transfer*, as-is, the metadata of an existing database to a new location. Thus they do help to create dataless environments on test machine, thereby simulating the production environment. The major drawback with such utilities is the necessity of the existing setup. They do not allow an *ab initio* creation of metadata shell without using any data. Further, some of the these utilities do not allow updating the metadata values and those that allow it have a very labor-intensive and cumbersome procedure to accomplish it. The utility that can be considered closest to our work is optdiag [8], which exports the statistics in an external file, allows to modify that file and load the statistics back to database catalogs. But even for this utility, the initial setup is necessary. Also, user needs much knowledge of internals to modify the values. Moreover, The process of updating distribution statistics manually becomes practically infeasible with the increase in the number of tables and columns and is much error prone.

Organization. The remainder of this paper is organized as follows: In Section 2, we present the GUI of CODD tool and describe its various features. The technical details of various dataless modes and the implementation of graphical histogram is explained in Section 3. This section also includes the details of interengine portability. Section 4 gives a detailed description of metadata validation process. Metadata scaling is explained extensively in Section 5. Example instances of the functioning of CODD and its various application are illustrated in Section 6. Finally, in Section 8, we summarize our conclusions and outline future avenues.

2 The CODD Interface

Our objectives in designing the CODD interface were to:
(a) enable users to invoke the various modes without having to be aware of the underlying implementation details;
(b) make it vendor-neutral and uniform to the maximum extent possible; and
(c) support creation/updation of the statistical information over the entire spectrum of granularities, ranging from individual attributes to the complete database.

We showcase our realisation of the above design goals by presenting the ConstructMode interface for the DB2 engine. DB2 features a wide range of statistics, ranging from row, page and block cardinalities of tables, to idiosyncratic column fields such as HIGH2KEY and LOW2KEY, signifying the second-highest and second-lowest values in the attribute, respectively. These values can be input in the relation and attribute statistics segment of the interface.

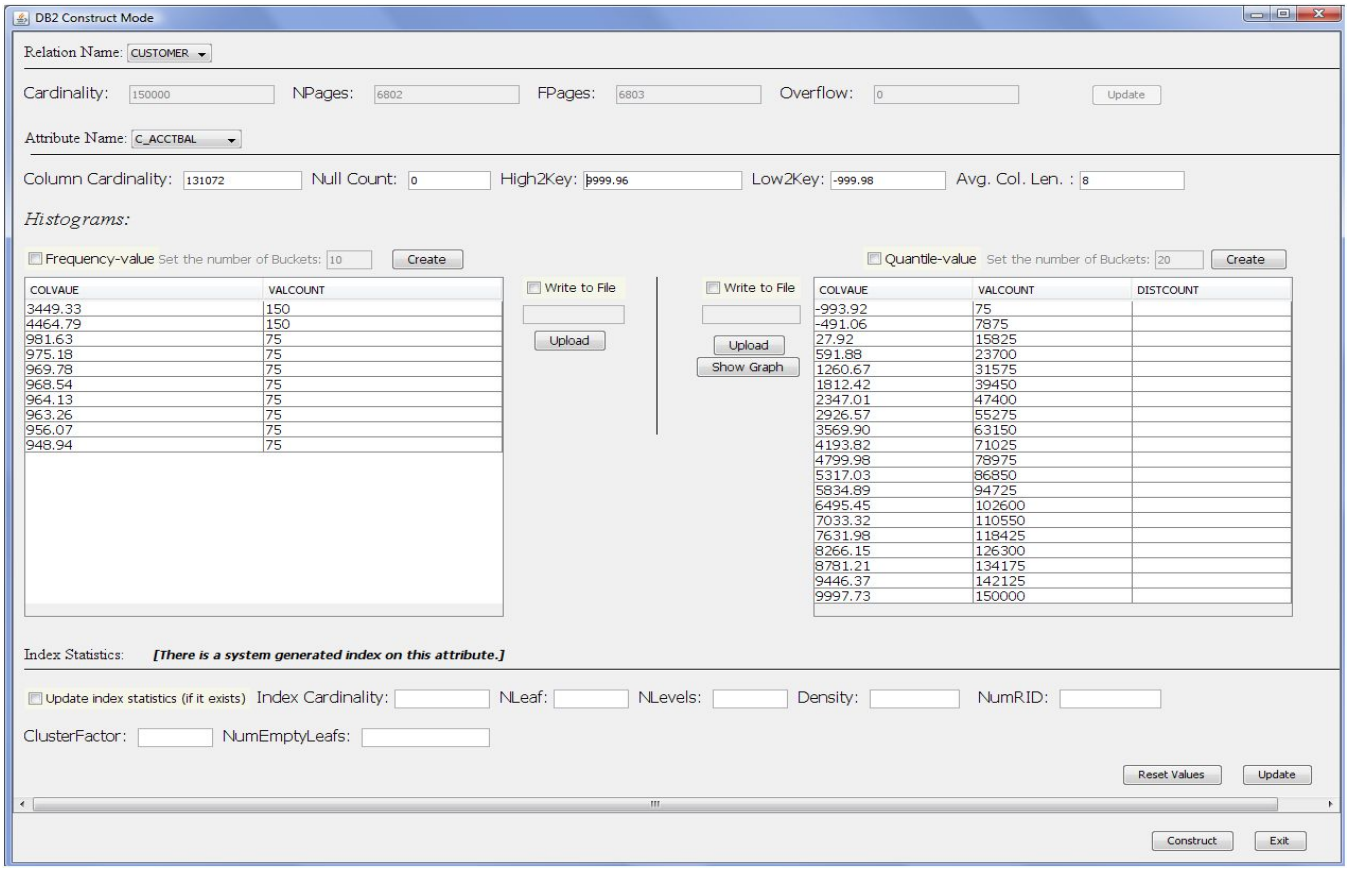


Figure 1: **CODD Interface (ConstructMode on DB2)**

For each relation, users take a simple walkthrough of the selected attributes of the relation using the next button, updating the statistics for these attributes along the way. When all the selected attributes for a relation have been updated, the user moves to the next relation using the update button. Before every statistics update, the input validation is automatically done in the background, thereby ensuring consistency. For example, if the HIGH2KEY value is entered less than the LOW2KEY value, an error message appears requiring the user to reenter the correct values.

DB2 hosts two kinds of column distribution statistics: *frequency histograms* (corresponding to the most common values) and *equi-depth histograms* wherein the frequency values are included in the equidepth histogram buckets [5]. For both these histogram types, the CODD interface allows the frequency values and the histogram bucket cardinalities to be input either from a file or manually. Subsequently, the constructed histogram can be viewed graphically. Further, a *graphical* histogram editing interface, shown in Figure 2, is included in CODD, wherein the current histogram's layout can be

visually altered to the desired geometry by simply reshaping the bucket boundaries with the mouse.

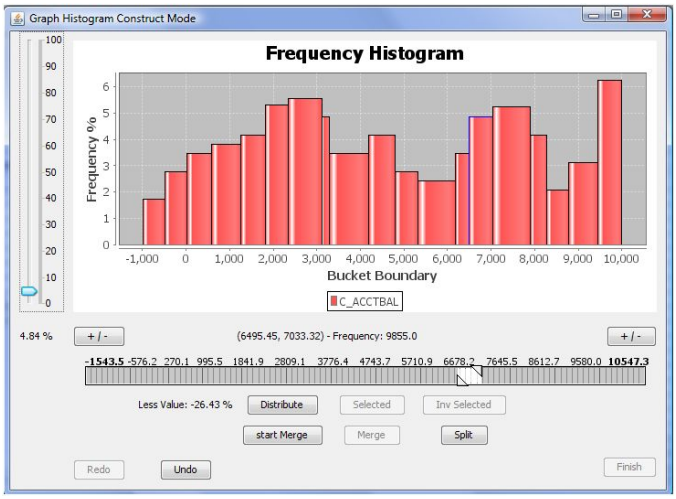


Figure 2: **Graphical Histogram Interface**

Finally, users can also create a new histogram by selecting one of a pre-defined set of classical distributions from a drop down menu, and providing summary information about its distributional properties such as the mean, value range, skew, etc.

As can be expected, the interface for the construct mode is specific to each engine since there are both syntactic and semantic differences across their statistics portfolios. However, this information can be broadly categorized into three components: *table*, *column* and *index* statistics, for all the engines. For RetainMode, CODD provides a single vendor-neutral interface which contains the list of user-selected relations from which the data is to be removed. Also, it contains the list of relations which are dependent on the user-selected relations as data needs to be removed also from the child relations to maintain consistency. CODD also provides an access to the scripting features natively available with each engine through a single interface common to all the engines where the user can specify the source and the target database involved in the metadata transfer.

In addition, CODD provides simple one-click interfaces for metadata scaling and porting metadata across different engines. Finally, as mentioned above, during the ConstructMode, whenever metadata validation fails, the user is provided with a guided message explaining the violation and steps required to resolve it. A more detailed view of interface can be realised in the demonstration video that can be downloaded at [16]. Also, the step-by-step guide to the usage of the interface can be found in the software manual available at [16].

3 Metadata Generation Internals

Having described the interface in the previous section, we now move on to presenting the technical details of the underlying implementation in CODD. The tool is completely written in Java, running to over 10K lines of code, and is operational on a rich suite of industrial-strength database systems including DB2, Oracle, SQLServer, Sybase and PostgreSQL. For the commercial engines, it functions

solely through the database APIs in a non-invasive manner, while for PostgreSQL, a few extra functions have been incorporated in the engine.

Meta-data information in modern database engines covers a variety of aspects, including schema organization, query processing, workload management, and performance tuning. Our focus here is on the statistical metadata related to query processing – the extension to the other aspects is straightforward. In particular, our meta-data is comprised of statistics on the following entities: (a) *relational tables* (row cardinality, row length, number of disk blocks, etc.); (b) *attribute columns* (column width, number of distinct values, value distribution histograms, etc.); (c) *attribute indexes* (number of leaf blocks, clustering factor, etc.); and (d) *system parameters* (sort memory size, CPU utilization, etc). Given this framework, CODD supports two dataless modes called ConstructMode and RetainMode.

These dataless modes available with CODD on different engines are summarized in Table 1. In this table, the annotation "fresh schema" indicates that the metadata update cannot be directly done in-place but indirectly through recreation of the schema, while "entire database" signifies that data dropping cannot be implemented at the level of individual relations, but for the entire schema as a whole. Finally, with regard to computational effort, setting up the dataless modes takes almost no time, completing in less than a second on all the engines.

Table 1: **Dataless Modes on DB Engines**

Engine	**ConstructMode**	**RetainMode**
DB2	Y	Y
Oracle	Y	Y
SQLServer	N (internal format)	Y (fresh schema)
Sybase	Y	Y (entire database)
PostgreSQL	Y (code addition)	Y

In the following discussion, we initially present the dataless modes and then discuss the inter engine protability feature in detail. We conclude the section with notes on how CODD provides unified access to the scripting features natively available with different database engines.

3.1 Metadata Construction (ConstructMode)

We start with the most potent, dataless mode, namely ConstructMode. During this discussion, we will assume that the logical and physical schema of the database is already in place, and that it is only necessary to fill in the statistical information.

DB2. It is feasible in principle to directly update the catalogs in DB2 using the standard update commands. However, it is somewhat cumbersome since a large number of such commands have to be issued in order to input the complete set of statistics for each selected relation. Further, there are subtleties related to the order in which updates are made, and to the initialization of default values. Finally, DB2 only allows for updates on the catalogs and not fresh inserts.

The first two problems are hidden from the user by the CODD interface internalizing these complexities, enabling the user to focus on "what", and not "how". For the last issue of update versus insert, our workaround is to first populate the catalogs by executing the RUNSTATS command on the empty

database, and then to use the update commands to overwrite these dummy values with the desired set of statistics. For some data types, DB2 expects the input to be in hex format, and this is again automatically done by the CODD tool. All parameters for which the user does not provide the input are set to their default values.

Oracle. Here, we use the statistics setting feature provided by the dbms_stats package, which can be operated at table, column and index levels. The setting of table and index statistics is straightforward, only requiring a set of SQL queries to be run on the backend after obtaining the inputs from the user interface. Setting the column statistics, however is more involved since this information, in particular the histograms, are stored in a special internal representation. We convert the users input to this internal representation leveraging the prepare_column_values utility provided with the dbms_stats package [28], customized for each data type. After we obtain the internal representations, we pass them as parameters along with the other column statistics to the SET_COLUMN_STATS utility which stores these values in the catalogs.

To combine all the above tasks, we define SQL procedures which carry out all the tasks step by step and finally set the column statistics. These procedures are created and executed on the fly, and are not persistently stored in the database. Figure 3 shows a code snippet which is used in CODD to implement the above functionalities.

SQLServer. Here, the statistics are stored in the system table SYSINDEXES and in a binary large object STATLOB stored in the SYS.SYSOBJVALUES table which is not directly accessible to users [31]. However, the UPDATE and CREATE STATISTICS commands have an (undocumented) option called STATS_STREAM which can be used to set all the statistics. This is a stream of hexadecimal values which can be viewed using the STATS_STREAM option [31] in conjunction with the DBCC_SHOW_STATISTICS command. However, since its format is currently proprietary, it is not possible to directly edit this file at this time. We hope that the format would eventually become public, enabling us to completely implement the ConstructMode mode.

Notwithstanding the above restriction, With regard to specifically two statistics, ROWCOUNT and PAGECOUNT, a change of values can be implemented using the rowcount and pagecount options with the update statistics command. When these options are used, the bucket frequencies in the histograms are proportionally altered to match the modified cardinality values. But, the number of distinct values cannot be altered with this mechanism.

Sybase. Sybase has two system tables SYSTABSTATS and SYSSTATISTICS that store the statistics. But, in Sybase it is difficult to persistently update these tables since the SYSTABSTATS table is periodically updated by the housekeeping processes that continually run in the background [33]. Another issue with Sybase is that there is a complex mapping of database objects to identifiers, and it is difficult for users to directly input these values.

On the other hand, the column and distribution statistics, which are stored in the SYSSTATISTICS table do not get flushed or controlled by any automatic process. However, these statistics are stored in the form of their internal representation (mostly binary values) and they are rather difficult to modify directly due to large amount of columns and data stored in the table.

The major task for CODD here is to generate a file similar to the one generated by optdiag so that this file can be provided as input rather than the one provided with the statistics gathered from the existing database. The necessary steps have been taken to create the file in the specific format as required by the utility and also the user values are converted to their equivalent binary format using the inbuilt convert() function [33].

For table statistics:

DBMS_STATS.SET_TABLE_STATS(own-name,tabname,numrows,
numblks,avgrlen);

For index statistics:

DBMS_STATS.SET_INDEX_STATS(ownname,indname,numrows,
numblks,numdist,avglblk,avgdblk,clstfct,indlevel);

For column with Frequency-Based Histogram:

Input: owner name, table name, column name, distinct count, density, null count, average length, endpoint number array, endpoint value array, number of buckets

```
DECLARE

m_distcnt number;

m_density number;

m_nullcnt number;

srec dbms_stats.statrec;

m_avgclen number;

n_array dbms_stats.numarray;

begin

m_distcnt := dist_cnt;

m_density := density;

m_nullcnt := null_cnt;

m_avgclencnt := avg_col_len;

n_array := dbms_stats.numarray(endpoint_value_input);

srec.bkvals := dbms_stats.numarray(endpoint_number_input);

srec.epc := buckets;

dbms_stats.prepare_column_values(srec, n_array);

dbms_stats.set_column_stats(ownname=>'''+
own_name.toUpperCase()+''',tabname=>'''+
tab_name.toUpperCase()+''',colname=>'''+column+
''',distcnt=>m_distcnt, density=>m_density, nullcnt=>m_nullcnt, srec=>srec,
avglen=>m_avgclen);

end;
```

Figure 3: **Procedure to update statistics in Oracle**

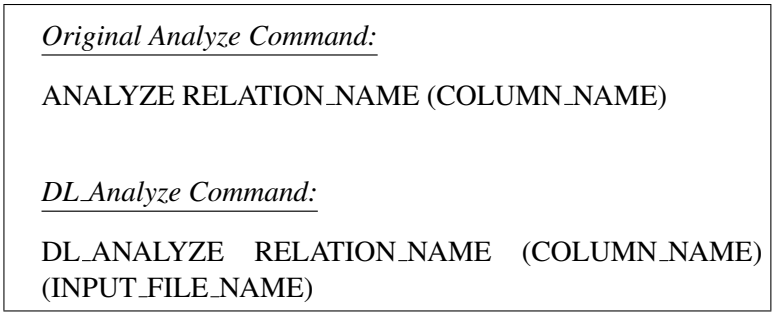


Figure 4: **Original and modified command for PostgreSQL**

PostgreSQL. Optimizer statistics in PostgreSQL are stored in two system tables, namely PG_CLASS and PG_STATISTICS. The view PG_STATS provides access to the information stored in the PG_STATISTICS catalog.

The table and index statistics are stored in the PG_CLASS catalog, and this table can be directly updated using the simple *update* and *alter table* commands. tables. However, we felt that it it is better to have internal system fields such as *relid* to be directly filled by the system, so we have chosen to update the statistics rather than inserting them. To do this, we first run an analyze command on the empty relations and then update the rows of PG_CLASS for these indexes and relations using the values provided by the user.

The column statistics stored in the PG_STATISTIC table are more difficult to change because this table cannot be altered externally. The reason is that the data type of two columns, MOST_COMMON_VALS and HISTOGRAM_BOUNDS, is *anyarray* which is in reality a pseudo-type [34]. PostgreSQL does not allow to alter or insert values from an external application into attributes having pseudo-types.

To address this problem, we tweaked the use of the in-built ANALYZE command which is used to update the statistics of attributes and relations. Specifically, we developed a new command, called DL_ANALYZE, which has similar construct as that of the current command but instead of analyzing the actual columns and relations as required to generate the column statistics, it now makes the statistics collector read the input file given by the user and fetch the statistics directly from that file. The usage of new command is shown in Figure 4

To achieve the above altered behavior, the primary code changes have been made in *gram.y* (parser file) and *analyze.c* (analyze relation file) of the PostgreSQL codebase.

3.2 Metadata Retention (RetainMode)

The main issue related to the Drop Mode is ensuring that the statistics are not recomputed when the database contents are dropped. Since the mode is not available natively, we have implemented the following engine-specific techniques to provide the functionality.

DB2. Here, the data from the selected relations is deleted using the TRUNCATE command. This command removes the rows in virtually no time, and more importantly, does not update the statistics associated with the relations. The storage space is reclaimed by using the drop storage option in conjunction with the TRUNCATE command. Also, we stop the automatic maintainance of statistics using the database configuration commands for the session in which data is removed so as to make sure that no metadata is affected during that time. Later, these parameters are set to on as they do not affect

the metadata of the truncated table then.

Oracle. Here, the DBMS_STATS package is used to lock in the catalogs, the table statistics rows pertaining to the relations that are be truncated, so as to disable any subsequent updates. Then, the contents of these relations are removed using the TRUNCATE command. We specifically choose the TRUNCATE command as opposed to the DELETE command because it provides an automatic commit and no logging, whereas the latter has to be committed explicitly and all its actions placed in the undo log for recovery purposes.

SQLServer. Here, the availability of scripting facilities makes RetainMode easy to implement as it substantially overlaps with the natively available TransferMode mode. But a major difference with regard to the other engines is that the relations whose contents are to be removed have to be completely *dropped* and their schema subsequently recreated. Specifically, the scripts for the relations to be dropped are first generated, including only meta-data information. Then, the relations are completely eliminated from the database using the DROP command after disabling all constraints. To reclaim the storage space, the SHRINK DATABASE command is used on the *mdf* database file. Then the script file is run against the database to first recreate the schemas of the relations that were dropped, and then restore their statistics.

Sybase. Here, the optdiag facility is used to initially dump all the schema and statistical information into a file [33]. However, it is not possible to retain the statistics intact while removing the data since (a) there are no locking mechanisms provided on the system tables, and (b) the housekeeping functions flush the statistics as soon as there is an update to the database and these functions cannot be disabled. Therefore, we can only *simulate* the statistics when required, and these simulated statistics are only persistent for the duration of the current user session.

An idiosyncratic feature of Sybase is that the database is stored in segments on a logical device and the size of the device must be large enough to store the database residing on it. Due to this requirement, whenever the size of the database increases, the size of the database device must also be increased. But, when the data is deleted, the space used by the device is not decreased or released – hence the device also needs to be deleted in order to reclaim the storage space. Therefore, in our strategy, the entire database is first dropped and the associated logical device is deleted. Then a new device of a smaller size is created and a new database with the same logical and physical schema as before is created. The optdiag utility is then run to simulate the statistics using the earlier output file as an input parameter.

PostgreSQL. Here, there is an automatic vacuum daemon which is responsible for updating the statistics automatically whenever there is any update to the database. To achieve our RetainMode mode goal, this daemon is disabled during the truncation process by switching off the TRACK_COUNTS and AUTO_VACUUM configuration parameters [34]. Then the data from the relations is removed using the TRUNCATE command. In this process, the column statistics remain intact but the table and index statistics get flushed out. They are then restored in the following manner: Before the truncation, the statistics for the selected relations in the PG_CLASS table are dumped to a file. This file is then used to update the PG_CLASS table after the truncation completes.

As an aside, we would also like to mention that for all the engines, it is not permissible to insert new tuples into the relations on which RetainMode has been applied since doing so may trigger an update of the statistics based on the current contents, in the process overwriting the previous values.

3.3 Metadata Transfer (Inter-Engine Mode)

An attractive feature of CODD is that it supports automatic porting, to the extent possible, of the statistical metadata across database engines, thereby facilitating comparative studies of systems as well as early assessments of the impact of data migration. While most database engines do provide the facility to transfer the metadata from one database to another on their *own platform*, to our knowledge, none of them support porting metadata across different engines. To achieve this goal in CODD, a semantic mapping has been carefully worked out between the statistical information appearing in the various engines.

Although each engine has its own idiosyncratic metadata, we have found that most of the table level, column level and index level statistics are fully portable across all the engines. However, the transfer of distribution statistics is only partially feasible across some pairs of engines. To achieve the maximum possible fidelity, we first convert the source distribution statistics to a canonical form (which resembles the style of DB2), and then convert this information to a format compatible with the target engine.

We exemplify the above process by showing the detailed procedure for inter engine transfer from DB2 to Oracle. The correspondence between the parameters of DB2 and Oracle is shown in Table 2

Table 2: **DB2-Oracle Mapping**

Statistics Level	**Oracle**	**Corrsponding statistics in DB2**
Table	No. of Rows	CARD
	No. of Blocks	NPages
	Avg. Row Size	-
Attribute	No. of Distinct Values	COLCARD
	No. of Null Values	NUM_NULLS
	Avg. Col. Length	Avg. Col. Length
Distribution	Height-Balanced/Frequency Histogram	Quantile and Frequency Histogram
Index	No. of leaf blocks	Nleafs
	Index Levels	INDLEVEL
	Cluster Factor	CLUSTERFACTOR

The overall feasibility of the metadata transfer across different pairs of engines is summarized in Table 3. In this table, a Y entry signifies that more than 95% of the metadata can be translated, whereas a Partial entry means that about two-thirds of the data can be populated, while the N entry indicates that the transfer is infeasible. As can be seen, it is only with SQLServer that conversion is not possible due to its proprietary format for communicating statistics.

Table 3: **Inter-Engine Metadata Transfer**

Engine	**DB2**	**Oracle**	**MSSQL**	**Sybase**	**PostgreSQL**
DB2	-	Y	N	Partial	Y
Oracle	Partial	-	N	Partial	Partial
SQLServer	Y	Y	-	Y	Y
Sybase	Y	Y	N	-	Partial
PostgreSQL	Y	Y	N	Partial	-

3.4 Metadata Transfer (Native Mode)

Finally we note down the details on the usage of various natively available scripting abilities with each database engine that we have leveraged upon, to provide the user with the same functionality in CODD.

DB2. Here, the db2look utility is used to transfer the statistical information, following the procedure described in [7]. However, there are additional configuration parameters such as STMT_HEAP SIZE and SORT_HEAP SIZE, which are *reset* when transfer is done by db2look – the updating of these parameters to their original values is additionally handled by CODD.

Oracle. Here, the DBMS_STATS package is used to transfer the statistical information, following the procedure described in [26].

SQLServer. Here, the native scripting facility is used to transfer the statistical meta-data, following the procedure described in [30]. As mentioned earlier, the column information is encoded in a proprietary stream format.

Sybase. Here, the optdiag utility is used in conjunction with its *simulate* option to generate a statistical metadata file that has the simulate tag associated with its contents. This file can be loaded onto the destination machine again using the optdiag facility, following the procedure described in [8].

PostgreSQL. Here, it is possible to transfer the *entire database* from one site to another [34], but there is no explicit transfer mode available solely for meta-data. The reason is the following: The table and index statistics are maintained in the PG_CLASS table while the column statistics are maintained in the PG_STATISTIC table. While it is straightforward to directly dump and restore values in the PG_CLASS table, this is not the case with the PG_STATISTIC table, however, since two of its columns (MOST_COMMON_VALS and HISTOGRAM_BOUNDS) are locked out with regard to external updates.

Therefore, we were forced to modify the PostgreSQL codebase in order to circumvent the above problem. In particular, we use the above mentioned newly devised DL_ANALYZE command that is similar to the standard ANALYZE command [34] for updating the statistics, but supports changes in the PG_STATISTIC table as well. After dumping the distribution statistics into the file, we put these values in the format required by DL_ANALYZE command. We then use this command to import the statistics into the PG_STATISTIC table which completes the metadata transfer.

3.5 Graphical Histogram Details

As describe in Section 2, CODD provides a feature to modify the data distribution of a column through graphical interface. We use JFreeChart [18] to implement the graphical histogram. JFreeChart is a free chart library to produce charts and graphs with extensive set of features. The Graph Histogram takes the total row count, total distinct column values in the count and an initial histogram as input and produces the initial graph histogram. Now the user can reshape the graph to get desired data distribution. The graphical histogram is used in two modes of operation as given below:

- Frequency Mode - The graph allows the user to operate on the frequency values of the buckets.
- Distinct Count Mode - The graph allows the user to operate on the distinct count values of the buckets.

Figure 2 shows the instance of modified graph histogram of s_acctbal column of TPC-H supplier relation. The reshaping on graph is done through a set of operations with mouse and buttons in the graph

histogram. Reshaping is constrained to the total row count , total distinct of the column and other metadata value. The constraints ensure the consistency among metadata. For example, in DB2, the VALCOUNT of highest COLVALUE must be equal to the CARD of the relation. Such consistency constraints checked before the operation is performed. In case of any violation of the constraint, error is reported to the user. In short, Graph Histogram takes consistent input and allows the user to reshape the graph with consistency and returns consistent histogram. The graph shows the row counts of bucket in terms of percentage instead of absolute values. The major features of CODD's graphical histogram to reshape the histogram are as follows:

- Bucket height can be changed (increase or decrease).
- Bucket width can be changed for columns of type INTEGER and DOUBLE.
- Two or more adjacent buckets can be merged into one bucket.
- A bucket can be split into multiple buckets by defining the intermediate values and row count percentage for new buckets.
- Buckets can be added or removed at both ends of the histogram.
- Reshaping the initial buckets may bring the total row, distinct count percentage to be more or less than 100%. In such cases, the excess or less row, distinct count percentage can be distributed among selected buckets of the histogram.

Also, CODD graph histogram stores last 10 reshaping operations to allow the user to undo or redo in case if she wants to revert an operation. Reshaping operations are constrained by the legal and consistent values. For example, a bucket distinct count can not be increased beyond its frequency value.

4 Metadata Validation

In RetainMode, the meta-data is guaranteed to be valid since it is sourced from a real database instance. However, in ConstructMode, since users are directly allowed to enter the meta-data, we need to ensure that the inputted information is both *legal* (valid type and range) and *consistent* (compatible with other metadata values). We now discuss how these issues are tackled in CODD. For ease of exposition, we will restrict our attention to the DB2 engine here – the handling of the other engines is similar in flavor.

Our validation approach is to first construct a *directed acyclic constraint graph* that concisely represents all the applicable constraints. Specifically, each node in the graph represents a single metadata entity that is annotated with associated legality constraints, and the currently assigned value which must adhere to these constraints. The directed edges, on the other hand, are used to represent statistical value dependencies between the metadata entities. Since the dependencies are typically bi-directional, to prevent duplication of edges, we adopt the convention that the edge will be directed from the node at the higher level of abstraction to the lower level node (e.g. from relation to attribute), while for nodes at the same level, the edge goes from the aggregate to the specific (e.g. from cardinality to distributions), and for the remainder, a lexicographic ordering is used. Our choice of convention attempts to reflect the natural manner in which schemas are usually developed by human users.

Let $G = (V, E)$ be a directed acyclic graph such that,

V - Set of nodes, where each node $v \in V$ represents a single entity of the metadata which includes the value assigned to it and the structural constraints. The value must adhere to the structural constraints.

E - Set of edges, where each edge $e(u, v) \in E$ represents the statistical consistency constraints associated with the two nodes. Directions on the edges specifies the traversal order.

Directions are added based on the database semantic information on the nodes. For example, the directions are added from relation level metadata to column level metadata. So with the added directions to the graph, we were able to get a DAG G. Graph G is just one of the many possible DAG's on the constrained graph.

Figure 5 shows the constructed Constrained DAG for DB2. Similar graph has been constructed for other engines and deployed in the Construct mode of CODD. DB2 updatable metadata gives the node set and the constraints on the nodes listed in [15] gives the edge set. The nodes are grouped into three categories as Relational level, Index level and Column level metadata. Each node has a structural constraint along with the value to the metadata. In Figure 5(a), the node **Card** shows the structural constraints (data type as integer and valid value is greater than or equal to 0 or -1). Edge between node **Card** and node **ColCard** imposes the constraint that **ColCard** must be less than or equal to **Card**.

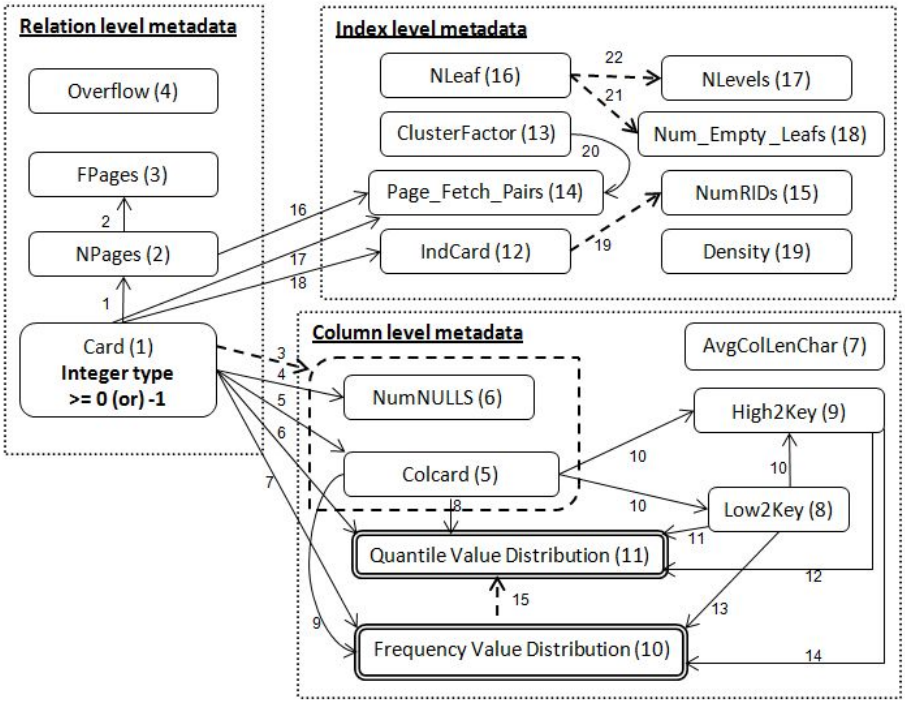
We observed that the DB2 vendor documented [15] constraints are not sufficient to define the consistent metadata. The following constraints are not specified in [15], but are needed to have the consistent metadata and thereby to get a legal database.

- Sum of NumNulls and ColCard must be less than or equal to Card of the relation.
- The VALCOUNT of a Quantile Histogram bin must be greater than the sum of all VALCOUNTs in the Frequency Histogram whose COLVALUE is less than the Quantile Histogram bin COLVALUE.

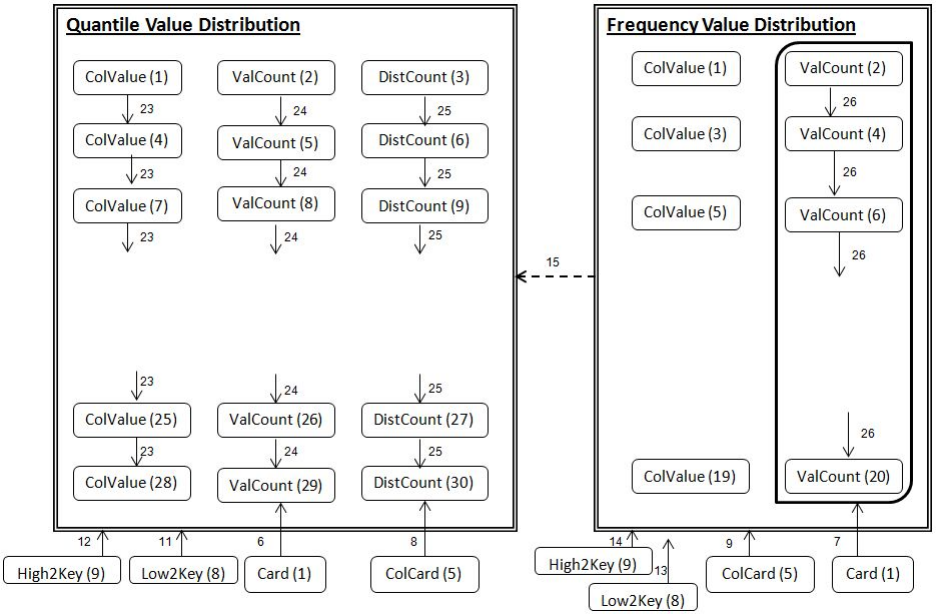
These constraints are added to the graph and shown in Figure 5(a) as dashed edges. Numbers in the edges represents the constraint numbers and the constraints are listed in Appendix A. The Figure 5(a) shows the nodes **Quantile Value Distribution**, **Frequency Value Distribution** with double line border. The double line border represents that each of these nodes has a graph inside it which is shown in Figure 5(b). Distribution graph shows the histogram bin values, frequency and distinct count as nodes and the constraint that the nodes are ordered as edges.

The graph has a complex structure. It has a few independent nodes as well as highly connected nodes. Node **Card** has the highest outdegree of 8, which is referenced by many other nodes. The total no. of nodes in the graph is 99 (= 4 + 7 + [60 (Q) + 20 (F)] + 8), assuming that there are 20 Quantile histogram bins and 10 Frequency histogram bins. The total number of edges in the graph is 90 (= 10 + 10 + [57 (Q) + 9 (F)] + 4]).

Finally, after the constraint graph $G(V, E)$ has been fully constructed and populated, we run a *topological sort* on G. The sort provides a linear ordering G_{linear} of the nodes, and can be accomplished in time complexity $O(|V| + |E|)$ [2]. Then, CODD guides the user through this linear ordering,



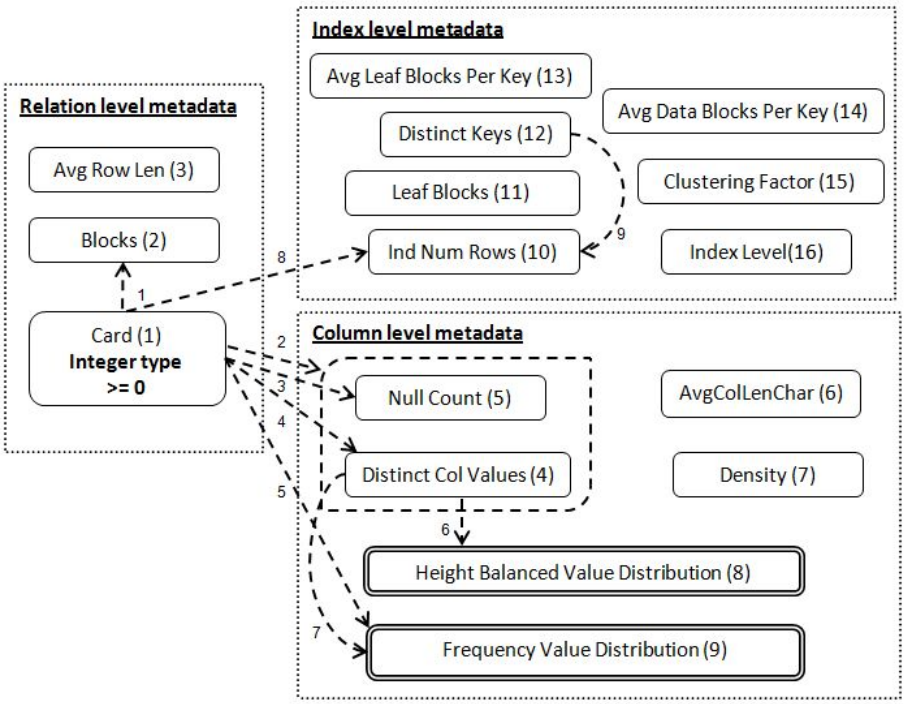
(a) Constraint Graph



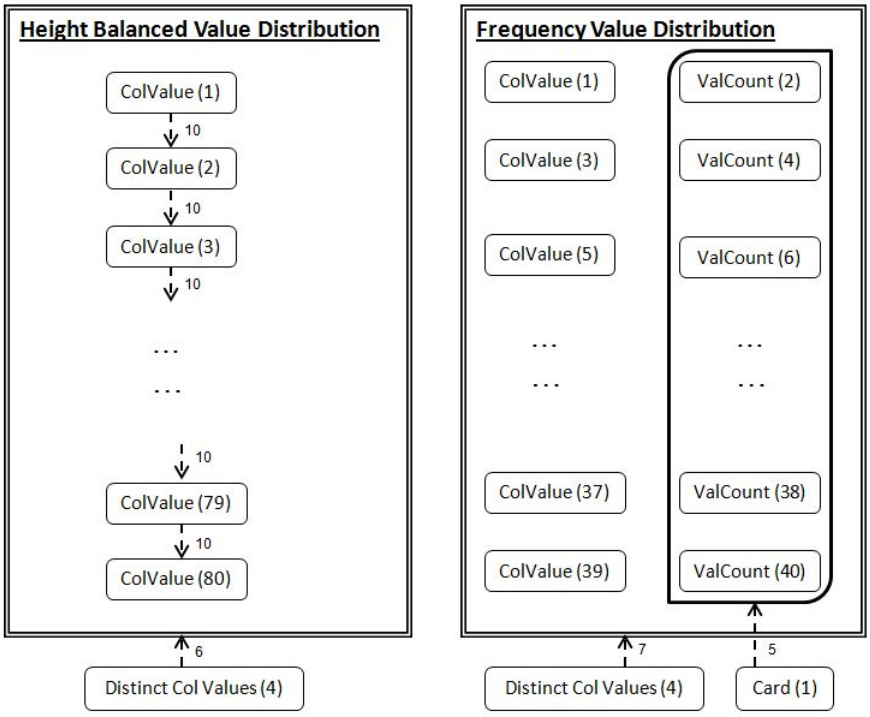
(b) Super nodes of Constraint Graph

Figure 5: DB2 Metadata Constraint Graph

requesting inputs at each new node, and ensuring that all applicable constraints are met by the freshly added entries. A sample linear ordering is shown through the numbers associated with the nodes in Figure 5(a), beginning with CARD (1) and ending with DENSITY (19). The user input / validation process starts with Relation level metadata followed by Column level and Index level metadata. At the



(a) Constraint Graph



(b) Super nodes of Constraint Graph

Figure 6: Oracle Metadata Constraint Graph

Data Distribution nodes, the inner graph is validated first and the main graph validation is continued. At the end of the validation process, the metadata will be consistent and thereby a legal database can be

obtained. The validated metadata is updated in the database catalogs to complete the construct mode.

Similarly for other engines, the constraint graph is constructed and topological sorted constraint graph validation is incorporated into CODD. Figure 6 shows the constraint graph for Oracle database engine and the constraints are listed in Appendix A.

5 Metadata Scaling

A common activity in database engine testing exercises is to assess the behavior of the system on scaled versions of the original database, and this is the reason that benchmarks such as TPC-H and TPC-DS are available in a variety of scale factors. Current benchmarks typically implement a size-based scaling approach – for example, in TPC-H, the relation cardinalities are linearly scaled, while domain-size scaling is implemented for the primary keys and foreign keys referencing the scaled tables. In TPC-DS too, the fact table is scaled linearly, but the dimensions undergo sub-linear expansions according to a hard-wired scaling assignment.

CODD supports these size-based scaling models of TPC-H and TPC-DS. In addition, it also provides a novel *cost-based* scaling model. Here, the aim is to scale the baseline metadata $\mathcal{M}$ such that the optimizer's estimated cost of executing a given query workload $\mathcal{Q}$ on the scaled version, $\mathcal{M}^\alpha$, is a specified multiple, α, of the cost of executing it on the original database. Initially, we attempt to produce a metadata instance such that each *individual* query in $\mathcal{Q}$ is scaled by α. However, this may often be fundamentally infeasible, in which case we settle for solving the following optimization problem:

Produce an $\mathcal{M}^\alpha$ such that the sum over $\mathcal{Q}$ of the individual squared deviations from α in cost scaling is minimized, subject to the constraint that the overall cost over $\mathcal{Q}$ is scaled by α.

That is, given relations $R_1, R_2, \ldots, R_h$ appearing in $\mathcal{Q}$, identify a size-scaling vector $(\alpha_1, \alpha_2, \ldots, \alpha_h)$ such that

$$\sum_{q_i \in \mathcal{Q}} [c^S_{q_i}/c^O_{q_i} - \alpha]^2$$

is minimized subject to

$$\sum_{q_i \in \mathcal{Q}} c^S_{q_i} = \alpha * \sum_{q_i \in \mathcal{Q}} c^O_{q_i}$$

where $c^O_{q_i}$ and $c^S_{q_i}$ represent the costs of q_i in the original and scaled databases, respectively.

Achieving the above objective function appears to be a hard problem since we need to be able to mathematically relate the overall costs of query plans to scaling factors on their base relations, and this is a rather complex proposition. However, if we (a) specified that, except for primary key columns, the scaling would be implemented such that the relative frequency distributions of the remaining columns are *identical* between the original and scaled databases, and (b) assumed that the choice of query plans is *retained* between the original and scaled databases, then it becomes quite feasible to generate a satisfying scaling vector. This is because of Lemma 1 (Proof is given in Appendix B) in Figure 7, – this lemma allows for constructing each plan operator's output cardinality in the scaled database solely in terms of the corresponding cardinality in the original database and the relation scaling vector.

Figure 8 delineates the summary procedure to compute the total cost of a query on the scaled database, expressed in terms of the scaling factors of the relations appearing in the query. For example, with Q14 of the TPC-H benchmark, which features the PARTS and LINEITEM relations, the

Lemma 1. Let $R_1, R_2, ...R_h$ be the input relations to operator *op* and $\alpha_1, ...\alpha_h$ be their scaling factors respectively. Then, if the relative frequency distributions of the scaled database (SD) and the original database are identical for non-key columns and if the domain is scaled for the key columns of the SD, then the output size of each operator *op* in the plan tree for the SD is expressible as

$s(\alpha_m, ...\alpha_n) \times Original\ output\ size$

where $\alpha_m, ...\alpha_n$ are the subset of scaling factors such that $\forall \alpha_i \in (\alpha_m ... \alpha_n)$, the relation R_i is not referenced by any other relation $R_j \in \{R_1, R_2, ...R_h\} \setminus R_i$; and s is a function on this subset of scaling factors. Further, the relative frequency distribution of the scaled output is identical to the frequency distribution of the original output.

Figure 7: **Scaled output size and distribution**

Input: Query q_i

Result: Cost function $c_{q_i}^S$

1. Obtain the query execution plan for the given query.

2. Determine the cost function for each operator in the execution plan with respect to the sizes of the inputs.

3. Using Lemma 1, determine the scaled output size in terms of scaling factors for each operator in the execution plan.

4. Calculate the cost of each operator for scaled inputs using the cost functions obtained in Step 2.

5. Compute the total cost of the query as the aggregate of the costs of the operators present in the execution plan.

Figure 8: **Query costs in scaled database**

following expression was obtained on DB2:

$$c_{q_{14}}^S = 391 \ * \ \alpha_p + 17827 \ * \ \alpha_l + 20 \ * \ \alpha_l + \frac{62 * (2 * 10^5 * \alpha_p + 80531 * \alpha_l)}{2 * 10^5 + 80531}$$

where α_p and α_l are the scaling factors for the PARTS and LINEITEM relations, respectively.

Armed with these individual query cost functions, we can now compute the scaling vector that would provide the best scaling configuration, using the optimization procedure enumerated in Figure 9. Using the sophisticated optimization methods, the solution is found to converge in less than a minute. When multiple solutions are available for the scaling vectors, the final choice made is to pick the solution that is closest to a traditional size-based scaling approach (Step 4 in Figure 9), since it is our expectation that this would result in more robustness with regard to (a) addition of new queries to the workload, and (b) retention of the same plans across the scaled databases.

As an example, consider a TPC-H workload consisting of queries Q12, Q13 that operate on relations Orders, Customers and Lineitem with equal probability and scaling factor of 3 (α). The scaling factors of Orders, Customer and Lineitem are assumed to be α_o, α_c and α_l. As a first step (Figure 9) to solve

Lemma 2. If the key columns of relations are domain scaled and the primary key columns $C_a, ...C_n$ of relation R_i are a combination of foreign key columns, which are referencing the relations $(R_a, ...R_n)$ respectively, then the scaling factor α_i of relation R_i is bounded by the product of $\alpha_a ...\alpha_n$, where $\alpha_a ...\alpha_n$ are the scaling factors of relations $R_a, ...R_n$ respectively.

Algorithm

Input: Metadata Instance $\mathcal{M}$, Scaling factor α, query workload $\mathcal{Q}$

Result: Scaled Metadata Instance $\mathcal{M}^\alpha$

1. Determine the cost of each query q_i using our cost model. We obtain $c^S_{q_i}(\alpha_1 ...\alpha_k)$.

2. Cost of executing query in the original database $c^O_{q_i}$ is obtained from the execution plan.

3. Solve the optimization problem,

$$Minimize \sum_{q_i \in \mathcal{Q}} [c^S_{q_i}(\alpha_1 ...\alpha_k)/c^O_{q_i} - \alpha]^2$$

subject to

$\sum_{q_i \in \mathcal{Q}} c^S_{q_i} = \alpha * \sum_{q_i \in \mathcal{Q}} c^O_{q_i}$

for i between 1 and k

$0 < \alpha_i <= Lemma\ 2\ Bound$, if applicable

$0 < \alpha_i < \infty$, otherwise

$c^S_{q_i}(\alpha_1 ...\alpha_k) = \alpha * c^O_{q_i}$, if cost of individual query q_i has to be scaled by α

4. From solutions S obtained in step 3, pick a solution $s \in S$ that minimizes the following:

$$\sum_{\alpha_i \in s} (\alpha - \alpha_i)^2$$

5. Scale the input relations with the scaling factors obtained in step 5 to get the required cost scaled metadata $\mathcal{M}^\alpha$.

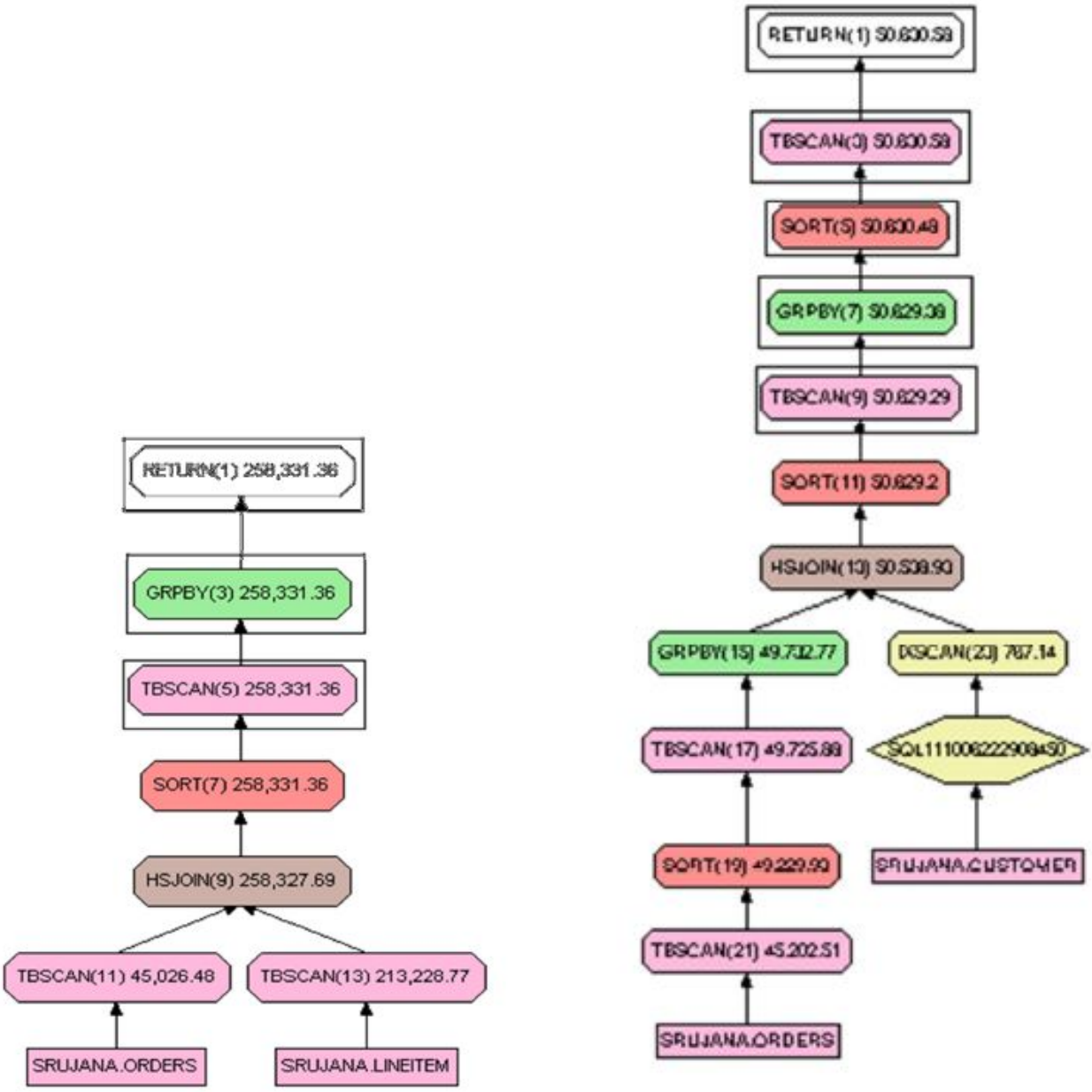
Figure 9: **Cost-scaling of metadata**

the cost scaling problem, we have to determine the cost of queries in scaled database as in the procedure (Figure 8). The execution plans for both the queries are obtained from DB2 optimizer and given in the Figures 10(a), 10(b). We consider CPU cost (in millions of instructions) as the cost of the query. We neglected the operations that have negligible CPU cost in calculating the cost function of the query. We assume the simple cost model for each operator cost as a function of inputs, which is given in the Table 4.

Cost Calculation for Q12

The steps 2 to 4 of Procedure (Figure8) to determine the cost of each operator in the plan tree for scaled database is as follows:

We illustrate the cost function calculation for the operator TBSCAN(11) in 10(a).



(a) Query 12 (b) Query 13

Figure 10: **Estimated plan trees for Q12 and Q13**

- Determine the cpu cost (in millions of instructions) of operator from the execution plan tree and let it be CPU_Cost.
  Eg: *CPU_Cost(TBSCAN(11)) = 2857*

- Determine inputs x, y to the operator and find the cost of the operator using our simple cost model 4 and let the cost of operator be SCM_OpCost.
  Eg: *SCM_OpCost(TBSCAN(11)) = 1500000 i.e The input to the operator is relation Orders. The input size to TBSCAN(11) is the cardinality of relation Orders.*

- Now, the DB2 cost function of this operator is obtained as
  $CostFn = (CPU\_Cost/SCM\_OpCost)$.
  Eg: *$CostFn$(TBSCAN(11)) = 2857 / 1500000*
- We use the Lemma 1 to determine the scaled outputs of each operator.
  *Scaled output of TBSCAN(11) = $alpha_o$ * 1500000*
- Determine the cost of operator for scaled relations and let it $Scaled\_SCM\_OpCost$.
  Eg: *$Scaled\_SCM\_OpCost$(TBSCAN(11)) = $alpha_o$ * 1500000 i.e scaled cardinality of Orders.*
- Now, the operator cost for scaled relations is obtained as follows:
  $CostFn * Scaled\_SCM\_OpCost =$
  $CPU\_Cost * (Scaled\_SCM\_OpCost/SCM\_OpCost)$.
  Eg: *Scaled Cost(TBSCAN(11)) = $CostFn * Scaled\_SCM\_OpCost$*
  *= (2857 / 1500000) * $alpha_o$ * 1500000 = 2857 * $alpha_o$.*

Let x and y be the inputs to the operators.	
Operator	**Cost**
Hash Join	$x + y$
NL Join	$x * y$
Index NL Join	$x + y$
Sort Merge Join	$x + y$
Table Scan	x
Index Scan	x
Filter	x
Group by	x
Sort	$xlogx$

Table 4: **Assumed DB2 Cost Function for Plan Operators**

Similar to the cost calculation of operator TBSCAN(11) for scaled realtions, we computed the cost of other operators of Query12 execution plan tree (Figure 10(a)):

1. TBSCAN(11) : 2857 α_o
2. TBSCAN(13) : 21102 α_l
3. HSJOIN(9) : 275 (23435 α_l +1500000 α_o)/1523435
4. SORT(7) : 14 α_l log(23435 α_l)/log 23435

The total cost is CostQ12(α_o,α_l) = 3127 * α_o + 21106 * α_l + 3 * α_l * log(24234 * α_l)

The original (before scaling) cost obtained from the optimizer (Sum of CPU cost of all operators) is 24248.

Cost Calculation for Q13

The cost of operators for scaled relations of Query13 execution plan tree (Figure 10(b)):

1. TBSCAN(21) : 3525 α_o
2. SORT(19) : 3002 α_o log(1128170 α_o)/log 1128170
3. TBSCAN(17) : 247 α_o
4. GRPBY(15) : 26 α_o
5. IXSCAN(23) : 262 α_c
6. HSJOIN(13) : 148 (150000 α_c +104448 α_o)/254448
7. SORT(11) : (343 α_o) log(150000 α_o)/log 150000

The total cost is CostQ13(α_o,α_c) = 3858 * α_o + 349 * α_c + 496 * α_o * log(1128170 * α_o)+ 66 * α_o * log(150000 * α_o)

The original (before scaling) cost obtained from the optimizer (Sum of CPU cost of all operators) is 7558.

Now minimizing the objective function
$((CostQ13(\alpha_o, \alpha_l)/24248)3)^2$ + $((CostQ13(\alpha_o, \alpha_c)/7558)3)^2$
on the constraints

$$CostQ12(\alpha_o, \alpha_l) + CostQ13(\alpha_o, \alpha_c) = 3(24248 + 7558),$$
$$0 < \alpha_o, \alpha_l, \alpha_c < \infty$$

The local minimum obtained is ($\alpha_o, \alpha_l, \alpha_c$) = (3, 3, 3) (rounded to nearest integer).
The relations Orders, Customer and Lineitem are scaled to the three times and the cost of queries before scaling and after scaling is given in Table 5. The cost of individual queries and query workload is scaled by the scaling factor 3.

Query / Cost	**Before Cost Scaling Total Time (timerons)**	**After Cost Scaling Total Time (timerons)**	**Obtained Scaling**
Q12	258331	774995	3
Q13	50630	153897	3

Table 5: **Cost of Queries before and after scaling**

6 Utility of CODD

Having described the interface and internal mechanisms of CODD, we now present a sample scenario that highlights the tool's utility. For this purpose, we will use the notion of "plan diagrams" [11], which have become potent tools in the design of database query optimizers. Specifically, given a parametrized SQL query template that defines a relational selectivity space, and a choice of database engine, a plan diagram is a visual representation of the plan choices made by the optimizer over this parameter space.

In a nutshell, plan diagrams visually capture the optimality regions of POSP [6], the parametric optimal set of plans.

To make this notion concrete, consider QT9, the parametrized SQL query template show in Figure 11, which is based on Query 9 of the TPC-H benchmark. Here, selectivity variations on the SUPPLIER and PARTSUPP relations are specified through the s_acctbal :varies and ps_supplycost :varies predicates, respectively.

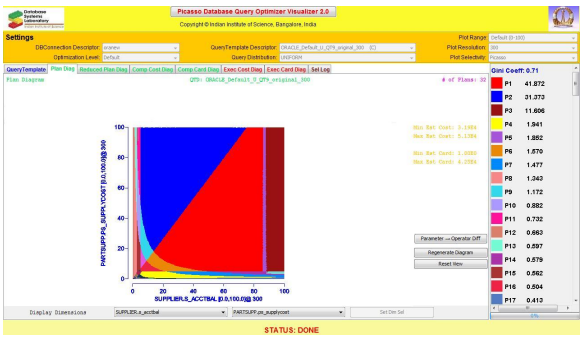
```
select n_name, o_year, sum(amount)

from (select n_name, o_orderdate, l_extendedprice

   from part, supplier, lineitem, partsupp, orders,
   nation

   where s_suppkey = l_suppkey and ps_suppkey
   = l_suppkey and ps_partkey = l_partkey and
   p_partkey = l_partkey and o_orderkey =
   l_orderkey and s_nationkey = n_nationkey and
   p_name like %green% and
   s_acctbal :varies and ps_supplycost :varies

   ) as all_nations

group by n_name, o_year

order by n_name, o_year desc
```

Figure 11: **Example Query Template: QT9**

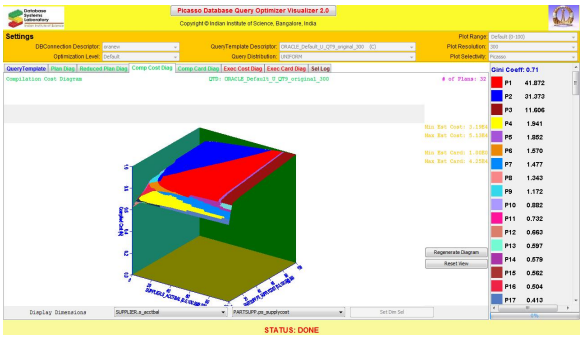
The associated plan diagram produced by Oracle on the baseline TPC-H [19] database of size 1 GB, using a lightweight laptop with a 64GB solid-state hard disk, is shown in Figure 12(a). In this picture, each colored region represents a specific plan, and a set of 32 different optimal plans, P1 through P32, cover the selectivity space. The value associated with each plan in the legend indicates the percentage area covered by that plan in the diagram – the biggest, P1, for example, covers about 42% of the space, whereas the smallest, P32, is chosen in only 0.002% of the space.

In addition to the plan diagram, we also show in Figure 12(b) the "cost diagram", which quantitatively depicts the estimated query processing costs of the plans shown in the plan diagram. We also show in Figure 12(c) the "reduced plan diagram", which shows the extent to which the original plan diagram may be simplified (by replacing some of the plans with their siblings in the plan diagram) without increasing the cost of any individual query by more than a user-specified threshold value – in this case, the value was set to 20%. The observation has been that cost-increase threshold is sufficient to bring the plan cardinality in the reduced diagram to "anorexic levels", that is, a small absolute number within or around ten.

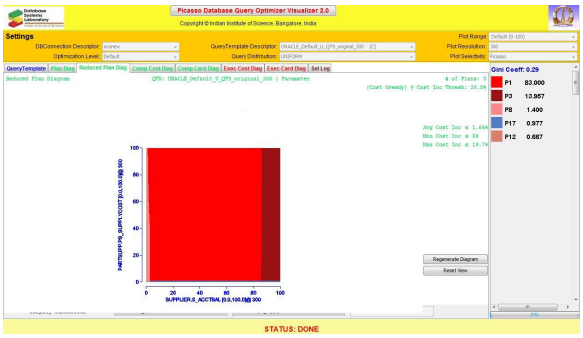
Now consider the situation where the Oracle optimizer developer would like to assess its behavior on the highest scale of the TPC-H benchmark, which runs to 100 TB, *five* orders of magnitude larger than the baseline size. Even in the highly unlikely event that provisioning this extremely large space were to be feasible, the time overheads of generating and loading the database are likely to prove impractical. On the 64GB laptop, this scenario is clearly out of the question. However, using the ConstructMode of



(a) Plan Diagram



(b) Cost Diagram



(c) Reduced Diagram

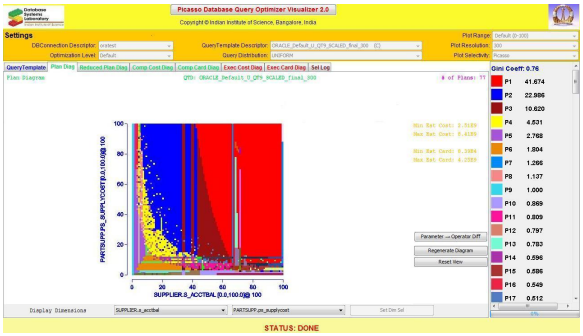
Figure 12: **Diagrams for 1GB TPCH database (QT9)**

CODD, we can easily create a meta-data shell that represents the 100 TB environment.

The suite of diagrams (*plan*, *cost*, *reduced plan*) produced by the Oracle optimizer on the *scaled* database is shown in Figure 13. Comparing the two sets of pictures, we observe first that the number of plans has now increased from the 32 of the baseline to 77 in the scaled version! Secondly, the geometries of the optimal plan regions have undergone a significant change. Turning our attention to the cost diagrams, we see that the slope of the cost diagram has almost doubled in the scaled version. Finally, while the reduced diagrams in both cases are essentially anorexic, the number of surviving plans in the scaled setup (15) is considerably more than that of the baseline environment (5).

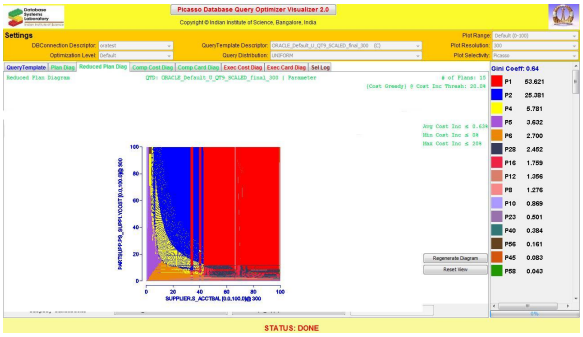
If we drill down further into the plan structures, we find that there are significant differences between the largest plan P1 of the two plan diagrams, although taking up roughly the same area (42%) in both diagrams and in similar locations. Specifically, they are different both in the join order and in the join operators. While in the baseline, the join order is PART $\bowtie$ LINEITEM $\bowtie$ SUPPLIER $\bowtie$ PARTSUPP $\bowtie$ ORDERS $\bowtie$ NATION, in the scaled version it is changed to PART $\bowtie$ LINEITEM $\bowtie$ PARTSUPP $\bowtie$ ORDERS $\bowtie$ SUPPLIER $\bowtie$ NATION. Further, while the baseline used purely hash joins, some of them are replaced by merge joins in the scaled version. These differences are shown in detail in Figure 14, which captures the plan structure of plan $P1$ in both environments.

The above experiment highlights how we can easily assess, using CODD, the optimizer's altered behavior in response to futuristic scenarios.



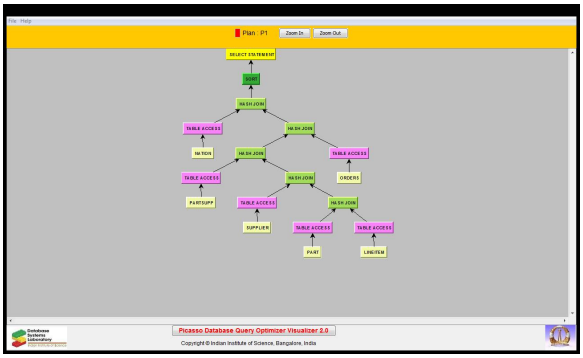
(a) Plan Diagram

(b) Cost Diagram

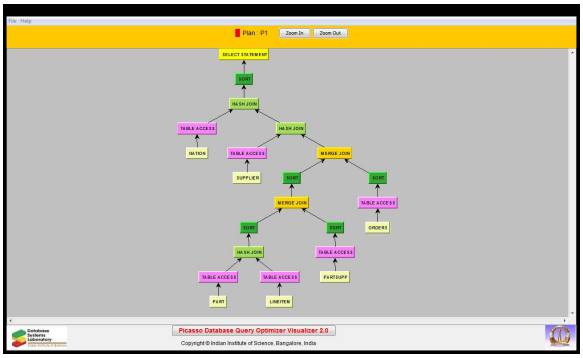


(c) Reduced Diagram

Figure 13: **Optimizer Diagrams for 100TB TPC-H database (QT9)**



(a) Baseline



(b) Scaled

Figure 14: **Plan P1 Operator Tree**

7 Future Work

Performance monitoring and physical design tuning are two crucial parts of any database testing procedures and hence the ability to produce alternative scenarios for these tasks will prove to be very useful. Unfortunately, as these tasks require data, the same limiting factors of time and space constraints as described earlier pose a hindrance.

A very natural extension to CODD will be an attempt to alleviate these constraints for execution

scenarios. One vital approach that we intend to do is to create what can be called "semi-data" environments using CODD by coupling it with the on-the-fly synthetic generator (e.g. [5]). The objective of this idea can be described as follows:

Given a query Q and a metadata instance M, one can traverse through the query tree T such that: At each level i the expected result R_i for that level can be estimated and then using this estimated result R_i and the data characteristics obtained from M, one can generate the data required for that level on-the-fly. Once the data is generated, the execution task for the level i can be finished and the performance be measured. Moving towards the next level i+1, only the required data can be propagated while the base data generated at the start of current level i can be deleted. This idea is still in an infant stage and opens a new research direction to make it completely realizable.

8 Summary

In this report, we presented the technical details of CODD tool, which permits users to construct dataless environments and simulate various alternative scenarios that may prove helpful in testing and debugging exercises. While allowing the user to play with arbitrary metadata values, CODD makes sure that these values are legal and consistent with the engine requirements. In addition, CODD provides two key features – Cost based Scaling and Inter-Engine Portability – that we expect would be of considerable benefit to testers. Finally, the tool is implemented with a convenient graphical interface that helps users to employ the features while remaining agnostic to the specifics of the underlying database engine. The CODD tool currently only supports the testing of meta-data based modules, but we are actively investigating its integration with data generation frameworks so as to facilitate inclusion of execution module testing.

References

[1] C. Binnig, D. Kossmann, E. Lo and M. Tamer Ozsu, "QAGen: Generating Query-Aware Test Databases", *SIGMOD 2007*.

[2] T. H. Cormen, C.E. Leiserson, R. L. Rivest, C. Stein "Introduction to Algorithms", *Third Edition*, 2009.

[3] P. Darera, Personal Communication, June 2010.

[4] A. El Helw, I. Ilyas and C. Zuzarte, "StatAdvisor: Recommending Statistical Views", *VLDB 2009*.

[5] D. Fechner, "Distribution statistics uses with the DB2 optimizer", `www.ibm.com/developerworks/data/library/techarticle/dm-0606fechner/index.html`. *June 2006*.

[6] A. Hulgeri and S. Sudarshan, "Parametric Query Optimization for Linear and Piecewise Linear Cost Functions", *Proc. of VLDB 2002*.

[7] S. Kapoor and K. Truuvert, "Recreate optimizer access plans using db2look", `www.ibm.com/developerworks/data/library/techarticle/dm-0508kapoor/`, *Aug 2005*.

[8] E. Miner, "Using Optdiag Simulate Statistics Mode", `m.sybase.com/detail?id=20472`

[9] M. Muralikrishna, "Using the Optimizer to Generate an Effective Regression Suite: A First Step ", *DBTest 2010*.

[10] T. Rabl and M. Poess, "Parallel Data Generation for Performance Analysis of Large, Complex RDBMS", *DBTest 2011*.

[11] N. Reddy and J. Haritsa, "Analyzing Plan Diagrams of Database Query Optimizers", *Proc. of VLDB 2005*.

[12] D. Slutz, "Massive Stochastic Testing of SQL", *Proc. of VLDB 1998*.

[13] F. Waas, L. Giakoumakis and S. Zhang, "Plan Space Analysis: An Early Warning System to Detect Plan Regressions in Costbased Optimizers", *DBTest 2011*.

[14] "Testing and Tuning of Database Systems", IEEE Data Engineering Bulletin, 31(1), March 2008.

[15] publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0005121.html

[16] dsl.serc.iisc.ernet.in/projects/CODD

[17] dsl.serc.iisc.ernet.in/projects/PICASSO

[18] http://www.jfree.org/jfreechart/

[19] www.tpc.org/tpch

[20] www.tpc.org/tpcds

[21] publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.admin.cmd.doc/doc/

[22] www-01.ibm.com/software/data/data-management/optim-solutions/

[23] www.oracle.com/technetwork/database/enterprise-edition/index-093639.html

[24] publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/t0004590.htm

[25] www.sqlmaestro.com/

[26] www.comp.dit.ie/btierney/oracle11gdoc

[27] download.oracle.com/docs/

[28] download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_stats.htm\#i1035422

[29] psoug.org/reference/truncate.html

[30] support.microsoft.com/kb/914288

[31] msdn.microsoft.com/en-us/library/dd535534(SQL.100).aspx

[32] blogs.msdn.com/b/queryoptteam/archive/2006/07/21/674350.aspx

[33] infocenter.sybase.com/help/

[34] www.postgresql.org/docs/8.4/

A Metadata Consistency Constraints

A.1 DB2 Metadata Consistency Constraints

Table 6 lists the metadata consistency constraints of DB2 Constraint Graph shown in Figure 5.

Constraint	Description
1	CARD must be greater than NPAGES.
2	FPAGES must be greater than NPAGES.
3	The sum of NUMNULLS and COLCARD must be lesser than the CARD in SYSSTAT.TABLES.
4	The number of null values in a column (NUMNULLS in SYSSTAT.COLUMNS) cannot be greater than the cardinality of its corresponding table or statistical view (CARD in SYSSTAT.TABLES).
5	The cardinality of a column (COLCARD in SYSSTAT.COLUMNS) cannot be greater than the cardinality of its corresponding table or statistical view (CARD in SYSSTAT.TABLES).
6	The largest COLVALUE value must have a corresponding entry in VALCOUNT that is equal to the number of rows in the column (CARD in SYSSTAT.TABLES).
7	The sum of the values in VALCOUNT must be less than or equal to the number of rows in the column, which is stored in SYSSTAT.TABLES.CARD.
8	The largest COLVALUE value must have a corresponding entry in DISTCOUNT that is equal to the COLCARD.
9	The number of COLVALUE values must be less than or equal to the number of distinct values in the column, which is stored in SYSSTAT.COLUMNS.COLCARD.
10	HIGH2KEY is greater than LOW2KEY whenever there are more than three distinct values in the corresponding column (COLCARD).
11	In most cases, COLVALUE values should lie between the second-highest and the second-lowest data values for the column, which are stored in HIGH2KEY and LOW2KEY in SYSSTAT.COLUMNS, respectively.
12	
13	In most cases, COLVALUE values should lie between the second-highest and the second-lowest data values for the column, which are stored in HIGH2KEY and LOW2KEY in SYSSTAT.COLUMNS, respectively. There can be one frequent value that is greater than HIGH2KEY and one frequent value that is less than LOW2KEY.
14	
15	The VALCOUNT of a Quantile Histogram bin b must be greater than the sum of VALCOUNTs in the Frequency Histogram whose COLVALUE is less than the b's COLVALUE.
16	NPAGES must be less than or equal to any "fetch" value in the PAGE_FETCH_PAIRS column of any index (assuming that this statistic is relevant to the index).

Continued on next page...

Continued from previous page...

Constraint	Description
17	CARD must not be less than or equal to any ”fetch” value in the PAGE_FETCH_PAIRS column of any index (assuming that this statistic is relevant to the index).
18	INDEXCARD must be equal to CARD.
19	NUMRIDS must be greater than or equal to the INDCARD.
20	If CLUSTERFACTOR is a positive value, It must be accompanied by a valid PAGE_FETCG_PAIRS value.
21	NUM_EMPTY_LEAFS must be less than or equal to the NLEAF.
22	NLEVELS must be less than or equal to the NLEAF.
23	COLVALUE values must be unchanging or increasing with increasing values of SEQNO.
24	VALCOUNT values must be unchanging or increasing with increasing values of SEQNO.
25	DISTCOUNT values must be unchanging or increasing with increasing values of SEQNO.
26	VALCOUNT values must be unchanging or decreasing with increasing values of SEQNO.

Table 6: DB2 Metadata Consistency Constraints

A.2 Oracle Metadata Consistency Constraints

Table 7 lists the metadata consistency constraints of Oracle Constraint Graph shown in Figure 6.

Constraint	Description
1	Cardinality must be greater than Blocks.
2	The sum of NULL Counts and Distinct Values must be lesser than the cardinality of its corresponding table.
3	The number of null values in a column cannot be greater than the cardinality of its corresponding table.
4	The number of distinct values present in a column cannot be greater than the cardinality of its corresponding table.
5	The sum of the values in VALCOUNT must be less than or equal to the cardinality of its corresponding table.
6	The number of COLVALUE values must be less than or equal to the number of distinct values in the column.
7	
8	Index number of rows must be equal to the cardinality of its corresponding table.
9	Number of Distinct Keys in the index must be less than or equal to the index cardinality.

Continued on next page...

Continued from previous page...	
Constraint	**Description**
10	COLVALUE values must be unchanging or decreasing.

Table 7: Oracle Metadata Consistency Constraints

B Lemma Proof

This section presents the proof for Lemma 1 and 2 used in time scaling (Chapter 5).

B.1 Lemma 1 Proof

Notations.

A_k - Domain set of attribute k of relation R

F_k - Frequency distribution of k over A_k

i.e. $F_k : A_k \rightarrow \mathbb{Z}_+$

and $\sum_{a_k \in A_k} F_k(a_k) = Card(R)$.

f_k - Relative frequency distribution of k over A_k

i.e. $f_k : A_k \rightarrow \mathbb{R}_+$

defined as $f_k(a_k) = \frac{F_k(a_k)}{Card(R)} \ \ \forall a_k \in A_k$

and $\sum_{a_k \in A_k} f_k(a_k) = 1$.

We present the proof of Lemma 1 on operator basis.

1. Select (Relational Access) Operator [e.g. Table Scan, Index Scan, Index Seek]

Let A be the relation which is selected with attributes $a_1, a_2, ...a_m$. Let N be the relation cardinality and α_a be the scaling factor of relation A. Let $S_i \subseteq A_i$ be the domain of values selected on attribute a_i after applying the predicate on it (if there is no predicate on a_i, then $S_i = A_i$), where $i = 1, 2, ..m$.

The output size of a select operator on multiple attributes is assumed by the optimizer using attribute value independence assumption whereby the selectivity of each attribute is multiplied. i.e. The output cardinality of $Select\ A_{\{a_i \in S_i, \forall i\}}$ is defined as:

$$\text{Original output cardinality} = N \ * \sum_{v \ \in \ S_1} f_1(v) \ * \sum_{v \ \in \ S_2} f_2(v) \ * \ ... \ * \sum_{v \ \in \ S_m} f_m(v)$$

The scaled cardinality of relation A is given by $N \ * \ \alpha_a$. Hence, the scaled output cardinality of select operator,

$$\text{Scaled output cardinality} = (\alpha_a * N) \ * \sum_{v \ \in \ S_1} f_1(v) \ * \sum_{v \ \in \ S_2} f_2(v) \ * \ ... \ * \sum_{v \ \in \ S_m} f_m(v)$$

$$= \alpha_a * \text{Original output cardinality}$$

This proves that, the output cardinality is expressed as a function of not referenced relations (At leaf level, there is only one relation A, which is not referenced by any other relation in its subtree) scaling factor and original output size ■

2. Join Operator

2.1 Single predicate PK-FK equi-join Operator

We prove the lemma by induction on the level of the operator. Level 0 represents the join nodes, where both of its input node subtree does not have any other join node. Level l join nodes contain exactly $l-1$ join operators in its input node subtree.

Let a, b be the joining attributes of relations A, B respectively, where b is a foreign key referencing to a. Let F_a, F_a be the frequency distribution, f_a, f_b be the relative frequency distribution and D_a, D_b be the domain of joining attributes a, b respectively. Let α_a, α_b be the scaling factors of relations A, B respectively.

Basis Step (For level 0 join nodes): Let N_a, N_b be the output cardinality of join operator input nodes, where N_a, N_b are the cardinality (or cardinality of filtered output tuples if there are base predicates) of relations A, B respectively. The output cardinality of a join operator $A\infty_{a=b}B$ is defined as:

$$\text{Original output cardinality} = N_b * \sum_{v \in D_a \cap D_b} f_b(v)$$

The scaled cardinality of input nodes is given by $N_a * \alpha_a$, $N_b * \alpha_b$. Hence, the scaled output cardinality of join operator,

$$\text{Scaled output cardinality} = (\alpha_b * N_b) * \sum_{v \in D_a \cap D_b} f_b(v)$$
$$= \alpha_b * \text{Original output cardinality}$$

This proves that, the output cardinality is expressed as a function of not referenced relations (Among the join input relations A and B, A is referenced by B and B is not referenced by any one) scaling factor and original output size.

Induction Step (For level $>$ 1 join nodes): We assume, that the claim is true till level $l-1$ and here we prove it for level l. Let N_a, N_b be the output cardinality of input nodes, where N_a, N_b comes from subtree containing relations A, B respectively. The output cardinality of a join operator $A\infty_{a=b}B$ is defined as:

$$\text{Original output cardinality} = N_b * \sum_{v \in D_a \cap D_b} f_b(v)$$

The scaled cardinality of input nodes is given by $N_a * s(\alpha_{a1}, ...)$, $N_b * s(\alpha_{b1}, ..)$, where $s(\alpha_{a1}, ...)$, $s(\alpha_{b1}, ..)$ are functions of scaling factors derived at input node operators. The scaled output cardinality of join operator,

$$\text{Scaled output cardinality}$$
$$= (s(\alpha_{b1}, ..) * N_b) * \sum_{v \in D_a \cap D_b} f_b(v)$$
$$= s(\alpha_{b1}, ..) * \text{Original output cardinality}$$

This proves that, the output cardinality is expressed as a function of not referenced relations (function derived at input node containing relation B in its subtree) scaling factor and original output size ■

2.2 Multiple predicate PK-FK equi-join Operator

This proof is similar to Single predicate PK-FK equi-join except for the additional terms in the original and scaled cardinalities corresponding to multiple predicates. Let $b1$ of B be an another FK attribute corresponding to $a1$ of A. Thus the original and scaled output cardinality of join operator $A\infty_{a=b,\ a1=b1}B$ is written as,

$$\text{Original output cardinality} = N_b * \sum_{v \in D_a \cap D_b} f_b(v) * \sum_{v \in D_{a1} \cap D_{b1}} f_{b1}(v)$$

Scaled output cardinality = $(\alpha_b * N_b) \ * \sum_{v \in D_a \cap D_b} f_b(v) \ * \sum_{v \in D_{a1} \cap D_{b1}} f_{b1}(v)$

Other things follows as Single predicate PK-FK equi-join. Thus Lemma 1 is proved ■

2.3 Other join operators

We prove the lemma by induction on the level of the operator. Let a, b be the joining attributes of relations A, B respectively. Let F_a, F_a be the frequency distribution, f_a, f_b be the relative frequency distribution and D_a, D_b be the domain of joining attributes a, b respectively. Let α_a, α_b be the scaling factors of relations A, B respectively. Let $S_a \subseteq D_a$, $S_b \subseteq D_b$ be the selected values of attributes a, b after applying join predicates on them.

Basis Step (For level 0 join nodes): Let N_a, N_b be the output cardinality of join operator input nodes, where N_a, N_b are the cardinality (or cardinality of filtered output tuples if there are base predicates) of relations A, B respectively. The output cardinality of a join operator $A \infty B$ is defined as (cross product of two relations):

Original output cardinality = $N_a \ * \ N_b \ * \sum_{v \in S_a} f_a(v) \ * \sum_{v \in S_b} f_b(v)$

The scaled cardinality of input nodes is given by $N_a * \alpha_a$, $N_b * \alpha_b$. Hence, the scaled output cardinality of join operator,

Scaled output cardinality = $(\alpha_a * N_a) \ * \ (\alpha_b * N_b) \ * \sum_{v \in S_a} f_a(v) \ * \sum_{v \in S_b} f_b(v)$

= $\alpha_a \ * \ \alpha_b *$ Original output cardinality

This proves that, the output cardinality is expressed as a function of not referenced relations (Among the join input relations A and B are not referenced by each other) scaling factor and original output size.

Induction Step (For level $>$ 1 join nodes): We assume, that the claim is true till level $l-1$ and here we prove it for level l. Let N_a, N_b be the output cardinality of input nodes, where N_a, N_b comes from subtree containing relations A, B respectively. The output cardinality of a join operator $A \infty B$ is defined as:

Original output cardinality = $N_a \ * \ N_b \ * \sum_{v \in S_a} f_a(v) \ * \sum_{v \in S_b} f_b(v)$

The scaled cardinality of input nodes is given by $N_a * s(\alpha_{a1}, ...)$, $N_b * s(\alpha_{b1}, ..)$, where $s(\alpha_{a1}, ...)$, $s(\alpha_{b1}, ..)$ are functions of scaling factors derived at input node operators. The scaled output cardinality of join operator,

Scaled output cardinality

= $(s(\alpha_{a1}, ..) * N_a) \ * \ (s(\alpha_{b1}, ..) * N_b) \ * \sum_{v \in S_a} f_a(v) \ * \sum_{v \in S_b} f_b(v)$

= $s(\alpha_{a1}, ..) * \ s(\alpha_{b1}, ..) \ *$ Original output cardinality

This proves that, the output cardinality is expressed as a function of not referenced relations (function derived at input nodes) scaling factor and original output size ■

3. Aggregate Operator

The size of aggregate operator is 1 and will remain unchanged in the scaled database ■

4. Group by Operator

The output cardinality of a group by operator on an attribute is simply the number of distinct attribute values in it. Since the relative frequency distribution of attribute is retained, number of distinct values in original and scaled relations would be the same. Therefore, for a group by operator, the scaled output size will be same as the original output cardinality ■

5. Sort Operator

Sort operator output cardinality is same as its input cardinality. Hence, for sort operator,

Original output cardinality = N

where N is input cardinality to the sort operator.

After scaling,

Scaled output cardinality = $s(\alpha_a, ..) * N$

= $s(\alpha_a, ..)$ Original output cardinality

where $s(\alpha_a, ..)$ is the function derived at input node.

This proves that, the output cardinality is expressed as a function of not referenced relations (function derived at input node) scaling factor and original output size ■

Similar proof can be written for other operators. Our assumption of retaining relative frequency distribution produces output whose relative frequency distribution of attributes is same as original output. Hence, Lemma 1 is proved ■

B.2 Lemma 2 Proof

Notations.

$C'_a, ...C'_n$ - Referenced PK columns of $C_a, ...C_n$ belonging to the relations $R_a, ...R_n$, respectively.

$d_a, ...d_n$ - Distinct count (number of distinct values) present in the columns $C'_a, ...C'_n$, respectively.

The maximum possible *unique* keys after combining the columns $C'_a, ...C'_n$ is the product of distinct count present in the combing columns, which defines the upper bound on *cardinality* of relation R_i.

$Card(R_i) \leq d_a * ... * d_n$

Domain scaling on key columns, brings the distinct count of columns $C'_a, ...C'_n$ in the scaled relations to be $\alpha_a * d_a, ...\alpha_n * d_n$, respectively. Thus the maximum possible *unique* keys after combining the columns $C'_a, ...C'_n$ of scaled relation is the product of distinct count present in the combing columns of scaled relations, which defines the upper bound on *cardinality* of scaled relation R_i.

$Card(scaled\ R_i) \leq (\alpha_a * d_a) * ... * (\alpha_n * d_n)$

$\implies \alpha_i * Card(R_i) \leq (\alpha_a * ... * \alpha_n) * (d_a * ... * d_n)$

$\implies \alpha_i * Card(R_i) \leq (\alpha_a * ... * \alpha_n) * Card(R_i)$

and $\alpha_i * Card(R_i) > (\alpha_a * ... * \alpha_n) * Card(R_i)$

$\implies \alpha_i \leq \alpha_a * ... * \alpha_n$

and $\alpha_i > \alpha_a * ... * \alpha_n$

$\alpha_i > \alpha_a * ... * \alpha_n$ can happen only if $Card(R_i) < d_a * ... * d_n$ i.e relation R_i does not have all possible *unique* keys. In such scenarios, our scaling implementation does not generate the missing unique values for the scaled database as well. So this scenario is not possible in our scaling implementation and can be ruled out.

$\implies$ Thus, $\alpha_i \leq \alpha_a * ... * \alpha_n$ ■

Example: Let us consider a query workload containing TPC-H relations PART, SUPPLIER and PARTSUPP. Scaling factor of relation PARTSUPP is bounded as follows: $\alpha_{ps} \leq \alpha_p * \alpha_s$, where α_p, α_s and α_{ps} are the scaling factors of relations PART, SUPPLIER and PARTSUPP respectively.