# CODD Metadata Processor

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

**Master of Engineering**

IN

COMPUTER SCIENCE AND ENGINEERING

by

**I. Nilavalagan**


Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

June 2012

TO

*My Family and Friends*

# Acknowledgements

I am grateful to Prof. Jayant R. Haritsa for his guidance, enthusiasm and supervision. I would like to sincerely acknowledge his invaluable support in all forms throughout my stay in IISc.

I would like to thank all the members of DSL for all the help and suggestions. Also I would like to thank all my CSA friends who have made my stay at IISc memorable.

Finally, I am indebted with gratitude to my parents for their constant support and motivation throughout my career.

# Publications based on this Thesis

Rakshit S. Trivedi, I. Nilavalagan and Jayant R. Haritsa

"*CODD: COnstructing Dataless Databases*"

Proc. of 5th Intl. Workshop on Testing Database Systems (DBTest), Scottsdale, USA, May 2012.

# Abstract

*Design and testing of database engines and applications requires to construct and evaluate various alternative scenarios with respect to the database contents. To construct a scenario, existing methodology consumes space and time at least proportional to database size. This may limit the desired scenarios and/or make it infeasible to model them. In this work, we present* **CODD***, a graphical tool that alleviates the time and space constraints through the construction of "dataless databases". Specifically, CODD implements a unified visual interface through which databases with the desired metadata characteristics can be efficiently simulated without the explicit presence of data. Input metadata values are validated to ensure that the simulated database is both legal and consistent. Additionally CODD provides two other features, which is relevant to database test teams. First, it provides automated metadata transfer across database engines to facilitate comparative study of systems. Second, it supports space and time based metadata scaling. CODD is currently operational on a rich suite of popular database engines. We present here the ability of CODD to construct alternative scenarios and its various features.*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Design and testing of database engines and applications requires to construct and evaluate various alternative scenarios with respect to the database contents. Evaluating such various scenarios exercises different segments of the codebase, or profiles module behaviour over a range of parameters [4, 5, 6]. To construct the scenario, existing methodology consumes space and time at least proportional to database size. Generating data and loading it to database are the major time consuming processes in constructing the scenario. This may limit the desired scenarios and/or make it infeasible to model them. In Database Management Systems there exists a class of important functionalities, such as query plan generators, system monitoring tools and schema advisory modules for which the inputs comprise solely the *metadata*, derived from the underlying data. For such functionalities, developing a software that creates a database instance with only metadata (i.e. without associated raw data) would be extremely useful. In this work, we present **CODD** [1], a graphical tool that supports the *ab initio* creation of metadata. CODD provides the following modes of operations, which covers the construction of various alternative scenarios and various features of CODD.

**1. Metadata Construction:** This mode lets the user construct various alternative scenarios ranging from *empty* to *Big-Data* (for instance, *yottabyte* [$10^{24}$]) sized relational tables, *uniform* to *skew* attribute-value distribution, etc., without requiring the presence of any prior data instance. CODD provides engine specific metadata input interface, wherein the user has to input

[1] In archaic English, *cod* means "empty shell", symbolising our dataless context.

relation cardinality, attribute-value distribution and other metadata statistics of the desired scenario. CODD also provides a graphical editing interface to alter attribute-value distributions visually. Further, CODD incorporates a graph-based validation algorithm to ensure that the input metadata values are both *legal* (valid range, correct type) and *consistent* (compatible with the other metadata values). After validating input metadata statistics, they are updated into catalogs and thus the construction of desired scenario is completed. The whole process of metadata input including visual alteration of histograms, validation and catalog updates can be completed in few minutes.

**2. Metadata Retention:** In environments where metadata statistics are required to be sourced only from the actual data, CODD makes it feasible to subsequently *drop* the raw data without affecting the metadata statistics and gets back the raw data storage space.

**3. Metadata Porting:** CODD facilitates comparative studies of different systems by porting the metadata statistics across database engines. Given source and destination database engines, CODD transfers the metadata statistics from source to destination database engine based on the predefined metadata statistics *mapping* between them.

**4. Metadata Scaling:** Testing on scaled versions of original database is a usual practice in database engine testing. CODD achieves the scaling on metadata. It produces the scaled metadata instance given a baseline metadata instance and a user-specified scaling factor. For example, given a metadata instance of 1GB and scaling factor 100, CODD produces the scaled metadata instance of 100GB. This is achieved by *space scaling* models that mimic the TPC-H [16] and TPC-DS [15] data generators.

In addition, CODD provides a novel *time scaling* model in which metadata instance is scaled such that the overall estimated execution time of a query workload is scaled by the user-specified scaling 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.

In a nutshell, CODD is an easy-to-use graphical tool for the automated creation, verification, retention, porting and scaling of database metadata configurations.

## 1.1 Organization

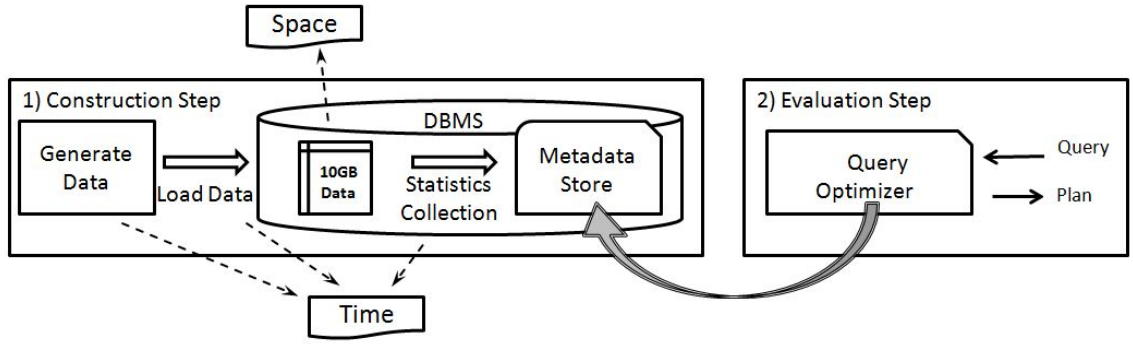
The remainder of this thesis is organized as follows: Chapter 2 presents the motivation behind the development of CODD. The overview of CODD tool is presented in Chapter 3. The process of ab initio metadata creation is explained in Chapter 4. This chapter also includes the details of visual construction of histograms and metadata validation process. Chapter 5 gives a detailed description of metadata retention process and Chapter 6 presents the metadata inter-engine portability. Metadata scaling is explained extensively in Chapter 7. Finally, in Chapter 8, we summarize our conclusions and outline future works.

# Chapter 2

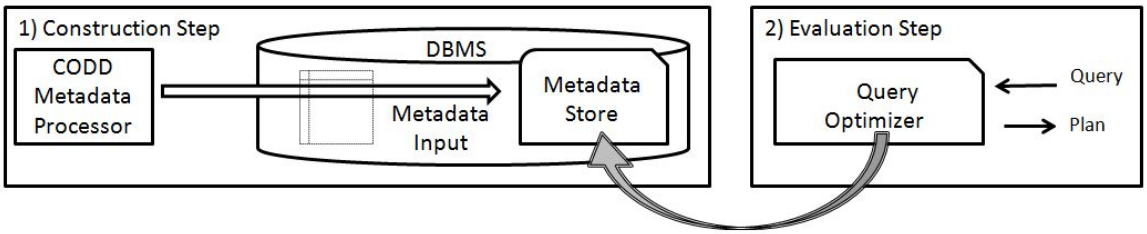
# Motivation

Let us consider a simple scenario of evaluating database query optimizer with relations of size 10GB. Figure 2.1(a) shows the execution of this scenario, which involves two steps: First, construction step, desired database scenario (10GB sized relations) is constructed. The whole process of construction step includes the sub-steps of data generation, data loading and statistics collection. The first two sub-steps take time at least proportional to the database size (10GB). Loading of data consumes additional time to check for referential integrity constraints, if such constraints are present in the schema. Time taken by the last sub-step is dependent on the method by which it is done. Statistics collection with database scan takes time proportional to database size (10GB) whereas sampling takes constant time. Database consumes space at least of database size (10GB) to store the data. Additional space is consumed if physical schema constructs like indexes are present. Second, evaluation step, query optimizer is evaluated on the constructed database scenario by obtaining the execution plan for the input query. The evaluation of query optimizer depends only on the metadata store, which has the metadata of the data. But the metadata store gets its metadata statistics of data only after the laborious construction step.

The two step procedure is repeated for executing each of the various alternative scenarios. It is clear from the two step procedure that the laborious construction step is a bottle neck in the execution of scenarios and this may limit the desired scenarios. In worst case, it is infeasible to model some of the desired scenarios such as futuristic Big-Data setup featuring *yottabyte*




(a) Existing execution method of a scenario




(b) Desired execution method of a scenario

Figure 2.1: Motivation test scenario

($10^{24}$) sized relational tables. In such cases, it would be good to have an execution method of a scenario as in Figure 2.1(b), where the required database scenario is created by directly inputting the metadata statistics. In this work, we deliver **CODD Metadata Processor** tool which supports *ab initio* creation of metadata instance.

# Chapter 3

# CODD Overview

CODD Metadata Processor alleviates the time and space constraints in creating the alternative scenarios through the construction of dataless databases. A dataless database is defined by its metadata statistics. In modern database engines, metadata statistics cover a variety of aspects, including schema organization, query processing, workload management, and performance tuning. In CODD, we focus only on the metadata statistics related to query processing and the extension to the other aspects is straightforward. In particular, our metadata statistics include 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).

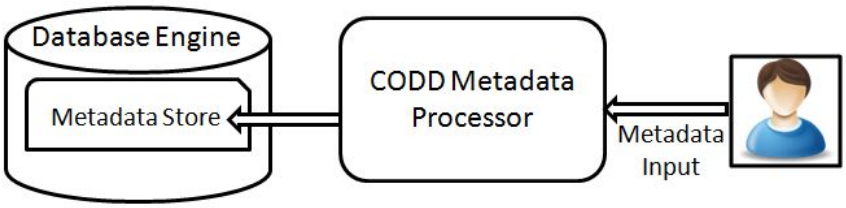



Figure 3.1: CODD Metadata Processor Overview

Figure 3.1 shows the overview of CODD Metadata processor. It provides a vendor-neutral interface through which the user can input the metadata statistics which represent the desired

database scenario. The input metadata values are validated for legality and consistency, then they are updated into the metadata store of database engine to complete the construction of desired database scenario. It also provides other features such as metadata retention, porting and scaling, which are explained in further sections.

CODD is completely written in Java, running to over 40K lines of code, and is operational on a rich suite of industrial-strength database systems including DB2, Oracle and SQLServer. It functions solely through the database APIs in a non-invasive manner. Further, the graphical interfaces are designed such that the user can focus only on the logical metadata semantics without knowing about the implementation specifics of individual engines. The tool is freely downloadable at [8].

# Chapter 4

# Metadata Construction

In this section, we showcase the ability of CODD to create or edit the metadata statistics without requiring presence of any prior data instance, followed by the visual construction of histograms and the technical details of underlying implementation of Metadata Construction in CODD. A fundamental concern in ab initio creation of metadata is to ensure the input metadata values are legal and consistent. CODD implements a graph based model to validate the input metadata values and the validation process is described in the Section 4.4.

## 4.1 Ab initio metadata construction

In this section, we show the steps involved in constructing a metadata-only data instance for DB2. Other database engines have a similar procedure. The relations (logical schema) are created first followed by indexes (physical schema) and then the metadata construction procedure is started.

Figure 4.1 shows the metadata construct interface for DB2. It features a wide range of metadata statistics, which are grouped into three categories as relation level, column level and index level based on the nature of values it represents. Relation level metadata statistics includes the relation cardinality and page count. Column level includes the metadata statistics specific to a column such as the number of distinct values, number of null values, average

column length (for string type columns), DB2 specific HIGH2KEY, LOW2KEY and data distribution (histograms). Metadata entities HIGH2KEY and LOW2KEY signify the second-highest and second-lowest values in the column, respectively. Index level metadata statistics include B+-tree information such as counts of levels, leaf pages, empty leaf pages, density and cluster factor.

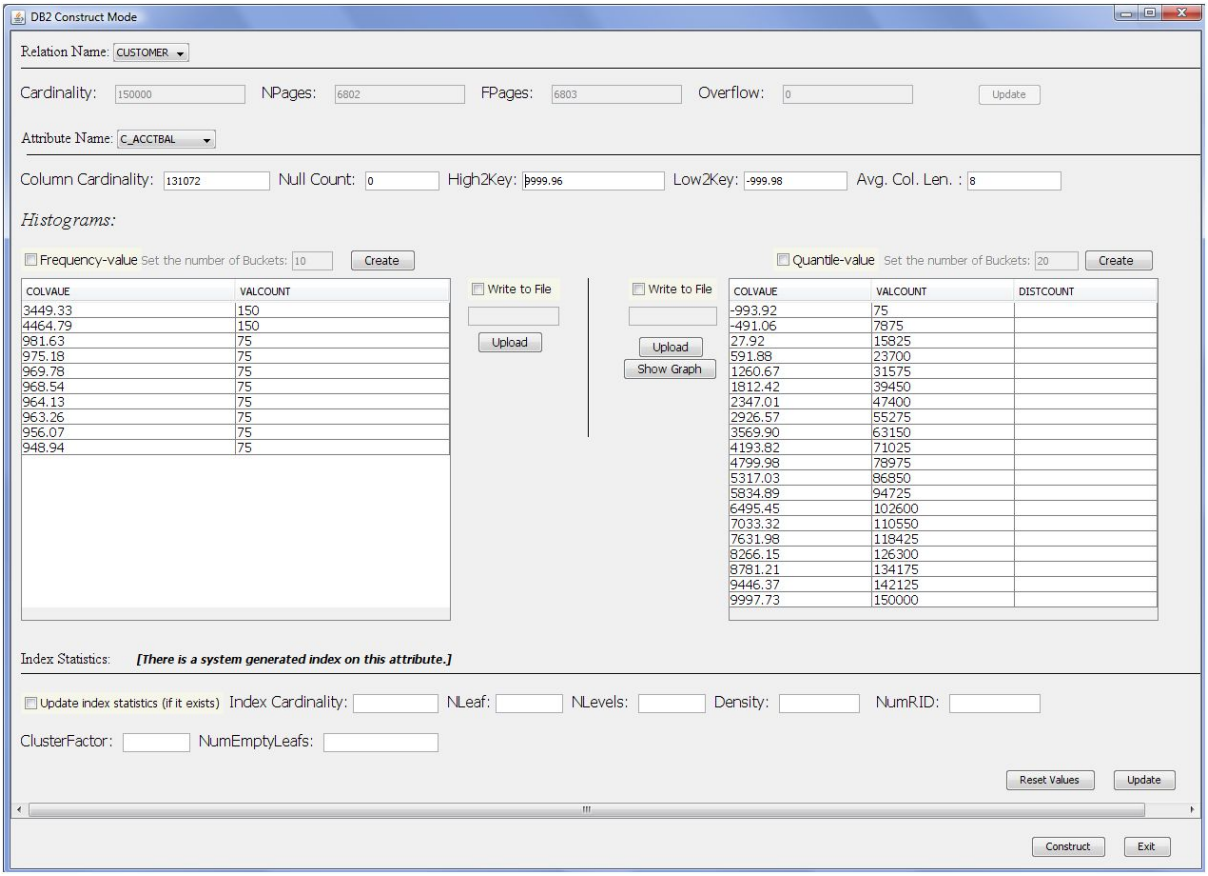



Figure 4.1: CODD Interface (Metadata Construction on DB2)

For each relation, user inputs the relation level metadata statistics, followed by the column level and index level metadata statistics for selected attributes of the relation using the Update button. After each update (relation level or column level), CODD validates the input metadata statistics and reports the user with the appropriate error message, if there is a validation error. For example, if the entered HIGH2KEY value is less than the LOW2KEY value, then an error message appears asking user to re-enter the correct values.

DB2 hosts two kinds of column distribution statistics: First, *frequency histograms* corresponding to the most common values. Second, *quantile histograms* which summarizes the data distribution with a set of buckets [2]. For both these histogram types, the CODD interface allows the user to input either manually or from a file. Subsequently, the constructed histogram can be viewed graphically and its layout can be visually altered to the desired geometry (reflecting the data distribution) by simply reshaping the bucket boundaries. Section 4.2 describes the details of visual histogram and its implementation details. After inputting the required metadata statistics the user can click on the Construct button to do the actual construction of data instance by updating the database catalogs with the provided metadata values.

## 4.2 Graphical Histogram Details

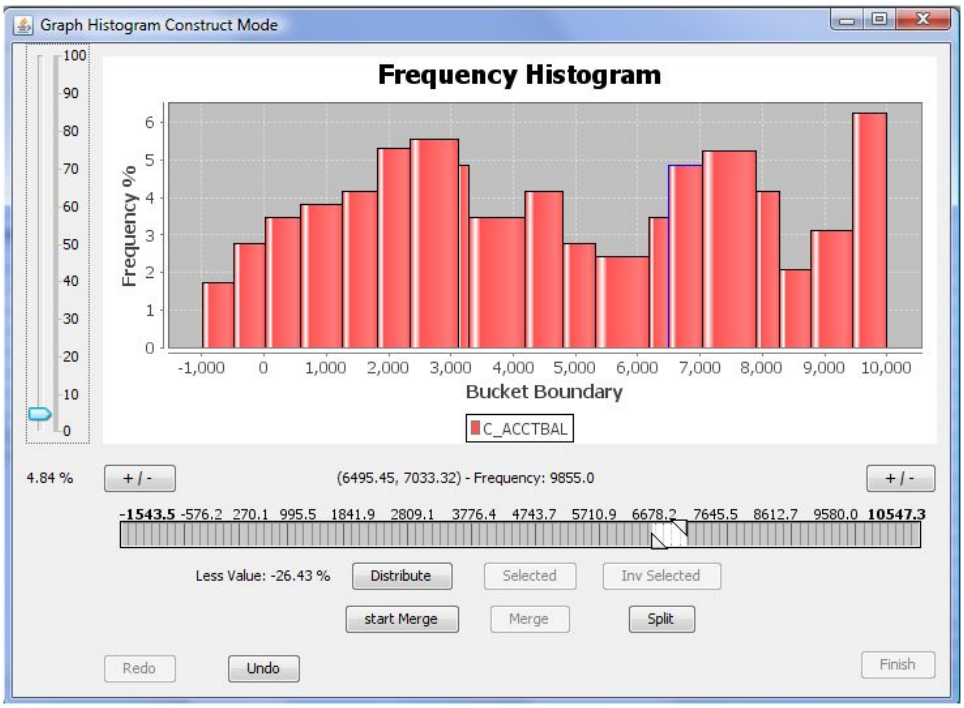



Figure 4.2: Graphical Histogram Interface

CODD provides a feature to modify the data distribution of a column through graphical

interface. We use JFreeChart [13] to implement the graphical histogram. JFreeChart is a free chart library to produce charts and graphs with extensive set of features. Figure 4.2 shows the modified graph histogram instance of S_ACCTBAL column of TPC-H SUPPLIER relation. The Graph Histogram takes the total row count, total distinct values (distinct count) present in the column and an initial histogram as input to produce the initial graph histogram. Histogram is a set of buckets, where each bucket associates a range of column values (specified by its minimum and maximum values) to count(i.e frequency) and distinct count of bucket range values present in the column. Graph histogram shows each bucket as a bar in the graph, where height represents the frequency or distinct count in percentage and width represents the range. For the initial histogram and the final modified histogram, the total frequency and distinct count percentage must be 100.

The graphical histogram is used in two modes of operation as given below:

- Frequency Mode - The graph represents and operates on the frequency values of the buckets.
- Distinct Count Mode - The graph represents and operates on the distinct count values of the buckets.

The user can reshape the graph to get the desired data distribution. Graph is reshaped through a set of operations with mouse and buttons in the graph histogram interface. CODD supports the following reshaping operations:

- 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.

## 4.3 Implementation Details

In this section, we present the mechanism by which the input metadata statistics are added or updated into the metadata store (catalog tables) of CODD supporting database engines. If the user has not provided the input values for any of the metadata, then engine specific default values are used for it.

**DB2.** Catalog tables SYSSTAT.TABLES, SYSSTAT.COLUMNS, SYSSTAT.COLDIST and SYSSTAT.INDEXES stores the relation, column, column data distribution and index level metadata statistics respectively. DB2 supports only UPDATE command on these tables and fresh addition to it is possible only through the statistics collection done by RUNSTATS command. So CODD performs the ab initio metadata construction in two steps: First, it populates the catalog tables by executing RUNSTATS command, and then updates (UPDATE command) them with the user provided input metadata statistics.

**Oracle.** Catalog tables ALL_TABLES, ALL_TAB_COL_STATISTICS, ALL_TAB_HISTOGRAMS and ALL_IND_STATISTICS stores the relation, column, column data distribution and index level metadata statistics respectively. Oracle does not support direct insert or update on these catalog views. However, it provides sub-programs SET_TABLE_STATS, SET_COLUMN_STATS and SET_INDEX_STATS through DBMS_STATS package [7] to update the catalog tables. Each of these sub-program takes the metadata statistics value as arguments. All the arguments are of simple data type except for column data distribution, which requires the input to be in a special internal representation. CODD uses PREPARE_COLUMN_VALUES sub-program of DBMS_STATS package to convert the user input to the required internal representation. CODD defines dynamic SQL procedures to prepare user provided metadata statistics and to call these

*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 4.3: Procedure to update statistics in Oracle

sub programs with them. Then the dynamic procedures are executed, which update the catalogs and thus completes the ab initio metadata construction. Figure 4.3 shows a code snippet which is used in CODD to implement the metadata construction.

**SQLServer.** Catalog tables SYS.SYSOBJVALUES and SYSINDEXES store the relation and index level metadata statistics respectively. Column data distribution is stored as a large binary object (STATLOB) in SYS.SYSOBJVALUES. The catalog tables are not directly accessible to users [9]. However, commands UPDATE and CREATE STATISTICS have an 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 [9] in conjunction with the DBCC SHOW_STATISTICS command. Since its format is currently proprietary, it is not possible to directly create or edit these hexadecimal values. So currently CODD does not support metadata construction for SQLServer. However, these commands can be used in inter engine metadata statistics transfer as described in Chapter 6.

## 4.4 Metadata Validation

In metadata construction, user inputs the metadata statistics. Before updating these values to the database catalogs, we need to ensure that each of the input metadata value satisfies the following two constraints:

- **Structural Constraint:** Input metadata value must be of *specified type* and it must fall in the *specified range* of values. For example, *cardinality* of a relation must be an integer type and the value must be greater than or equal to zero.
- **Consistency Constraint:** Input metadata value must be compatible with other metadata values. For example, *number of distinct values* present in a column of a relation must be less than or equal to the *cardinality* of that relation.

In our validation process, we first construct a *directed acyclic constraint graph*, which represent all the metadata entities along with its structural and consistency constraints precisely as follows: 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 given by user and its 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 between the two nodes. Directions on the edges specifies the traversal order.

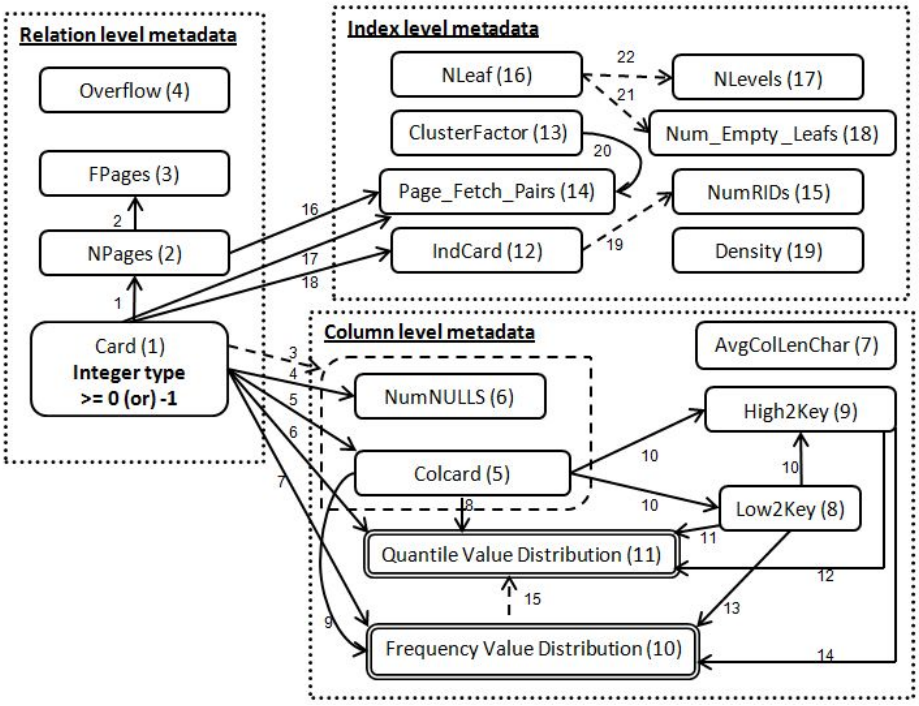
Since the consistency constraints are typically bi-directional, we keep a directed edge to prevent duplicate edges. So the consistency constraint between any two nodes will be represented as a directed edge between them, where the direction is decided based on the following rules. These rules are formulated such that it reflects the natural way in which schemas are usually developed by human users.

- If the nodes are at different levels, then direction is added from the node at higher level of abstraction to the node at lower level (e.g from relation to column level, relation to index level).
- If the nodes are at same level, then direction is added from the node which represents the aggregate information to the node representing detailed information (e.g for column level nodes, direction is added from number of distinct values node to data distribution node).
- For other nodes, lexicographic ordering of nodes is used to decide the direction.

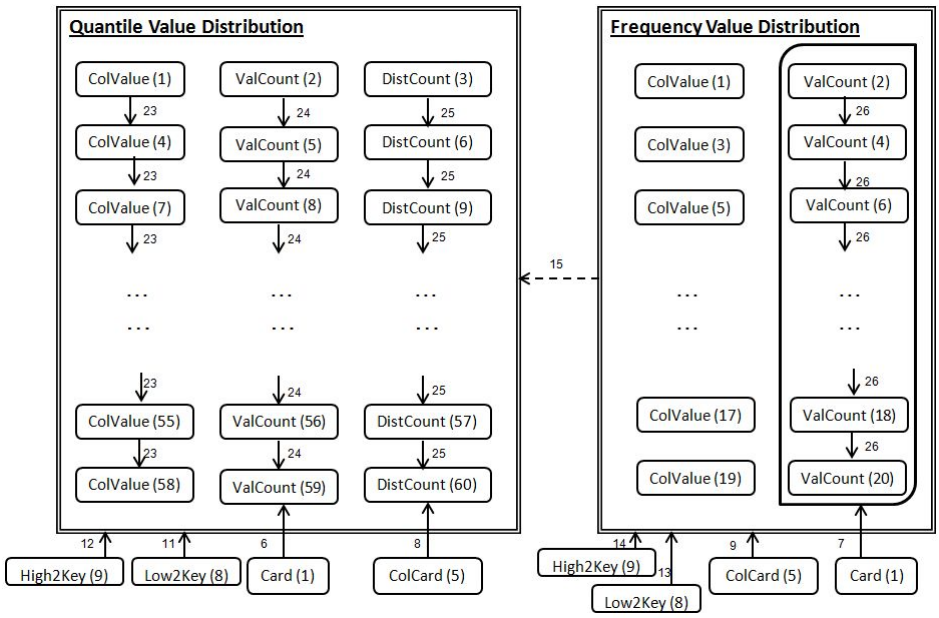
Figure 4.4(a) shows the constructed directed acyclic constraint graph for DB2. The graph nodes are populated from the Catalog table fields covering relation level, column level and index level metadata entities. A sample structural constraint is shown for node CARD, which represents the cardinality of the relation. The structural constraints of this node specifies that the value should be a whole number (or the default value -1 signifying that the statistics is not collected). The graph edges are added based on the consistency constraints between metadata entities as listed in [11]. For example, edge connecting nodes CARD and COLCARD represents the consistency constraint COLCARD $\leq$ CARD.

We observed that some applicable constraints are not listed in [11] and/or not enforced during update. So we have added all such applicable constraints and shown as dashed edges in constraint graph. Specifically the following are a few of the added constraints:

- 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




(a) Constraint Graph




(b) Super nodes of Constraint Graph

Figure 4.4: DB2 Metadata Constraint Graph

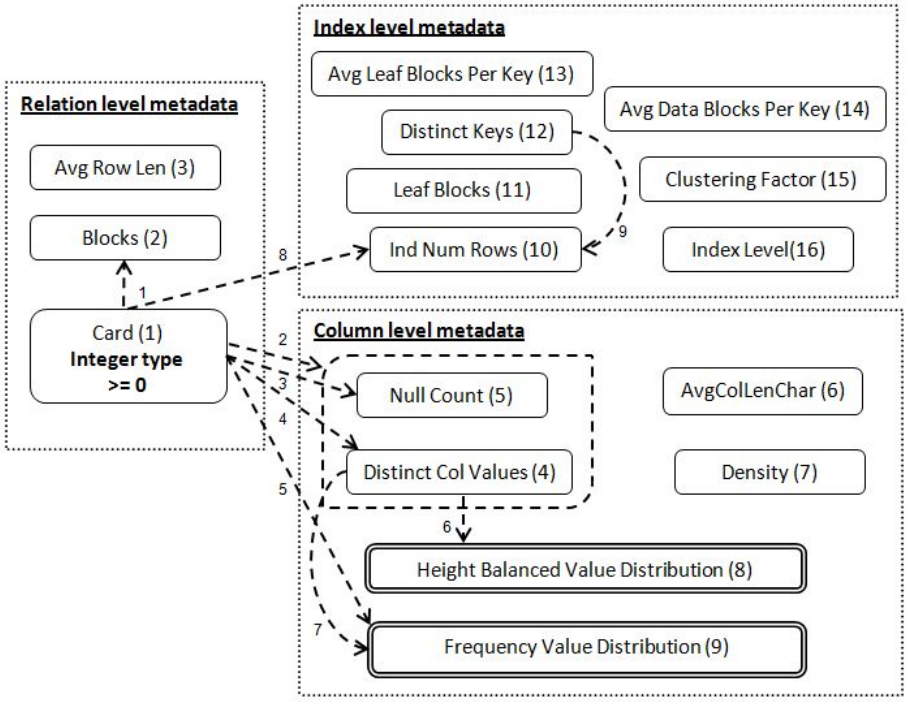
VALCOUNTs in the Frequency Histogram whose COLVALUE is less than the Quantile Histogram bin COLVALUE.

Numbers on the edges (in Figure 4.4(a)) represents the constraint numbers and the corresponding constraints are listed in Appendix A. Figure 4.4(a) shows the nodes QUANTILE VALUE DISTRIBUTION, FREQUENCY VALUE DISTRIBUTION with double line border. These nodes are called as "super-node", as they are representing a graph inside it. Figure 4.4(b) shows the expanded graph of the super-nodes, where the histogram bin values, frequency and distinct count are represented as nodes and the ordering of the values are represented as edge constraints. Dashed edge between QUANTILE VALUE DISTRIBUTION and FREQUENCY VALUE DISTRIBUTION shows that the former distribution is constrained by the latter one.

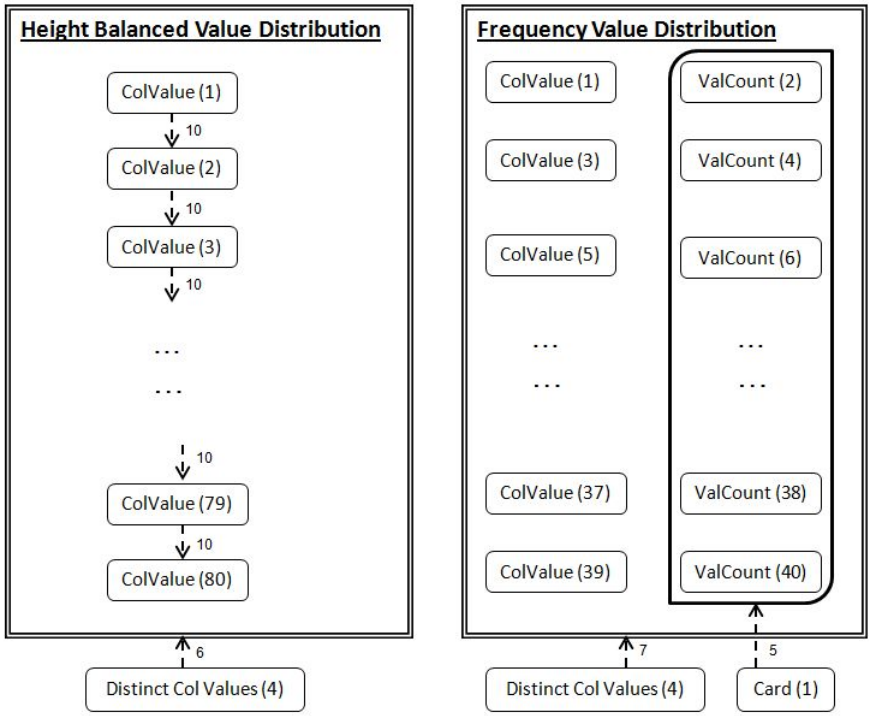
The constraint graph has a complex structure. It has a few independent nodes as well as highly connected nodes. Node CARD has the highest outdegree of 8, which is referenced by many other nodes. The total no. of nodes in the graph is 99 ( = 4 + 7 + [60 (Q) + 20 (F)] + 8), assuming that there are 20 Quantile 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 constructing the constraint graph $G(V, E)$, 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 $O(|V| + |E|)$ [1]. A sample linear ordering is shown through the numbers associated with the nodes in Figure 4.4(a), beginning with CARD (1) and ending with DENSITY (19). After getting the input from the user through CODD metadata construct interface (Figure 4.1), the input values are validated by traversing through the linear ordering and at each node validating the structural and consistency constraints. If any of the constraint is not satisfied, then it is reported to the user. After validating all the user input metadata values, they are updated into the database catalogs to complete the metadata construction.

Similarly for other engines, the constraint graph is constructed and topological sorted constraint graph validation is incorporated into CODD. Figure 4.5 shows the constraint graph for Oracle database engine and the constraints are listed in Appendix A.




(a) Constraint Graph




(b) Super nodes of Constraint Graph

Figure 4.5: Oracle Metadata Constraint Graph

# Chapter 5

# Metadata Retention

Metadata retention drops the data and reclaims the raw data space back without affecting the metadata statistics. This can be helpful in environments where

- metadata statistics are required to be sourced only from the actual data. As a case in point, testers would like to temporarily load the real-world database scenarios without incurring the storage and maintenance overheads of data during the testing process.
- database is already loaded with data and now the user wants to get rid of the data to get the raw data space back.

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. The challenge, of course, is to do so without this removal being reflected in the metadata, since data updates automatically activate engine triggers that refresh the catalogs (AutoStatsUpdateTrigger). The following subsection provides the implementation details of Metadata Retention along with the handling of AutoStatsUpdateTrigger.

## 5.1 Implementation Details

In this section, we present the mechanism by which the data is dropped without affecting the metadata statistics for CODD supporting database engines. RDBMS referential integrity constraint does not allow a foreign key relation tuple to be present without its corresponding

primary key relation tuple. So if a relation's data has to be dropped, all its dependent relations must be dropped before dropping the relation. For example, in TPC-H benchmark, dropping REGION requires NATION to be dropped first as NATION refers to REGION by N_REGIONKEY. Given the set of relations to drop the data and CODD finds the dependent relations and adds them to the drop list.

**DB2.** Since the referential constraints do not allow the data to be deleted from the relation, all the referential constraints of drop list relations are dropped first. Then the data is deleted and referential constraints are created back. Command TRUNCATE 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. Automatic maintenance of statistics is stopped by setting the AUTO_MAINT configuration to OFF for the database.

**Oracle.** In order to delete the data, all the referential constraints are disabled first. Then the data is deleted and the constraints are enabled back. Command TRUNCATE deletes the data from the relation and also releases the space occupied by the raw data. The catalog tables are locked using DBMS_STATS package to block the update on catalog tables during data drop operation. Disabling auto optimizer stats collection stops the automatic maintenance of statistics.

**SQLServer.** Availability of scripting facilities makes Metadata Retention easy to implement as it substantially overlaps with the natively available metadata scripting. 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 metadata information, using SQL Server Management Objects (SMO) [10] in windows power-shell program. Then the drop list relations are completely eliminated from the database using DROP command. To reclaim the storage space, the SHRINK DATABASE command is used on the *mdf* database file. Then the generated script file is run against the database to first recreate the schemas of the relations that were dropped, and then to restore their statistics. Figure 5.1 shows the shell

program used to script the database objects along with its metadata.

```
function global:RetainScript([string]$server, [string]$dbname, [string]$table) {
[System.Reflection.Assembly]::LoadWithPartialName ("Microsoft.SqlServer.SMO")
$SMOserver = New-Object ('Microsoft.SqlServer.Management.Smo.Server') -argumentlist $server
$db = $SMOserver.databases[$dbname]
$Objects = $db.Tables
$SavePath = $($dbname) + "\\"
foreach ($ScriptThis in $Objects where | {!($_.IsSystemObject) }) {
 $ScriptFile = $ScriptThis -replace "\[|\]"
 if($ScriptFile -eq $table) {
  $scriptr = new-object ('Microsoft.SqlServer.Management.Smo.Scripter') ($SMOserver)
  $scriptr.Options.AppendToFile = $False
  $scriptr.Options.AllowSystemObjects = $False
  $scriptr.Options.ClusteredIndexes = $True
  $scriptr.Options.DriAll = $True
  $scriptr.Options.DriIncludeSystemNames = $True
  $scriptr.Options.ScriptDrops = $False
  $scriptr.Options.IncludeHeaders = $True
  $scriptr.Options.ToFileOnly = $True
  $scriptr.Options.Indexes = $True
  $scriptr.Options.Permissions = $True
  $scriptr.Options.WithDependencies = $False
  $scriptr.Options.Statistics = $True
  $scriptr.Options.OptimizerData = $True
```

**Continued on next page....**

**Continued from previous page....**

```
$ScriptDrop = new-object ('Microsoft.SqlServer.Management.Smo.Scripter') ($SMOserver)
$ScriptDrop.Options.AppendToFile = $False
$ScriptDrop.Options.AllowSystemObjects = $False
$ScriptDrop.Options.ClusteredIndexes = $True
$ScriptDrop.Options.DriAll = $True
$scriptr.Options.DriIncludeSystemNames = $True
$ScriptDrop.Options.ScriptDrops = $True
$ScriptDrop.Options.IncludeHeaders = $True
$ScriptDrop.Options.ToFileOnly = $True
$ScriptDrop.Options.Indexes = $True
$ScriptDrop.Options.WithDependencies = $False

$TypeFolder=$ScriptThis.GetType().Name
"Scripting Out "+$TypeFolder + " " + $ScriptThis
$ScriptDrop.Options.FileName = "" + $($SavePath) + $($ScriptFile) + "-drop.SQL"
$scriptr.Options.FileName = "" + $($SavePath) + $($ScriptFile) + "-metadata.SQL"

$ScriptDrop.Script($ScriptThis)
$scriptr.Script($ScriptThis)
 }
 }
}
```

Figure 5.1: Procedure to script statistics in SQL Server

# Chapter 6

# Metadata Porting

Given the source and destination database engines, Metadata Porting transfers the metadata statistics from source to destination database engine based on the predefined metadata statistics *mapping* between them. It facilitates comparative studies of different systems by porting the metadata statistics across database engines. 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.

As a first step in achieving the Metadata Porting, we have carefully worked out the semantic mapping of metadata statistics across database engines. Although each engine has its own idiosyncratic metadata, like HIGH2KEY and LOW2KEY in DB2, we have found that most of relation, column and index level metadata statistics are portable across database engines. However, the data distribution is stored differently in each of the engines. So we adopted a common canonical representation for data distribution (which resembles DB2 style). During statistics transfer, source data distribution is converted to canonical form and then converted back to target engine format. Table 6.1 shows the metadata statistics mapping between DB2 and Oracle database engines.

The overall feasibility of the metadata transfer across different pairs of engines is summarized in Table 6.2. In this table, a Y entry signifies that most of the metadata statistics can be transferred, whereas a Partial entry means that all of the metadata statistics except column

| Statistics Level | **Oracle** | **Corresponding statistics in DB2** |
|---|---|---|
| **Table** | NUM_ROWS<br>BLOCKS<br>AVG_ROW_LEN | CARD<br>NPAGES<br>- |
| **Attribute** | NUM_DISTINCT<br>NUM_NULLS<br>AVG_COL_LEN | COLCARD<br>NUM_NULLS<br>AVGCOLLEN |
| **Distribution** | Height-Balanced or Frequency Histogram | Quantile and Frequency Histogram |
| **Index** | IND_LEAF_BLOCKS<br>IND_LEVELS<br>CLUSTERING_FACTOR | NLEAFS<br>INDLEVEL<br>CLUSTERFACTOR |

Table 6.1: DB2-Oracle Mapping

level data distribution can be transferred, while the N entry indicates that the transfer is infeasible. As can be seen, it is only with SQLServer to which conversion is not possible due to its proprietary format for communicating statistics. And the diagonal entries, where the metadata statistics is transferred from one database instance to other instance of same engine, intra-engine transfer, is always possible with 100% transformation.

| Engine | **DB2** | **Oracle** | **SQLServer** |
|---|---|---|---|
| **DB2** | - | Y | N |
| **Oracle** | Partial | - | N |
| **SQLServer** | Y | Y | - |

Table 6.2: Inter-Engine Metadata Transfer

## 6.1 Implementation Details

CODD provides the Metadata Porting in two ways as follows:

(a) **on-line transfer.** Source and target database instances are up and running. Metadata statistics are read from source, transformed and written into target engine catalogs.

(b) **off-line transfer.** Only the source database instance is up and running. Metadata statistics are read from source, transformed into canonical form and written into a file. Later (after the target database instance is up and running) the file is read and metadata statistics are transformed into target engine specific format and written into the target engine catalogs.

In summary, metadata statistics porting involves three steps: *read*, *transform* and *write* metadata statistics. Step *read* is described later in this section for each of the CODD supporting database engines. Step *transform* is achieved by the worked out metadata statistics mapping across the database engines. All the mappings are incorporated into CODD and the appropriate transformation is done based on the source and target database engines. The tail step, *write*, is achieved as in Metadata Construction where the transferred metadata statistics is the user input.

Though the database engines provide native scripting facilities (db2look utility [3] in DB2 and Script Wizard [12] in SQLServer) to transfer (inter-engine) the metadata statistics, CODD automates the steps in transfer and packages them such that the user need not know about the internal details.

**DB2.** SELECT commands on the catalog tables SYSSTAT.TABLES, SYSSTAT.COLUMNS, SYSSTAT.COLDIST and SYSSTAT.INDEXES are used to read metadata statistics.

**Oracle.** Sub-programs GET_TABLE_STATS, GET_COLUMN_STATS and GET_INDEX_STATS provided by DBMS_STATS package [7] are used to read metadata statistics.

**SQLServer.** Command DBCC SHOW_STATISTICS is used to read metadata statistics which can be ported to other database engines. But other engine metadata statistics can not be ported to SQLServer as it does not support Metadata Construction. However, intra-engine transfer is possible. We have used native scripting facility to transfer the metadata statistics. Specifically, we have automated the procedure described in [12] to complete the transfer. We have written a dynamic windows power-shell program using SQL Server Management Objects (SMO) [10] to script the metadata statistics information. Later the script is run on the target engine to recreate the same metadata statistics environment.

# Chapter 7

# Metadata Scaling

A common activity in database engine testing exercises is to assess the behaviour 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.

CODD supports these space-based scaling models of TPC-H and TPC-DS. In addition, it also provides a novel *time-based* scaling model. These two scaling models are explained in the further subsections.

## 7.1 Space Scaling

### 7.1.1 Problem Statement

Given a baseline metadata instance and a user specified scaling factor $\alpha$, produce a scaled metadata instance such that the space (size in Bytes) occupied by the scaled metadata instance is $\alpha$ times of the baseline metadata instance.

### 7.1.2 Approach

We assume the average row length and average column width for each column of scaled database is same as baseline metadata. Thus achieving cardinality scaling of relation produces space scaling on the relation.

### 7.1.3 Implementation Details

Cardinality scaling on metadata is implemented at relation, column and index level. At the relation level, we scale the cardinality, pages or blocks metadata values by $\alpha$. At the column level, cardinality scaling is implemented differently for key columns and non-key columns. For key columns domain scaling (data distribution domain is scaled) is implemented and the distinct count is scaled by $\alpha$. For non key columns, we keep the relative frequency distribution same as baseline metadata. At the index level, number of leaf pages or blocks is scaled by $\alpha$.

## 7.2 Time Scaling

### 7.2.1 Problem Statement

Given a baseline metadata $\mathcal{M}$, query workload $\mathcal{Q}$ and a user specified scaling factor $\alpha$, produce a scaled metadata instance $\mathcal{M}^\alpha$ such that the total optimizer's estimated cost (time) of executing $\mathcal{Q}$ on the scaled version is $\alpha$ times the total optimizer's estimated cost of executing $\mathcal{Q}$ on the baseline metadata instance.

### 7.2.2 Approach

Scaled metadata instance $\mathcal{M}^\alpha$ can be constructed if we have the individual scaling factors of relations participating in the query workload $\mathcal{Q}$. To obtain scaling factors for individual relations, we solve the following optimization problem:

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

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 baseline and scaled databases, respectively. We obtain $c^O_{q_i}$ from the estimated execution plan of $q_i$ in baseline metadata instance. Computing $c^S_{q_i}$ is harder as it has to be modelled as a function of individual scaling factors of relations participating in $\mathcal{Q}$. However, we make the following assumptions for scaled metadata instance to make the modelling of $c^S_{q_i}$ possible.

- Metadata scaling is implemented such that relative frequency distribution on non key columns remains same as baseline metadata instance and for key columns domain (number of distinct values) is scaled.
- Execution plan tree for all the queries $q_i$ in $\mathcal{Q}$ remains same as baseline metadata instance.
- The cost of each operator in query plan tree can be written as a simple function of input data sizes (cardinalities).

**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 7.1: Scaled output size and distribution

Given these assumptions, the output size of each operator is determined by Lemma 1 (Proof

**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 7.2: Query costs in scaled database

| 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 7.1: Simple cost function of plan operators

is given in Appendix B) in Figure 7.1 and the steps to compute the cost function, $c_{q_i}^S$, for each query $q_i$ in $\mathcal{Q}$, is given in Figure 7.2. As mentioned in the assumptions, we use the simple cost model provided in Table 7.1 to compute cost function. Our cost model considers only the CPU processing required for the operator. CPU processing cost of an operator is modelled as a function of input cardinalities. Our scaling implementation assumption makes the average tuple width of scaled database to be the same as the original database and thus makes the

per tuple cost function to be same for both original and scaled database. Since per tuple cost function for each operator is same for both original and scaled database, our cost model works well and we have observed it for TPC-H query workloads.

Given the steps in calculating the cost function, we can now compute the individual scaling factors by solving the optimization problem as described in Figure 7.3. Our assumption on the implementation of metadata scaling introduces an additional constraint as described in Lemma 2. It bounds the scaling factor of relation in specific cases. The proof of Lemma 2 is presented in Appendix B. When multiple solutions are available, we choose the result which is closest to a traditional size-based scaling (Step 4 in Figure 7.3), since it is our expectation that this would be more robust with regard to (a) addition of new queries to the workload, and (b) retention of the same plans across the scaled databases.

### 7.2.3 Implementation Details

SuanShu [14] Java optimization library is used to solve the minimization problem. The library takes initial point as an argument. In order to search the solution from multiple initial points, we defined a $k$-dimensional cube space, where dimension $i$ represents the scaling factor for relation $R_i$. We considered values 1 and (10 times of $\alpha$, where $\alpha$ is the desired time scaling factor) as the end points in each dimension of the cube. The corner points of the cube and a vector with traditional size-based scaling factor as multiple initial points to the optimization library. The solution is reported to the user with their objective function values. Metadata space scaling is done on the relations with user chosen solution vector to complete the time scaling.

### 7.2.4 Example

Consider a 1GB TPC-H workload consisting of queries Q1, Q14, Q17 that operate on relations PART and LINEITEM with probabilities (0.5, 0.48, 0.02) and scaling factor of 2 ($\alpha$). The desired scaling factors of PART and LINEITEM are assumed to be $\alpha_p$ and $\alpha_l$. As a first step (Figure 7.3) to solve the cost scaling problem, we have to determine the cost of queries in scaled database

**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 $\alpha_i$ 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 $\alpha$, 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 $\alpha$

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

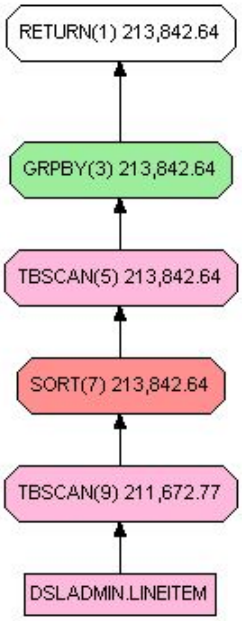
$$\sum_{\alpha_i \in s} (\alpha - \alpha_i)^2$$

**5.** Scale the input relations with the scaling factors obtained in step 4 to get the required cost scaled metadata $\mathcal{M}^\alpha$.

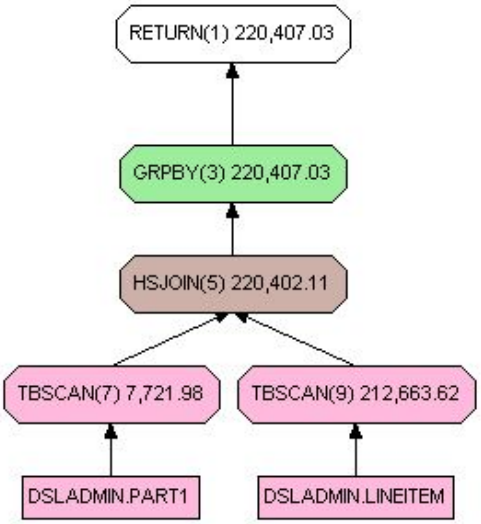
Figure 7.3: Time scaling of metadata

as in the procedure (Figure 7.2). The execution plans for the queries are obtained from DB2 optimizer and given in the Figures 7.4(a), 7.4(b), 7.4(c).

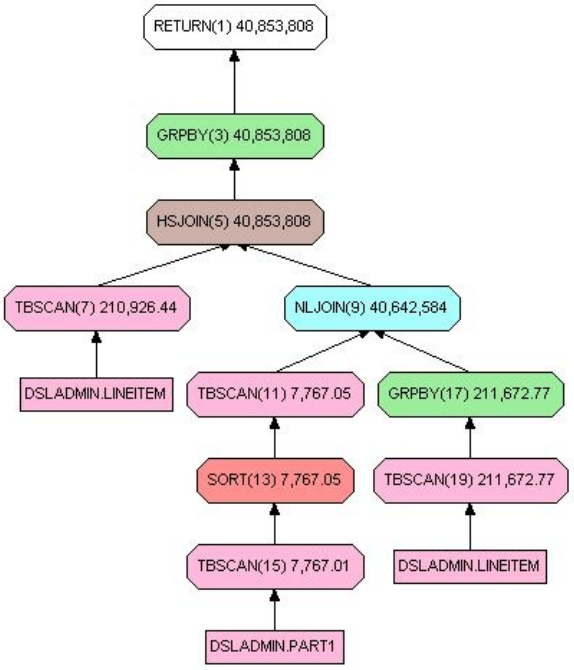
**Cost Calculation for Q14:** We illustrate here the steps 2 to 4 of Procedure (Figure 7.2) to determine the cost of operator HSJOIN(5) in Figure 7.4(b). In a similar fashion, cost of other operators can be calculated.




(a) Query 1




(b) Query 14




(c) Query 17

Figure 7.4: Plan trees for TPC-H Queries Q1, Q14 and Q17

**Cost of HSJOIN(5):**

- Determine the CPU cost (in millions of CPU instructions [**M CPUInsts**]) of the operator from the execution plan tree and let it be $CPU\_Cost$.
  ***E.g.:*** $CPU\_Cost$*( HSJOIN(5) ) = 59* **M CPUInsts**
- Determine inputs $x$, $y$ to the operator and find the cost of the operator using our simple cost model in Table 7.1 (for Hash Join, cost is $x$ + $y$) and let the cost of operator be $SCM\_OpCost$.
  ***E.g.:*** $SCM\_OpCost$*( HSJOIN(5) ) = 200000 + 70390 = 270390* **M CPUInsts** *i.e The inputs to the operator HSJOIN(5) are the cardinality of TBSCAN operators [TBSCAN(7) outputs entire* PART *relation and TBSCAN(9) outputs 70389.875 tuples after filtering* LINEITEM*].*
- Now, the DB2 cost function of this operator is obtained as
  $CostFn = (CPU\_Cost/SCM\_OpCost)$.
  ***E.g.:*** $CostFn$*( HSJOIN(5) ) = 59 / 270390* **M CPUInsts**
- We use the Lemma 1 to determine the scaled outputs of each input operator.
  *Scaled output of TBSCAN(7) =* $\alpha_p$ ** 200000* **tuples***, TBSCAN(9) =* $\alpha_l$ ** 70390* **tuples**
- Determine the cost of operator for scaled relations and let it $Scaled\_SCM\_OpCost$.
  ***E.g.:*** $Scaled\_SCM\_OpCost$*( HSJOIN(5) ) =* $\alpha_p$ ** 200000 +* $\alpha_l$ ** 70389.875* **M CPUInsts** *i.e scaled cardinality of operators TBSCAN(7) and TBSCAN(9).*
- Now, the operator cost for scaled relations is obtained as follows:
  $CostFn * Scaled\_SCM\_OpCost =$
  $CPU\_Cost * (Scaled\_SCM\_OpCost/SCM\_OpCost)$.
  ***E.g.:*** *Scaled Cost( HSJOIN(5) ) =* $CostFn * Scaled\_SCM\_OpCost$
  *= ( 59 / 270390) * (*$\alpha_p$ ** 200000 +* $\alpha_l$ ** 70390)* **M CPUInsts**

Similar to the cost calculation of operator HSJOIN(5) for scaled relations, we computed the cost (in **M CPUInsts**) of other operators of Query 14 execution plan tree (Figure 7.4(b)):

1. TBSCAN(7) : 391.54 * $\alpha_p$

2. TBSCAN(9) : 17836 * $\alpha_l$

3. GRPBY(3) : 18 * $\alpha_l$

The total cost function (sum of cost of all operators) for **Q14** is,

$$CostQ14(\alpha_p, \alpha_l) = 435 * \alpha_p + 17869 * \alpha_l \textbf{ M CPUInsts}$$

The original (before scaling) cost obtained from the estimated plan (Sum of CPU cost of all operators) is 18303 **M CPUInsts**.

In a similar fashion, cost function for queries Q1 and Q17 are calculated.

Cost function of **Q1**:

$$CostQ1(\alpha_l) = 14290 * \alpha_l + 1147 * \alpha_l * log(5899930 * \alpha_l) \textbf{ M CPUInsts}$$

The original (before scaling) cost is 22054 **M CPUInsts**.

Cost function of **Q17**:

$$CostQ17(\alpha_p, \alpha_l) = 2729002 * \alpha_p * \alpha_l + 552.65056 * \alpha_p$$

$$+25910 * \alpha_l + 0.064 * \alpha_p * log(192 * \alpha_p) \textbf{ M CPUInsts}$$

The original (before scaling) cost is 2756519 **M CPUInsts**.

Now minimizing the objective function

$$((CostQ1(\alpha_l)/22054) - 2)^2 + ((CostQ14(\alpha_p, \alpha_l)/18303) - 2)^2$$

$$+((CostQ17(\alpha_p, \alpha_l)/2756519) - 2)^2$$

on the constraints

$$CostQ1(\alpha_l) * 0.5 + CostQ14(\alpha_p, \alpha_l) * 0.48 + CostQ17(\alpha_p, \alpha_l) * 0.02$$

$$= 2 * (22054 * 0.5 + 18303 * 0.48 + 2756519 * 0.02) \; and$$

$$0 < \alpha_p, \alpha_l < \infty$$

The local minimum obtained is $(\alpha_p, \alpha_l)$ = (1, 2) (rounded to nearest integer). Relation LINEITEM is scaled by two times and the cost (**M CPUInsts**) of queries before scaling and after scaling is given in Table 7.2. The cost (**M CPUInsts**) of query workload is scaled by the scaling factor 1.99. Similar cost scaling for individual queries and query workload are achieved for the total cost (**timerons**).

| **Query / Cost** | **Before Cost Scaling**<br><br>**Total Time (M CPUInsts)** | **After Cost Scaling**<br><br>**Total Time (M CPUInsts)** | **Obtained Scaling** |
|---|---|---|---|
| Q1 | 22055 | 44108 | 1.99 |
| Q14 | 18303 | 36180 | 1.97 |
| Q17 | 2756519 | 5512481 | 1.99 |

Table 7.2: Cost of queries before and after scaling

# Chapter 8

# Conclusion

In this work, we have designed and implemented CODD software on a rich suite of popular database engines. CODD provides a unified visual interface wherein the user inputs metadata to construct various alternative scenarios within few minutes. Graphical histogram interface provided with CODD lets the user to modify the column data distributions visually. While allowing the user to play with arbitrary metadata values, CODD ensures that these values are legal and consistent with engine requirements. CODD metadata retention is helpful in environments where metadata is constructed from a data instance. It drops the data without affecting the metadata and gets back the space occupied by the data. In addition, CODD provides two other features such as metadata porting and scaling to help the database test teams.

The CODD software currently can be used only in testing of metadata based modules such as query optimizers. A very natural extension to CODD will be to construct alternative scenarios for testing execution modules. As mentioned earlier, CODD constructs “dataless databases”, where in only the metadata statistics are stored and the associated data is either removed or never created. However, execution module requires data in evaluating scenarios. One vital approach would be to create ”dynamic-data” environments using CODD by coupling it with the on-the-fly synthetic generator. 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. 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 for the level $i$ can be finished and the performance can 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.

# Appendix A

# Metadata Consistency Constraints

## A.1 DB2 Metadata Consistency Constraints

Table A.1 lists the metadata consistency constraints of DB2 Constraint Graph shown in Figure 4.4.

| 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). |
| | **Continued on next page...** |

**Continued from previous page...**

| Constraint | Description |
|---|---|
| 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 A.1: DB2 Metadata Consistency Constraints

## A.2 Oracle Metadata Consistency Constraints

Table A.2 lists the metadata consistency constraints of Oracle Constraint Graph shown in Figure 4.5.

| 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. |
| 10 | COLVALUE values must be unchanging or decreasing. |

Table A.2: Oracle Metadata Consistency Constraints

# Appendix B

# Lemma Proof

This section presents the proof for Lemma 1 and 2 used in time scaling (Chapter 7).

## 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 $\alpha_a$ 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. For example, in Figure 7.4(c), NLJOIN(9) is a level 0 join node and HSJOIN(5) is a level 1 join node.

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 $\alpha_a$, $\alpha_b$ 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,

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

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 $\alpha_a$, $\alpha_b$ 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):

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

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,

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

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

This proves that, the output cardinality is expressed as a function of not referenced relations (Among the join input relations $A$ and $B$ 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:

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

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,

$$\text{Scaled output cardinality}$$

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

= $s(\alpha_{a1}, ..) * s(\alpha_{b1}, ..) *$ 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 $\alpha_p$, $\alpha_s$ and $\alpha_{ps}$ are the scaling factors of relations PART, SUPPLIER and PARTSUPP respectively.

# Bibliography


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

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

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

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

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

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

[7] `download.oracle.com/docs/cd/B19306\_01/appdev.102/b14258/d\_stats.htm\#i1035422`

[8] `dsl.serc.iisc.ernet.in/projects/CODD`

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

[10] msdn.microsoft.com/en-us/library/ms162153

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

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

[13] www.jfree.org/jfreechart/

[14] www.numericalmethod.com/wiki/numericalmethod/wiki/SuanShu

[15] www.tpc.org/tpcds

[16] www.tpc.org/tpch