Constructing DWARF For Storing The Data Cube or a Selected View Subset

Emmanouil I. Vogiatzis

A Thesis presented for Graduate Diploma Degree

Software Technology And Network Applications Laboratory
SoftNet

Department of Electronics & Computer Engineering
Technical University of Crete

Committee:
Deligiannakis Antonios (Assistant Professor)
Christodoulakis Stavros (Professor)
Garofalakis Minos (Professor)

Chania, Greece

July, 2011

Abstract

Many practical applications, such as Data Warehousing, Market-basket analysis and Information Retrieval, rely on efficient methods of computing, storing and querying data cubes. Online-Analytical Processing has been a field of competing technologies for the past ten years. Recently, a very clever multi-dimensional index structure termed Dwarf was proposed. It is well known for achieving dramatic reduction in size and time is Dwarf [1] . The original work achieved a Petabyte 25-dimensional cube shrunk to a 2.3 GB Dwarf Cube, in less than 20 minutes, contriving a 1:400000 storage ratio.

In our thesis we propose an improved implementation of Dwarf build. We developed a Decision Support Algorithm, in order to evaluate a given set of views which can be supported by the Dwarf data cube index. This is mainly based on user's demands on which views will be requested from the cube. An efficient algorithm decides on the best permutation of the dimensions of a given fact table, and then, builds the dwarf cube efficiently. Our purpose is to be able to built only specific parts of the index cube, thus reduce size.

Also, we have re-implemented the Dwarf index from scratch and developed an improved memory-mapped disk version, maintaining most of the original specs, succeeding better size and time reduction, further more than the original Dwarf algorithm. We also aim to provide query optimization for processing Dwarf queries fast, as well as provide a mapper-utility for minimizing the size needed to store data while expanding the input data types, to achieve additional size reduction. Also, we apply an algorithm decision based on bitmaps of the dimensions needed for each view. Finally, we compare with the existing dwarf algorithm times and sizes.

---

[1] Sismanis Yannis, Deligiannakis Antonios, Roussopoulos Nick, and Kotidis Yannis. Dwarf: Shrinking the PetaCube. In *ACM SIGMOD*, 2002.

Acknowledgements

I would like to express my heartfelt gratitude and appreciation to my supervisor professor Dr. Antonios Deligiannakis for his invaluable suggestions, encouragement and moral support throughout this thesis work. His valuable patience, critical analyses and knowledge were influential for the completion of this work, and my future studies.

Also, I would like to extend my sincere thanks to Professors Dr. Minos Garofalakis and Dr. Stavros Christodoulakis, for their deliberation of this dissertation, their comments and guidance throughout my studies at Technical University of Crete.

Dedicated to my parents and Maria…

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data cube has been playing an essential role in improving the efficiency of fast Online Analytical Processing (OLAP) [6] operations over the past ten years [11]. Data warehouses are collections of historical, summarized, non-volatile data, which are accumulated from transactional databases. Since many practical applications, such as Data Warehousing, Information Retrieval and Market-basket analysis use online analytical processing, the idea of pre-computing and storing aggregates has always been necessary. Besides large data warehouse applications, there are other kinds of applications like bioinformatics, survey-based statistical analysis, and text processing that need the OLAP-styled data analysis. Such decision support systems, business intelligence and data mining, acquire very fast query response on queries that try to discover trends of patterns in the data set. OLAP queries typically touch considerable portions of the data. Also, size reduction has been obsolete, while an enormous amount of data might not be important for the majority of the users of such systems.

Decision Support Systems, are mainly classified into inputs (factors, numbers, and characteristics to analyze), user knowledge and expertise (inputs requiring manual analysis by the user), outputs (transformed data from which DSS "decisions" are generated and decisions (results generated by the DSS based on user criteria). In common, a dilemma has plagued DSSs since they were invented. On one hand, we know that if the data build index pre-computes everything then users do not have to worry about response times, as

they will get the best possible information no matter the nature of the data. But on the other hand, by pre-computing everything, we waste space, and especially in large cubes, huge parts of the calculated cube will or may never be queried. This leads to create user-defined stored functions can be used to encapsulate some of business logic and enhance the index performance.

Therefore, data cube construction has been on focus of much research and many algorithms have been proposed such as BUC [7], H-cubing [8], Star-cubing [9], multi-way array aggregation [10], CURE [20] etc. Materialization of a data cube consists of pre-computing and storing multi-dimensional aggregates so that analysis on any levels of the cube can be performed on the fly. Therefore, the data cube operator encapsulates all possible groupings for datasets.

A significant fraction of this work has been on ROLAP techniques, which are based on relational technology to store and manage warehouse data. This approach implements multidimensional schemas directly on top of an existing relational DBMS; however, it extends DBMSs by appropriate indexing techniques to speed-up query processing. Important extensions were bitmap-indexes, which were pioneered by Model 204 [12], Sybase IQ [13, 14] and Oracle. Existing Relational OLAP (ROLAP) and Cubing solutions mainly focus on "flat" datasets, which do not include hierarchies in their dimensions. Analytical tools need very fast query response on queries that try to discover patterns in the data set. Materialized views extension such as the CUBE BY operator [15], which is a multidimensional extension of the standard GROUP BY operator, computes all possible combinations of the grouping attributes.

Difficulties with the cube operators are size, both for computing and storing. A fact table is a table in the data warehouse that contains facts consisting of numerical performance measures, and foreign keys that tie the fact data to the dimension tables. Dimension attributes define the constraints of the Data Warehouse. A fact table can often very large and sparse and

dimensions are often split into hierarchical branches because of the hierarchical nature of organizations. In such cases, the size of a group-by is possibly close to the size of the fact table. So the size of a data cube increases exponentially after computation.

1.1 Problem description

As expected, hierarchies have many complicated matters to solve such as:

- The number of nodes in a cube lattice which increase dramatically
- The number of unique values in the higher levels of a dimension hierarchy may be very limited, while partitioning data into fragments that fit in memory and include all entries of a particular value may often be impossible
- The number of tuples that need to be materialized in the fact cube which increase dramatically

Computation of data cubes, though valuable for low-dimensional databases may not be so beneficial for high-dimensional ones. Typically, a high-dimensional data cube requires massive memory and storage space. Existing algorithms are unable to materialize the full cube under such conditions. Since a data cube grows exponentially with the number of dimensions, it becomes too costly in both computation time and storage space to materialize a fully high-dimensional data cube.

Dwarf is a highly compressed tree-like structure for computing, storing, and querying data cubes. Identifies prefix and suffix structural redundancies and factors them out by coalescing their store. Also, provides a 100% precision

on cube queries and is a self-sufficient structure, which requires no access to the fact table. A simple pass over the tuples is enough, and as mentioned in [15] where authors proposed mapping string dimension types to integers for restoring the storage, it keeps storage consumptions in extremely low levels considering the size of the fact table. The algorithms are designed to use bounded amounts of memory and processing.

Suffix redundancy has the dominant factor in sparse cubes. Its elimination has the highest outcome both in storage and computation time. Reducing the computation of a number of sub-dwarfs during data cube construction would improve performance and storage consumption.

Existing dwarf algorithm had an option to pre-compute only aggregates whose computation would be enough costly to be done on the fly, using the minimum granularity metric. The authors of the original dwarf structure propose a G_{min} parameter in order to eliminate the creation of sub-dwarfs on dimensions that contain a number of tuples less than this parameter.

But, a dataset often contains information that some users may never query. In sparse cubes, dwarf size can be quite larger than the fact table. It would be essential if data cubes were built based on the demands of the users.

Therefore, eliminating computation of sub-dwarfs on such areas could be beneficial for:

- Memory requirements
- Construction times
- Cube size
- Query performance
- Update times

1.2 Proposed Solution

This thesis utilizes the original Dwarf as a basic structure of pre-computing and storing data. We developed the idea of ***notSafe*** and ***Safe*** dimensions view computation. *NotSafe* views are chosen based on a list of given views by the user. A hybrid algorithm operates before the executions of dwarf build algorithms. Each *View* consists of different dimensions over the fact table, in any order. This thesis proposes some rules defining which dimensions are *safe* to compute suffix coalescing. Before the beginning of the built, an additional algorithm reorders the dimensions and enables the elimination of any "undesirable" sub-dwarf tree computation, which leads to decreasing the size and time needed for construction, far more than the original dwarf.

As known, dwarf data cube model calculates and stores every possible aggregation with 100% precision. A recent work in [2], the authors present a surprising analytical result proving that the size of dwarf cubes grows polynomial with the dimensionality of the data set. This polynomial complexity reformulates the context of cube management, and has already redefined most of the problems associated with data-warehousing and OLAP as a full data cube at 100% precision is not inherently cursed by high dimensionality.

Considering the above, an improvement of dwarf was a motivation for us. As long as a user might not query every dimension of the fact table, then the idea of eliminating computations and constructions on some dimensions of the plan, would improve the performance. Given a set of views to materialize, we create DWARF representation that can store these views efficiently. Dwarf is selected due to huge storage/computation savings against other ROLAP, MOLAP models. Dwarf has been constructed for materializing ALL views of

data cube, and again, this research has not been conducted before for materializing subset of Views in Multidimensional OLAP.

1.3 Contributions of Present Work

The contributions of this work are summarized below:

1. A new version of dwarf algorithm implementation, based on memory mapping buffered I/O with fast construction times.
2. A decision algorithm for evaluating, sorting, and reordering Dwarf cube construction order.
3. A generic TupleReader interface, to read input from text and sql-based source input.
4. A mapper that generates unique integer ID's for any input data type enabling the option for storing more data types than integers.
5. A tool for answering Roll-up, Drill-down or point queries on the dwarf cubes that are generated, with fast response times

1.4 Thesis Outline

In this thesis, we focus on work related to Dwarf indexes, and extend the functionality for constructing index cubes based on specific view subsets.

Background knowledge and related research are discussed in Chapter 2. We describe the main aspects of the original dwarf algorithm and a quick description of the two basic algorithms, whose authors have proposed in [1].

In Chapter 3, the idea and work of this thesis are described. More specifically, the safe view algorithm decision rules and algorithms are

proposed, suggested and analyzed. We denote some major aspects of coalescing limitations of current algorithm, and how they can be avoided in order to achieve a reduced index cube.

In Chapter 4, we describe some of the major aspects of the current implementation work, which lead to this dissertation. As current work started from a memory-based version built, and passed off to a memory-mapped disk version, we describe some of the main implementation ideas and coding methods related to this work.

In Chapter 5, our experimental work is presented, both for the original dwarf as well as for the dwarf constructed for specific views. We attempt to compare the results with a more recent work presented in 2008 [18], while maintaining the experimental methodology of [1]. Thus, we scale both dimensions as well as the number of tuples, concluding in excellent runtime and size results.

In Chapter 6, query performance of the dwarf index cube is examined, following a methodology which suggest of creating two workloads, one including only unrelated queries, and the other, constituted by roll-up or drill-down queries.

In Chapter 7, conclusions and future work are proposed, as well as a general outcome of this thesis work.

Chapter 2

Related Work Analysis

Dwarf index was first presented in SIGMOD 2002 and also patented. Below we present the main aspects of the algorithm, in order to discuss in-depth some key points, which lead our research to focus on implementing dwarf based on view subsets.

2.1 An introduction to original DWARF algorithm

A data cube is a conceptual n-dimensional array of values. It aids in fast analysis and manipulation of data contained into dimensions. In the context of this thesis, each dimension of the cube corresponds to an attribute of the data in question and each point along the dimension's axis is a value of the corresponding attribute. In this section we formalize the redundancies found in the structure of the cube and explain how they are used in the dwarf algorithm to achieve a high compression ratio. To explain, we will use Table 1 as an example throughout this thesis, where we have 4 dimensions `Store`, `Customer`, `Product` and a measure `Price`.

Store	Customer	Product	Price
S1	C1	P3	$45
S1	C2	P4	$10
S2	C3	P3	$70
S2	C3	P4	$35

Table 1: Fact Table for cube Sales

2.2 Prefix and Suffix Redundancies

Prefix redundancy is the first to be identified by the algorithm, and can be easily understood why it is used to build indexes over the structure of the cube. It occurs if a value of a dimension (for example S1 from `Store`) appears in more than one group-bys. For example, S1 appears in the following: ⟨S1⟩, ⟨S1, C2⟩, ⟨S1, C2, P4⟩, ⟨S1, C1⟩, ⟨S1, C1, P3⟩, ⟨S1, P4⟩, ⟨S1, P3⟩a total of 7 times. Those group-bys share the prefix of the first dimension. Any prefix of a cube will appear in $2^{(d-n)}$ group-bys (d is the number of cube's dimensions and n is the length of the prefix) [21] and possibly many times in each group-by. Dwarf stores the corresponding values only once, while avoiding replicating the same values over the views. This is prefix reduction. In general, this can be extended on more prefixes, and reduces the amount of storage required for these tuples.

Suffix redundancy occurs when a common suffix if shared between two or more group-bys. An example is described below: We have three dimensions ⟨a, b, c⟩. The suffix ⟨b, c⟩ is shared between views ⟨a, b, c⟩and ⟨b, c⟩. In [1], suffix redundancy is defined when a set of tuples of the fact table contributes the exact same aggregates to different groupings. For example, we have ⟨C1, P3 $45⟩ and ⟨S1, C1, P3 $45⟩. In general, a value of a dimension b, called b_j, will appear in the fact table as ⟨bj , x⟩. Also, for a single value from dimension a, called a_i, the group-bys ⟨ai , bj , x⟩ and ⟨bj , x⟩ will have common values. This happens whenever two or more partitions share common participating dimensions and cover the same subset of fact table tuples. During construction, suffix redundancy is identified and only one copy of the sub-Dwarfs will be constructed, thus eliminating space.

The original dwarf has no need of any previous information about the value distribution. It takes as input a sorted data set, and then automatically identifies prefix and suffix redundancies.

2.3 A Dwarf Example

First of all we will describe an example of Dwarf structure using data from table 1 and the aggregate function SUM. Then we briefly define the properties of Dwarf algorithm.

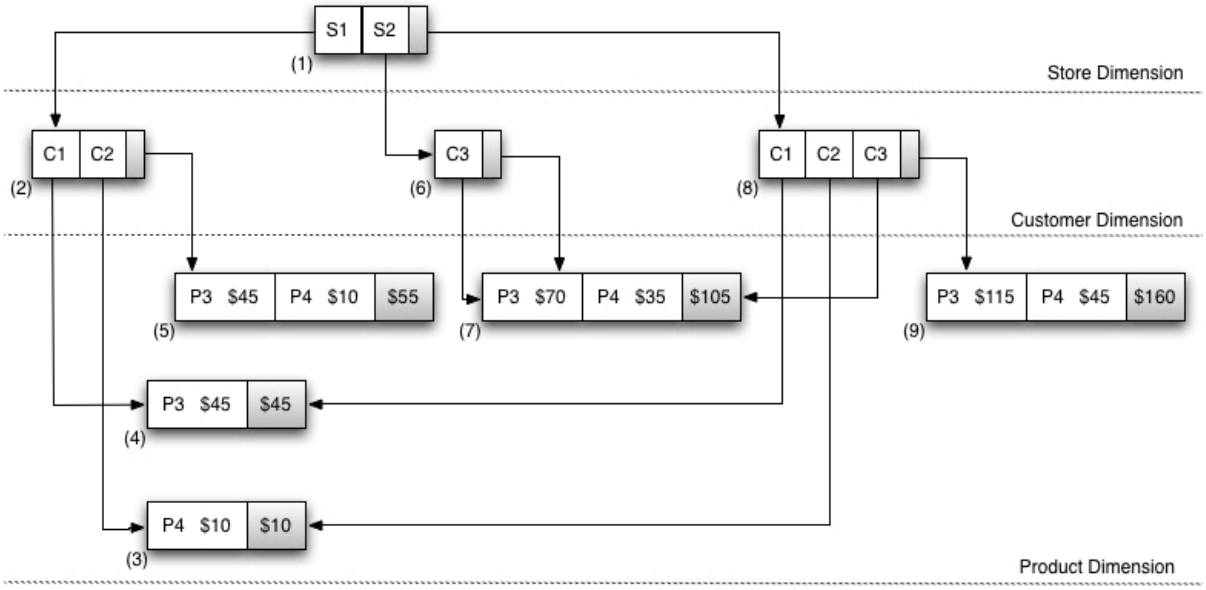


Figure 1: The Dwarf Cube for Table 1

To describe the construction of Dwarf, we decided to build the above cube based on the aggregate function *sum*. The numbers of nodes correspond to the creation order. The height of Dwarf is obviously equal to the number of dimensions of the given fact table. Each node can be either a *LeafNode* or a *NonLeafNode* node. The root node contains cells of the form [key, pointer], one for each distinct value of the first dimension. The pointer of each cell, points to the node below containing all the distinct values of the next dimension that are associated with the cell's key. LeafNodes are nodes that are found at the last dimension of the fact table and contain the aggregate value [key, AggValue]. Each cell designates all those nodes that are determined by the cell itself. For example, cell *C3* of *Customer* Dimension designates the node containing keys

P3, P4 of *Product* Dimension. While *LeafNodes* contain aggregate values, the *NonLeafNodes* contain an extra cell called, *ALLCell*. This is described as a gray small area at the end of each *NonLeafNode*. This is an extra pointer, which contains all the aggregate values of each node.

Following the path from the root to each leaf node is practically a single tuple of the fact table, which corresponds to one or more aggregate values. For example $\langle S2, C3, P3\rangle$ corresponds to the value \$70. Also, we may follow the path $\langle S2, C3, ALL\rangle$ which corresponds to the LeafNode containing $\langle P3, \$70\rangle$ and $\langle P4, \$35\rangle$. Those values are the sum of the Prices actually paid by Customer C3. In addition, the node created on step 7, contains both nodes ⟨P3, \$70⟩ and ⟨P4, \$35⟩, as well as the special ALL cell holding the price \$105. In general, leafNodes are of the form [key, aggValue], and if this node contains more than one key, the new aggValue is summed with the existing values in this node.

If we follow the path ⟨ALL, ALL, ALL⟩ we observe that we reach a leafNode, which contains all the products P3, P4 that are contained in our fact table. In addition, we have calculated the total number of Prices. This is also described as a query of the form (group-by NONE). Following paths ⟨S1, C1, P3⟩ , ⟨S1, ALL, P3⟩ and ⟨S1, C1, ALL⟩ , which consist of second raw of the table, lead to the cell [P4, \$10]. These coalescing redundancies are identified by dwarf, which when published for the first time, completely changed the perception of a data cube from an unordered collection of distinct groupings, into a complex network of interleaved groupings and aggregates, that eliminate both *prefix* and *suffix* redundancies.

2.3.1 Properties of Dwarf

We briefly introduce the main Dwarf properties, as described by the authors in [1]:

1. It is a directed acyclic graph (DAG) with just one root node and exactly D levels, where D is the number of cube's dimensions.
2. Nodes at the *D*-th level (*leafNodes*) contain cells of the form: [*key*, *aggregateValues*].
3. Nodes in levels other than D-th level (*nonLeafNodes)* contain cells of the form: [*key*, *pointer*]. A cell *C* in non-leaf node of level *i* points to a node at level *i* + 1, which it *dominates*. The *dominated* node then has the node of *C* as its *parent* node.
4. Each node also contains a *special cell*, which corresponds to the cell with the pseudo-value ALL as its key. This cell contains either a pointer to a non-leaf node or to the aggregateValues of a leaf node.
5. Cells belonging to nodes at level *i* of the structure contain keys that are values of the cube's *i*-th dimension. No two cells within the same node contain the same key value.
6. Each cell C_i at the *i*-th level of the structure, corresponds to the sequence S_i of *i* keys found in a path from the root to the cell's key. This sequence corresponds to a group-by with (*D* - *i*) dimensions unspecified. All group-bys having sequence S_i as their prefix, will correspond to cells that are descendants of C_i in the Dwarf structure. For all these group-bys, their common prefix will be stored exactly once in the structure.
7. When two or more nodes (either leaf or non-leaf) generate identical nodes and cells to the structure, their storage is coalesced, and only one copy of them is stored. In such a case, the coalesced node will be reachable through more than one path from the root, all of which will share a common suffix. For example, $\langle \mathrm{S1}, \mathrm{C2}, \mathrm{P4} \rangle$ and $\langle \mathrm{ALL}, \mathrm{C2}, \mathrm{P4} \rangle$ share the common suffix $\langle \mathrm{C2}, \mathrm{P4} \rangle$. Any nodes that are dominated by a coalesced node N will also be coalesced, since they can be reached from multiple paths from the root to the leaf nodes.

2.3.2 Constructing Dwarf cubes

Two main processes can describe the original Dwarf algorithm: The prefix expansion and the suffix coalescing. An important condition before the commencement of the construction is that the tuples are first using a fixed dimension order, and then they are inserted into the Dwarf structure one by one. In our previous example (Figure 1) the Store dimension is sorted. Another interesting point to emphasize is the fact that both algorithms do only a single scan over the fact table, while building the index, calculating and storing every possible aggregation with 100% precision.

The prefix expansion sequentially scans over the sorted fact table. Every time a new tuple is read, all the necessary nodes and cells are created. When a new tuple is read, the algorithm compares each dimension with the previous tuple, and when it finds common prefix, it stores the new key values (Figure 2). As we reach the last dimension, a common prefix P between *curTuple* and *prevTuple* is computed. For the first tuple of Table 1, we create $|P| + 1$ number of nodes. For D-dimensional cube, new $D - |P| - 1$ nodes are required. When a *leafNode* is closed, the "ALL*" cell is created by aggregating the contents of the other cells in the node. An aggregation cell maps to a measure. When a *nonleaf* node is closed, the "ALL" cell is created and call the suffix coalescing algorithm presented in Figure 2 to create the sub-Dwarf dominated by this cell. For a *nonLeaf* node at level L < D an aggregation cell points to a node at level L +1. Therefore, the content of a cell belonging to a node N, contains either *aggregateValues* (*leafNodes*) or sub-dwarfs (*nonLeafNodes*). The suffix compression turns the tree into a DAG. For instance, if we follow one of the paths ⟨S1, C1, P3⟩ , ⟨S1, ALL, P3⟩ , ⟨S1, C1, ALL⟩ or ⟨ALL, C1, P3⟩ we will end up at the same cell of node (4).

Algorithm 1 CreateDwarfCube Algorithm

Input: sorted FactTable *F*, D: MaxDimension
Output: DwarfIndex *dwarf*
1: //iterate over sorted fact table:
2: Create all nodes and cells for the first tuple
3: last_tuple ← first tuple of the fact tuple
4: **while** more tuples exist unprocessed **do**
5: current_tuple ← extract next tuple from sorted fact table
6: P ← common prefix of current_tuple, last_tuple
7: if new closed nodes exist **then**
8: write special cell for the leaf node homeNode where last_tuple was stored
9: For the rest $D - |P| - 2$ new closed nodes, starting from homeNode's parent node and moving bottom-up, create their ALL cells and call the SuffixCoalesce Algorithm
10: **end if**
11: Create necessary nodes and cells for current_tuple { $D - |P| - 1$ new nodes created }
12: last_tuple ← current_tuple
13: **end while**
14: write special cell for the leaf node homeNode where last_tuple was stored
15: For the other open nodes, starting from homeNode's parent node and moving bottom-up, create their ALL cells and call the SuffixCoalesce Algorithm (Algorithm 2)

Figure 2: CreateDwarfCube Algorithm

Algorithm 2 SuffixCoalesce Algorithm

Input: inputDwarfs = set of Dwarfs
1: **if** only one Dwarf inputDwarfs then
2: return Dwarf in inputDwarfs
3: **end if**
4: **while** unprocessed cells exist in the top nodes of inputDwarfs **do**
5: find unprocessed cells exist in the top nodes of inputDwarfs;
6: toMerge ← set of Cells of top nodes of inputDwarfs having keys with values equal to Key_{min} ;
7: **if** already in the last level of structure **then**
8: curAggr ← calculateAggregate (toMerge.aggregateValues)
9: write cell [Key_{min},currAggr]
10: **else**
11: write cell [Key_{min},suffixCoalesce(toMerge.sub-dwarfs)]
12: **end if**
13: **end while**
14: create the ALL cell for this node either by aggregation or by calling SuffixCoalesce, with the sub-Dwarfs of the node's normal cells as input;
15: return position in disk where resulting dwarf starts

Figure 3: SuffixCoalesce Algorithm

2.3.3 CreateDwarfCube algorithm execution

During the construction of the first algorithm, we decided to create a Mapping Utility, in order to save space, and extend the input data types. When a new tuple is read, the common prefix P is computed, while for each new value of the tuple, we assign a single Integer to it. This is also described by using the most compact representation of a unique dimension ID in the form of an integer. This, of course, simplifies implementation (no different data types in nodes) and speed (native data type faster to move around and compare). Therefore, for each new value of a tuple, we hold String to Dimension ID's representation of the values, and dwarf is constructed based on those assigned integers. For each new unassigned value, we add into the mapper list each new Integer ID. (Move it at efficient cube storage).

The above algorithm (Figure 2) is described by holding dimension level on every loop of the process. We decided that there is no need to keep dimension level information except in the outermost loop of the process. When a new tuple is processed, we search for common prefix between current and previous tuple, and therefore, the information that we finally need about Dimension level is only those dimensions where we do not have common prefixes.

2.3.4 Suffix Coalescing algorithm execution

Suffix Coalescing requires as input, a set of Dwarfs and merges them to construct the resulting Dwarf, called sub-Dwarf. For the root node of the resulting Dwarf, the sub-Dwarf of the cell with the value k is constructed by merging all those sub-Dwarfs of the cells in the top nodes of the input Dwarfs with value *k*, and the sub-Dwarf of the "ALL∗" cell is constructed by merging the sub-Dwarfs of other cells within the same node.

If the size of nodes that consist a list of nodes *toMerge* is 1, then it is obvious that the product of the coalescing will result to the Dwarf itself. If the

size of nodes *toMerge* list is greater than 1, continuous recursion of suffix Coalesce happens, and for all the keys that are into the *cellMap* (a list of leaf nodes that are about *toMerge*), we call suffixCoalesce recursively. This continues until we reach the last level of the fact table, where we finally call the aggregate function to SUM all the aggregate values of the leaf-Nodes that are dominated by the sub-dwarf, and produce the final $\{ALL *\}$ cell. Of course, our data cube supports more than one aggregate value on each tuple for example:

Store	Customer	Product	Price1	Price2	Price3
S1	C2	P4	$10	$20	$15
S1	C1	P3	$45	$35	$5
S2	C3	P3	$70	#80	$75
S2	C3	P4	$35	$90	$30

Table 2: Example of more aggregate Values

2.3.5 Memory Requirements

As analyzed in [16] it is easy to understand that dwarf algorithm has no major memory requirements. CreateDwarf Cube algorithm requires remembering only the previous read tuple, to compare with current tuple from the fact table. Suffix Coalesce is designed in order to keep in memory a list of which nodes are about to merge from each level down to the next. This also holds the path from the root to the leaf nodes. A memory-mapping version of the algorithm permits a quick look up procedure of a dimension's ID for its indexes into the nodes. It is evident that $MaxMemoryNeeded = c * \sum_{i=1}^{D} Card_i$, where *c* is the size of the cell, and *D* are the levels of the structure from root to the $\{ALL *\}$ Cell of the root node.

Chapter 3

Current Work

3.1 Clustering Dwarf Cubes

In 4.2 [1] the authors describe the idea of clustering the structure. The original algorithm has 100% precision for storing all the possible group-by's, which may be queried. We use the lattice representation [3] of the Data cube to demonstrate the computational dependencies between the group-by's of the cube, using a binary representation.

View	Binary Representation	Parents
abc	000	
ab	001	abc
ac	010	abc
a	011	ab, ac
bc	100	abc
b	101	ab, bc
c	110	ac, bc
none	111	a, b, c

Table 3: View ordering example

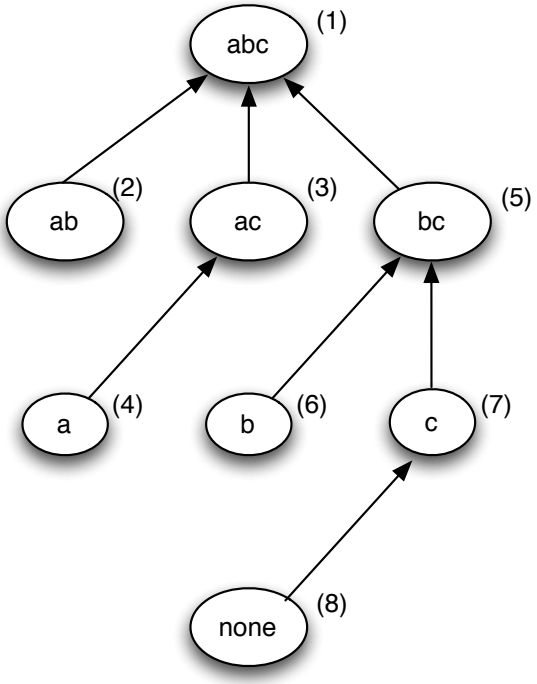


Figure 4: Processing Tree of Table 3

In the above lattice, each node corresponds to a group-by. Node ⟨bc⟩ represents the *bc* group-by view. It depends on dimensions *b* and *c.* Also, the node ⟨abc⟩ consists of the dimensions a, b and c. Using directed edges, the authors point out the computational dependencies among all the possible group-by's. Such dependencies occur when a group-by can be computed from another group-by as *ac*, which is computed by dimension *a*. The group-by abc, therefore, can be used to compute any group-by in the lattice.

3.1.1 Introducing the idea of View bitmap

The second column of Table 3, consists of the bitmap of each possible view. Each dimension is represented by "0" or "1" (False/True), and the total bits required are equal to the number of dimensions of a given fact table. "0" is assigned when a dimension is participating, whereas 1» is used when a dimension does not appear in the view. A simple example is the view *bc*, which is represented by "100", as we have 3 dimensions, or view *ac* which is represented by "010". In general, for every dimension of a view v_i we have:

$$\forall\, d_i \in D\ \exists\, b_i : b_i = \begin{cases} 0 & ,if\ i\ exists \\ 1 & ,if\ i\ not\ exists \end{cases}$$

A view given by user includes one or more dimensions of the fact table. A bit value (False) indicates the participation of the dimension into the view. For a total number of 4 dimensions, we create a list of all the possible bitmaps, starting from the 0000 bitmap which represents the view ⟨A, B, C, D⟩. For a total of X dimensions, all the possible views are 2^X, so we construct a total number of 16 bitmaps. This binary representation gives us the possibility to recognize which view can be computed from which, and therefore, eliminate suffix coalescing computation and construction, by utilizing the fact that a view may be computed from another earlier during dwarf coalescing.

For example, we introduce the following paradigm, which consists of 4 dimensions, using the ancestor v_i with the biggest common prefix for w ⟨A,B,C,D⟩ and a total of 16 bitmaps:

View	Binary Representation	Parents
ABCD	0000	
ABC	0001	ABCD
ABD	0010	ABCD
AB	0011	ABC, ABD
ACD	0100	ABCD
AC	0101	ABC, ACD
AD	0110	ABD, ACD
A	0111	AB, AC, AD
BCD	1000	ABCD
BC	1001	ABC, BCD
BD	1010	ABC, BCD
B	1011	AB, BC, BD
CD	1100	ACD, BCD
C	1101	AC, BC, CD
D	1110	AD, BD, CD
none	1111	A, B, C, D

Table 4: Bitmap representation with 4 dimensions

Based on the above example, we note that the binary representation for view v_i can be derived from a binary representation of w, changing any bit from true[1] to false[0]. Parents of a view v_i are denoted by w, and for each view, there exists one or more parents, depending on number of true[1] bits. When a bitmap contains more than one true[1] bit, then the number of candidate parents is equal to the number of those bits. For example, view $w = {}_{AB}[0011]$ contains two true[1] bits. As a consequence, anticipated parent views are $v_1 = {}_{ABC}[0001]$ and $v_2 = {}_{ABD}[0010]$, with $\{v_{1_{(BCD)}}\} < w_{(BCD)}$.

Each view ancestor occurs earlier as seen using Binary-Coded Decimal encoding. For the view $w = {}_{AB}[0011]$, both $v_1 = {}_{ABC}[0001]$ and $v_2 = {}_{ABD}[0010]$ exist, with v_1 having been computed earlier in comparison to v_2. Another example is view $w' = {}_{B}[1011]$, with prospective parent views $v_3 = {}_{AB}[0011]$, $v_4 = {}_{BC}[0011]$ and $v_5 = {}_{BD}[1010]$. The computation order of those views is $v_5 \rightarrow v_4 \rightarrow v_3$. The idea of choosing the most recent view as a parent, is based on the assumption that the most recent computed view, has more possibilities to be in memory, and caching the nodes will be less time consuming.

View	Binary Representation	Parents
ABCD	0000	
ABC	0001	**ABCD**
ABD	0010	**ABCD**
AB	0011	ABC, **ABD**
ACD	0100	**ABCD**
AC	0101	ABC, **ACD**
AD	0110	ABD, **ACD**
A	0111	AB, AC, **AD**
BCD	1000	**ABCD**
BC	1001	ABC, **BCD**
BD	1010	ABC, **BCD**
B	1011	AB, BC, **BD**
CD	1100	ACD, **BCD**
C	1101	AC, BC, **CD**
D	1110	AD, BD, **CD**
none	1111	A, B, C, **D**

Table 5: Ancestors selection based on most recent views are bolded

3.1.2 Optimizing view iteration

In OLAP cubes, multidimensional structure is defined as a variation of the relational model that uses multidimensional structures to organize data and express the relationships between data [7]. Each cell within a multidimensional structure contains aggregated data related to elements along each of its dimensions. According to the algorithm for constructing the Dwarf cube, certain views may span large areas of the disk. It became obvious that users tend to query only certain dimensions (based on their views bias) and not the entire cube, especially over a fact table that contains a considerable number of dimensions. An efficient algorithm for constructing dwarf cube based on subset of views would cause the elimination of any sub-dwarfs during the suffix coalescing of the nodes. Also, it could lead not to compute some $\{ALL *\}$ nodes. Therefore, our goal is to achieve faster construction, and size reduction of the cube, by avoiding storing the whole database structure.

Optimizing view iteration refers to ordering the views so that they are created in order to maximize locality, and optimize query performance. The cost of answering a query Q, is the number of rows present in the table for that query, which are coalesced to construct the cube. For example, computing $v_a = \{a, ALL *\}$ and then $v_b = \{ALL *,\ ALL *\}$, instead of doing the closure and coalescing in their natural order, could save some time in disk access during build.

Using View-to-Bitmap algorithm, we define for each possible view, the parents from which it can be computed. The general rule is that we choose as a parent, the view that is most recently computed based on binary representation in Table 5.

Algorithm 3 View-to-Bitmap Mapping Algorithm 3

```
Input: inputViews = set of Views {v1, v2, … , vk}
Dims: list of dimensions of the fact table
Output: set of bitMaps
1: while more views exist unprocessed into Input
2:       currentView ← extract next view from Input
2:       for( i = 1 to Dims.length ) do
3:              if( currentView.contains Dims[i] ) do
4:                     bitMap[i] ← 0
5:              else
6:                     bitMap[i] ← 1
7:       end if
8:       Output ← insert bitMap
9:   end while
10: sort Output list based on Binary representation
11: Call SafeView algorithm
```

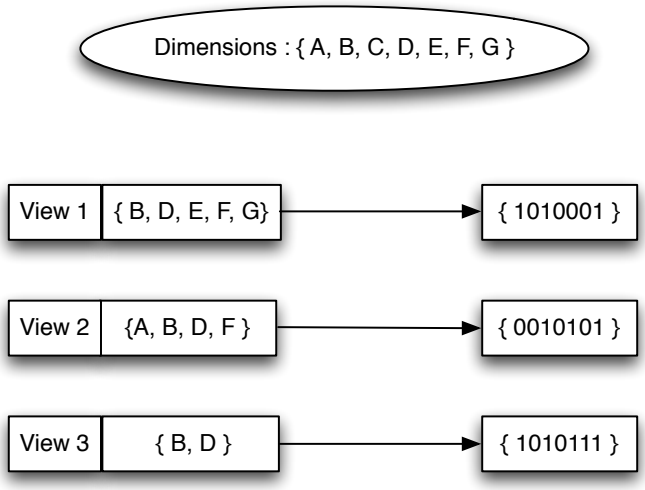
Figure 5: View to Bitmap Mapping Algorithm

Algorithm 3 describes how the bitmap mapper "transforms" views into binary representation of all views.

Each of these sorted views, probably contain one more common dimension. These dependencies suggest that some of the views are able to be computed from others, meaning that for a view $v_i = [x_1, x_2, \dots, x_k]$ and $v_j = [x_1, x_2, \dots, x_m]$ where $k \subseteq m$, then, if all of the dimensions of v_i are included in v_j, it is **safe** to compute v_i from v_j. For example, consider view $v_1 = {}_{CG}[0010001]_2$, $v_2 = {}_{BCEG}[0110011]_2$, $v_3 = {}_{EG}[0000011]_2$, and a fact table with dimensions <A,B,C,D,E,F,G>. Then, view v_2 and v_3 can be computed from v_1, if v_1 has already been stored into the build index. Views v_2 and v_3 are not related. Sorting these views from higher to lower binary representation, we have$v_2 = {}_{BCEG}[0110011]_2 > v_1 = {}_{CG}[0010001]_2 > v_3 = {}_{EG}[0000011]_2$

Therefore, computing only v_2, suggests that all the other given views can be safely answered, as the ALL$*$ nodes that will be queried have already been computed by the suffix coalescing.

What happens when a view can be computed from two or more other views? For example, consider the following views:



After sorting the above bitmaps, we have:

$\left|v_3 = {}_{BD}[1010111]_2\right| > \left|v_1 = {}_{BDEFG}[1010001]_2\right| > \left|v_2 = {}_{ABDF}[0010101]_2\right|$.

View v_3, can be computed equally from {view 1} and/or from {view 2}. But v_2 cannot be computed from v_1 because it contains dimension <A> which does not exist in v_1, alike v_1 cannot be computed from v_2 because it contains dimensions <E,G> which do not exist in v_2. Adopting the rule of the most recent parent, our algorithm will choose as a parent the view that "logically" would have been most recently computed in sorting order, (as in the example of table 5) which is v_1. The idea of most recent parent is adopted, as it is more possible that any sub-graphs required for constructing each view, will probably have already been computed/stored by having materialized a view whose bitmap is closest to us. Thus, v_3 can be safely computed from v_1. Or, in Table 5, we can chose as parent view for {a}, one of {ab}, {ac}, {bc}. But, according to previous assumptions, it is more probable that if we chose to be computed from {ad}, the data required for view {a} will be found during the construction of this view.

3.1.3 Coalescing problem and solution

In this thesis, we introduce the idea of *safe* and *notSafe* views. The authors of dwarf had a criteria deciding that if the algorithm had to coalesce things that were quite large, then, dwarf exited coalescing on those nodes, and executed a partial sorting on the fact table. The idea of being *safe* refers to the fact that all requested data which any possible query might request, will be available at the time a suffix coalescing has to be computed. This, of course, is recursive, denoting that if a view is computed from another which is safe, then all those data referring to those views, is guaranteed that will be found (safely) while traversing the tree.

A *view* is a set of dimensions, which belong to the whole data set of a given fact table. Supposing we have a data set of N dimensions, one view V_1 includes a set of x dimensions, where $x \leq N$. Another view V_2 may include a set of y dimensions, not identical to x. If both V_1 and V_2 contain the same dimensions, they are identical. Generally, each view corresponds to answering queries on specific dimensions, and supporting group-by's on those levels. Our idea is based on utilizing the existing dwarf cube to support each view, and also maintain the option of point queries.

We propose a decision algorithm based on the dimensions each view contains. Our idea is to construct dwarf index cube able to support each given view before the build. We maintain the plan of the cube, which means that users are able to execute point queries over the fact table, and Roll-up or Drill-down queries will be supported only on dimensions, which are defined by the views.

What was interesting was to manipulate how each dimension that does not appear in a view, could be bypassed in cases that we would have to coalesce

things, or create sub-dwarfs, or avoid computing the "ALL∗" cell on each level. This should also support the idea of being *safe*.

As we know, dwarf of a node X that is dominated by some cell of N is called sub-dwarf of N. Each sub-dwarf contains nodes of lower levels, as well as their aggregated "ALL∗" nodes, which support the group-by's aggregations. Indeed, there are cases when, during coalescing of a dimension, the construction algorithm might need to follow a path of an "ALL∗" node in order to compute a sub-dwarf of a given dimension, which might not have been computed.

One of the main problems, which occur during coalescing, is the fact that if we decide not to store all possible views, then, there are great possibilities that we may end up in sub-tree areas where the desired data might be missing. To be more specific, during prefix coalescing on higher levels of the tree, we merge nodes, which correspond to some group-by's. Each node has a prefix in the tree, which is the path to be followed until the node is reached, from the root of dwarf.

If we decide to follow a path from the root to the tree, and a specific coalescing has been dismissed/not created, then, the area where the index will try to read, may point to *null*, and therefore, not able to answer correct a query. Thus, to be computed from some other view V_i, then on each of respective levels, there must exist the aggregate cell for every sub-tree.

For example, in figure 6, the original Dwarf [1], would build the "ALL∗" on dimension 2, [C1, C2, C3]. This, of course, as the dimensions grow, has great effect on size cost. Node [C1, C2, C3] is accessed by the "ALL∗" node of root, leading to data from group-by's which can not be defined on their first dimension, or again, not having {∗} in first dimension.

To support a view V_i with prefix P_i whose prefix contains 1^{st} dimension but not 2^{nd}, (bitmap: [01…]), then both [C2, C3] and [C2] would have to build their {∗} during suffix coalescing. If none of the other views in the safe list has

defined dimensions [11…] over P_i, then, we only need to follow the path from the "ALL∗" of dimension 1 → "ALL∗" of dimension 2, to reach dimension 3.

Let's assume an output permutation from algorithm 3, with plan <A,B,C,D,E,F,G>, which produces maximum number safe views. Also, one of them is view v_1 = <D, F>. In such case we are willing to override intermediate dimension {E} (of course none of the other views has prefix on {E}), only keeping the paths of the nodes which point from {D} to the next {E} level, and of course from {E} to {F}. The bitmap of v_1 = <D, F> is [111010].

During construction, we traverse the path from the root ALL∗ {A} → ALL∗ {B} → ALL∗ {C} → ALL∗ {D}, then create coalescing over the nodes of {E}→ ALL∗ {F}, and finally, as we have reached leaf nodes on last level, coalescing over the nodes of {G}.

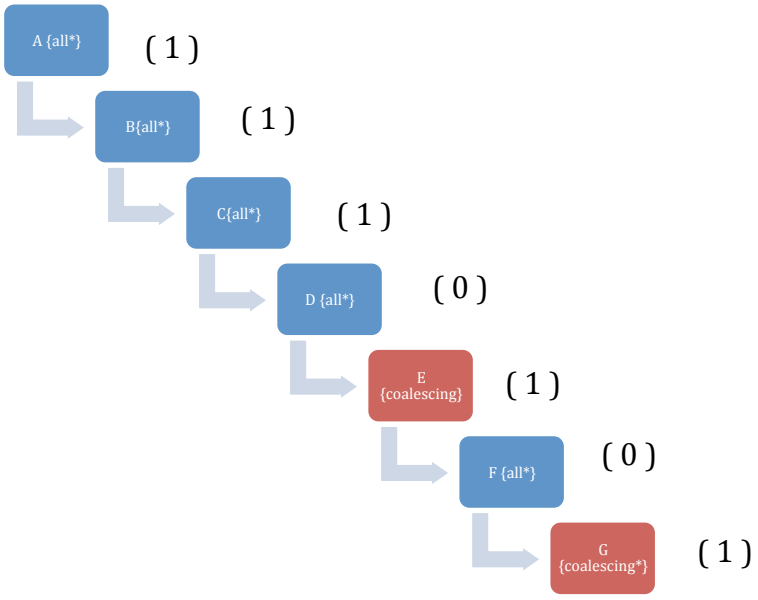


Figure 6: Traverse of a path for [1111010]

It can be understood more clearly in figure 7, and in particular when coalescing on level 3 occurs. As we can see on this simple dwarf cube, D4 appears twice on level 3. Thus, D4 points to F4, F5 and F3, and possibly more sub-dwarfs on a more complex cube.

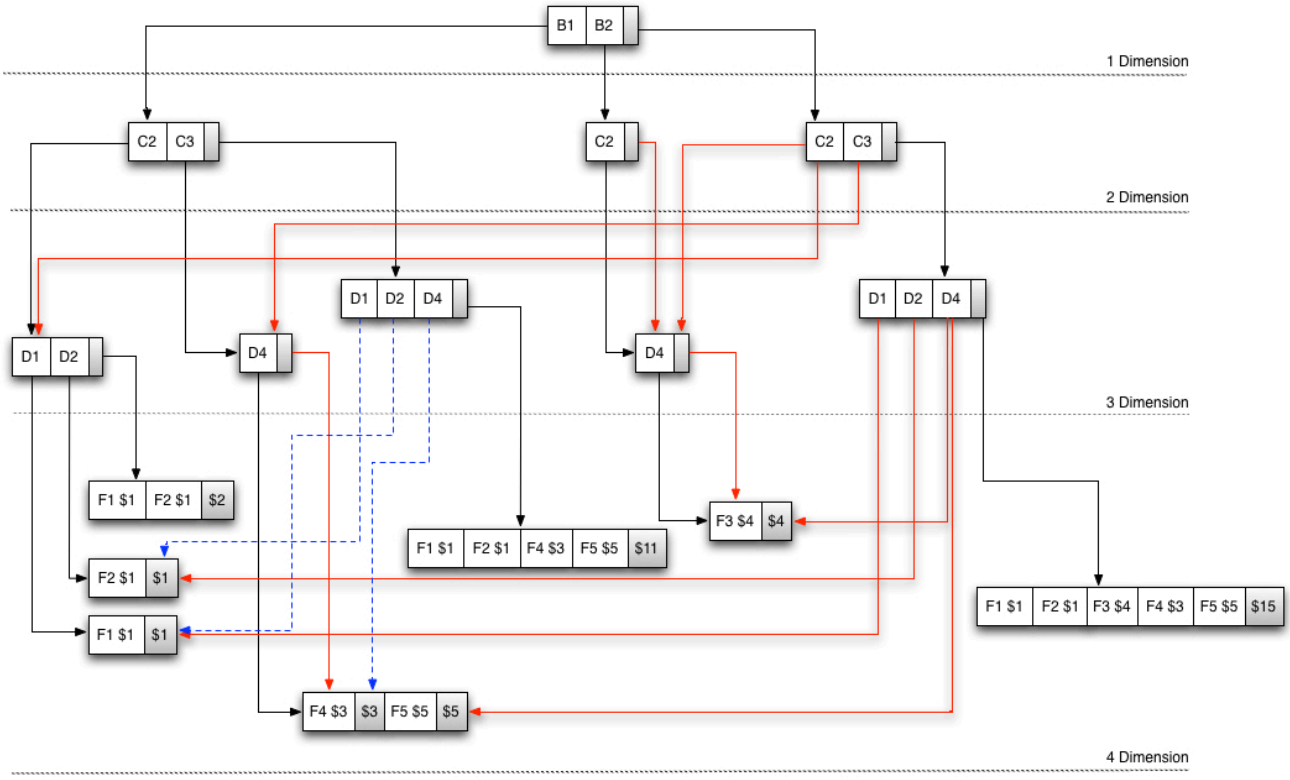


Figure 7: A more complex Dwarf example

During closure of D4 node, it coalesced F4 and F5, which could be sub-dwarfs and not simple leaf nodes. This possible coalescing into an intermediate dimension of our view cannot be overridden easily.

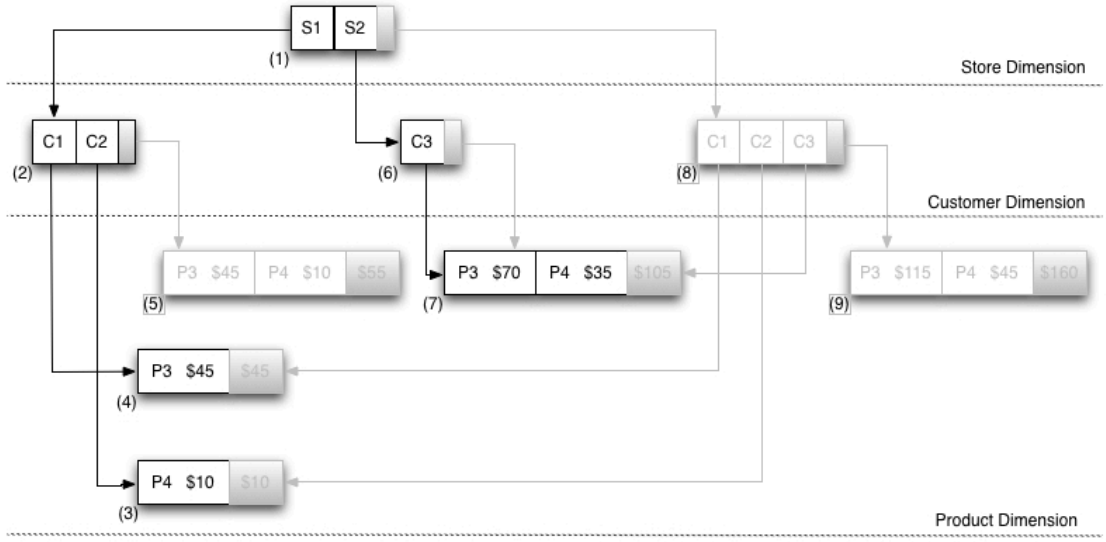
In such case, a solution to override {E}, would be to create a node below {E} to accommodate all the nodes of {F} dimensions or point straight to the fact table. In addition, it would require creating a copy of the sub-tree and compute the ALL∗ nodes on-the-fly if requested (for example the ALL∗ nodes of {F}).

Computing on the fly the {ALL∗} nodes of a dimension when requested, suggests that we will have to compute also the sub-dwarfs of the {ALL∗} of {F}.

This means that for the extra dimension between {D, F} for example, we coalesce, ensure that we have computed the {ALL∗} for that dimension, and if not, compute on the fly, and finally continue traversing down the path till we reach the leaf nodes.

Generally, we coalesce nodes and build their ALL cell, on dimensions where a given view will request a group-by on those nodes. If not, just compute the ALL cell containing every node on this dimension, in order to maintain the path of the tree to reach other nodes on lower levels.

Now, consider the following dwarf, which has been constructed to hold all the tuples of the fact table, for the view V = {a,b,c}.



If a query Q = {∗, b, ∗} appears, then, we need to traverse the path from Store dimension down to Customer Dimension, through the {ALL ∗} cell of (1) node. In our case, this is not possible, as we have decided to omit building the corresponding nodes. As a consequence, the query Q cannot be answered, and probably we will need to compute it on-the-fly, using lazy updates.

Again, we have to come up against problems such as:

1. Not having materialized subgraphs of Dwarf due to the fact of not having materialized some views.
2. Traversing the tree can lead to areas where "ALL∗" is not computed
3. Data or node pointers might be missing (null)
4. Problems in computation (cannot merge a subgraph that is not computed)

5. Queries (not able to find requested data, thus compute it on the fly using lazy updates).

3.2 Safe and Not Safe Views decision algorithm

To confront the above problems, we needed a rule that guarantees that each view in the dwarf is computed **SAFELY**. This denotes that during coalescing, all sub-dwarfs needed are stored. The problem of storing specific views is much harder than it seems, because as mentioned before, coalescing can lead to "bad" areas (not fully computed) in many cases.

After conducting further research, we purpose the following rules for examining which views need to be materialized and which require a more complex process. Redundancies between views occur, as denoted on 3.1.1, when view v_1 contains dimensions, which are found also in a view v_2.

- Rule 1:

Suppose a view V_i containing a number of dimensions X_i and V_j containing Y_j number of dimensions.

If $\exists\, X_i \;:\; X_i = \{d_1,\ d_2\ , d_3, \dots d_i\}$ and

$\exists\, Y_j \;:\; Y_j = \{d_1, d_2, d_3, \dots d_j\}\ where\ |i+1| = |j|$

Then V_i is candidate to be *Safely* computed from V_j.

- Rule 2:

Having two views V_i and V_j :

$\exists\ V_i : V_i = \{d_{n_1}, d_{n_2}, \dots d_{n_i}\}$,

$\exists\ V_j : V_j = \{d_{m_1}, d_{m_2}, \dots d_{m_k}\}\ \ where\ \ |k| = |i+1|$

If the last not stated dimension D_{n_i} in V_i is stated in V_j and Rule (1) is valid, then V_i is safe to be computed from V_j.

For example:

(01110 → 01100)

(01101 → 01100)

For example, supposing that view V_1 = { ∗, 2,∗ ,4, 5 } is safe, then :

V_3 = { ∗, 2, ∗, ∗, 5 } is safe

V_4 = { ∗, 2, ∗, 4, ∗ } is safe

V_5 = { ∗, ∗, ∗, 4, 5 } is not safe (this is only computed from { ∗, ∗,3 ,4, 5 })

Another example:

Supposing V_1 = { ∗, ∗, 3, 4, 5 } is safe then:

V_2 = { ∗, ∗, 3, 4, ∗ } is safe

V_2 = { ∗, ∗, ∗, 4, 5 } is safe

V_2 = { ∗, ∗, 3, ∗, 5 } is safe

Supposing V_1 = { ∗, 2, ∗, ∗, 5 } is safe, then this view produces only one safe view V_1 = { ∗, 2, ∗, ∗, ∗ }.

For a given set of views, it is necessary to disintegrate which views are safe and which are not, and then build dwarf. Below, we analyze this process.

The process starts by transforming each view, into a binary representation, based on the plan of the fact table as stated in (3.1.2). Then, it begins by comparing each view from the highest binary representation towards the plan (lowest binary representation), each time by confronting two views. When a view is characterized as “safe” it is removed from the list. As we have started from the highest binary number, the lowest binary representation will definitely be the view of the plan; V_{plan} = {A, B, C, D, E, F, G…} is represented

by {0,0,0,0,0,0,0,…}. This enables the option to compare all possible views, while maintaining the idea of choosing most recent parent introduced in (3.1.1). Until all views are safe, the algorithm creates all possible permutations of the dimensions of the fact table.

Suppose we have the following views:

V_1 = [1,2,3,4,5,6,7,8]	V_2 = [3,5,7]	V_3 = [2,3,5,7,6]	V_4 = [2,3,5]	V_5 = [3,5,6,7]
V_6 = [2,3,5,6]	V_7 = [1,3,6]	V_8 = [2,3,5,6,7]	V_9 = [2,3,4,5,6,7]	V_{10} = [1,2,3,4,5,6,7]

Table 6: Example of 10 possible views

Our permutation generator, has decided that the above views are safe from permutation [8, 1, 4, 5, 3, 7, 6, 2]. The ordering of the views is based on their binary representation from lowest to highest bitmap as shown below:

PLAN (View 1)			View 10	View 7	View 9	View 8	View 5	View 2	View 6	View 4	View 3
8	0		1	1	1	1	1	1	1	1	1
1	0		0	0	1	1	1	1	1	1	1
4	0		0	1	0	1	1	1	1	1	1
5	0		0	1	0	0	0	0	0	0	0
3	0		0	0	0	0	0	0	0	0	0
7	0		0	1	0	0	0	0	1	1	1
6	0		0	0	0	0	0	1	0	1	1
2	0		0	1	0	0	1	1	0	0	1

Table 7 : Example of binary representation of Views for permutation [8, 1, 4, 5, 3, 7, 6, 2]

Finally, our algorithm starts comparing views starting from View2 as described below (Table 7), comparing from right to left until we reach the plan. If the view is not found to be safe, then the algorithm characterizes it as *notSafe*, removing it from the list.

```
current permutation plan OF DWARF IS : [8, 1, 4, 5, 3, 7, 6, 2]

Sorted bitMaps of Safe Views:
[1=0000000, 10=10000000, 9=11000000, 8=11100000, 5=111000001, 6=11100100,
4=11100111, 3=11100111]
```

The above procedure is repeated by changing the ordering of the plan. For each permutation we examine all the views, and if all are found to be safe, then we terminate the process, and start building dwarf based on the *"perfect"* permutation plan. If not, we examine safe views from all the remaining, create support-Dwarfs with less dimensions if a dimension does not appear in any of the remaining views, until all views are finally safe.

Algorithm 4 SafeView Algorithm 4

Input: inputViews (set of Binary representation {v_1, v_2, … , v_k})
S: Fact Table, **NS**: other views Except S
StarsToCompute: where to build {ALL*}
Permutations: 2^Dims
Output: Permutation, Safe Views, StarsToCompute

```
1:  check all Views in NS if are safe
2:  if( NS.size() == 0 ) then
3:      call CreateDwarf_for_SafeViews Algorithm for S, S.StarsToCompute
4:  end if
5:  while ( NS.size() > 0 ) do
6:      find best permutation X for all views in NS
7:      copy all safe views of X in S
8:      remove all safe views from NS
9:  end while
10: call CreateDwarf_for_SafeViews Algorithm for X, X.StarsToCompute
11: while ( NS.size() != <empty> ) do
12:     find best permutation Y for all views left in NS
13:     copy all safe views of Y in S
14:     remove all safe views from NS
15:     call CreateDwarf_for_SafeViews Algorithm for supportDwarf cube for
        permutation on Y, Y.StarsToCompute
16: end while
```

Figure 8: Safe Views Decision Algorithm

3.3 Permutation Generator

In our java implementation, the *PermutationGenerator* Java class systematically generates permutations. It relies on the fact that any given set with n elements can be placed in one-to-one correspondence along with the set {1,2,3, …, n}. We implemented the algorithm described in [4] to generate all possible permutations for a given plan. In order to accelerate the procedure, all those dimensions, which do not appear in any views, are placed at the prefix side of the first combination. This accelerates the procedure, as we are able to search for the best permutation only among the

dimensions that are required to produce a safe Dwarf. The total number of permutations for x dimensions is $x!$. Reducing the dimensions, speeds up the generator, as less combinations are required.

Of course, as the dimensions and the views grow, this procedure can take up to 77 seconds for 254 views on 8 dimensions. During the process, if we have not found any perfect permutation to minimize the number of *notSafe* Views, then we keep in memory the permutation which produces a minimum number of *notSafe* Views. After we have finished building the first dwarf, we start a new search, this time only on the dimensions contained into *notSafe* Views, and not the whole plan.

At query time, when any roll-up or drill-down query needs to be executed, the query utility handles each request by searching on either the main Dwarf cube, or the *supportDwarfs* created.

3.4 Efficient reducing of sub-Dwarfs

In 3.2 we described the difficulties caused by interaction of the different dimensions appearing into a specific view, keeping the initial ordering. Using the permutations generator, the rules, the list of useful dims and, of course, the list of Safe Views, we start constructing dwarf.

- All useful dimensions are moved in front of the plan, and new sorting of the fact table is executed based on the first dimension of the new permutation-plan.
- NonLeafNodes contain an extra [cell] to store new aggregateValues
- For every node that is about to be closed, we examine if we have to compute its "ALL∗" value. This is decided based on the following: During construction of DWARF, each node has a *prefix*, which is actually the path

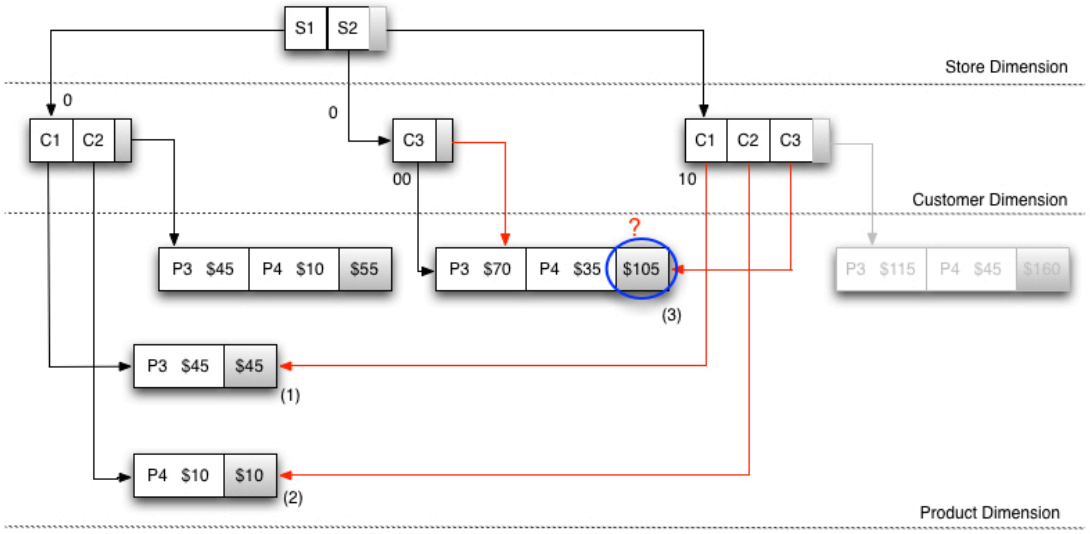
from the root to this node. If the path contains traversed dimensions and not "ALL∗" cells, then our prefix has "0" on each level, instead it contains "1". At building runtime of the "ALL∗", we compare the current path, which we have followed, with all the views, which are about to be built. If a view has the same prefix with us, and contains "1" at the same level where we are about to build the "ALL∗" node, then we build it. Otherwise, the building algorithm skates over this node.

- To avoid during query times, traversing the tree down to the leafNodes in order to fetch the requested aggregate values (of the current nodes which are about *toMerge* (Algorithm 2 *SuffixCoalesce*)), we store on higher levels the aggregate values of a sub-dwarf, depending on the list of safe Views. If not required, we write *null*.
- Algorithm 5 and 6 (*Updated_CreateDwarfCube* Algorithm) are executed (modified for safe Views), using an extra input array of useful Dimensions. While *Updated_CreateDwarfCube* compares each tuple with the previous one, coalescing occurs only on useful dimensions. Again, the builder identifies dimensions, and decides for every level, which nodes should compute their "ALL∗" cell, or write *null*. This depends on the fact that for every safe view, there are often dimensions, which are never queried.
- If a view has stated a dimension, for that level we follow the path to the nodes below (level+1), and compute their "ALL∗", and if required, compute the *newAggr* (line 12: algorithm 7). Thus, we are able to answer to specific group-by's on that dimension.
- If for none of the views, a dimension is not stated, we maintain the path by computing the final "ALL∗" on a level (bottom-up computation (line 9: algorithm 6) in order to keep the pointers of the nodes and maintain the path of the total dwarf index tree.

In general, we compute "ALL∗" cell of a node *iff* there exists a view to be stored which

- Starts by the same prefix, and
- Contains "ALL∗" in the dimension of the node

For example, let's assume that we have constructed the following DWARF for supporting the views {000},{010}, {011}, {100}.



When we have to compute node (3), we are about to build a node with prefix {01}, and need to compare this path with the given views. As there exists no other view containing the prefix {011…} we decide not to build the node (3), and for the same reasons, we do not build nodes (1) and (2).

With the above procedure, common dimensions are only recognized, and safe Views are used to reduce the number of sub-dwarfs stored. In addition, we preserve the path of the root to the nodes, overriding computation and storing of some "ALL∗" cells. Moving all the useless dimensions at the backend of the plan, gives us the advantage of avoiding traversing the whole tree index.

As mentioned above, coalescing may lead to undesirable conditions in arbitrary cases. Based on our rules, we will never construct nodes containing

"ALL∗" cell, but no other regular cells (with their corresponding subdwarfs). But may not completely rule out cases of having not calculated "ALL∗" cells due to coalescing multiple levels above. Thus, we cannot simply compute "ALL∗" cells based on the worst-case assumptions of coalescing, as size of Dwarf would explode and lead to storing the entire data cube, and again, if a query hits a dwarf region where an "ALL∗" cell is not computing, then we may compute the data missing on the fly.

Also, storing extra aggregates on nonLeafNodes cells, reduces the cost of traversing the tree down to the leafNodes, saves time and space compared to the original suffixCoalescing procedure. Suffix Coalesce is a major cause for latency, and, recognizing as early as possible which dimensions are not safe, accelerates the procedure. Suffix redundancy now is enabled only when required. Below we demonstrate the above procedures (Algorithm 5, 6, 7).

Algorithm 5 CreateDwarf for Safe Views Algorithm 5

Input: Permutation = order of Dimensions, S = ordered SafeViews,
stars = StarsToCompute
Output: finalPlan, notSafeDims, SafeViews
1: NotSafeDims ← Compute useless Dims from SafeViews and Permutation
2: Move all NotSafeDims.elements() at the end of Permutation
3: call Updated CreateDwarfCube Algorithm

Figure 9: CreateDwarf for Safe Views Algorithm 5

Algorithm 6 Updated CreateDwarfCube Algorithm 6

Input: SortedFactTable = finalPlan,
NSF = notSafeDims,
SafeViews = Safe Views from Algorithm 4
Output: DwarfIndex *dwarf*
1: sort Fact Table on first non-useless Dim of SortedFactTable
2: //iterate over sorted Fact Table based on SortedFactTable
3: last_tuple ← first tuple of the SortedFactTable
4: **while** more tuples exist unprocessed **do**
5: current_tuple ← extract next tuple from SortedFactTable
6: P ← common prefix of current_tuple, last_tuple only
on Dimensions not included in NSF.array()
7: **if** new closed nodes exist **then**
8: write special cell for the leaf node homeNode where last_tuple was stored
9: For the rest D - |P| - 2 new closed nodes starting from homeNode's parent node and moving bottom-up,
10: create their ALL cells and call SuffixCoalesce_New Algorithm
11: **end if**
12: Create necessary nodes and cells for current_tuple
{D - |P| - 1 new nodes created}
13: last_tuple ← current_tuple
14: **end while**
15: write special cell for the leaf node homeNode where last_tuple was stored
16: For the other open nodes, starting from homeNode's parent node and moving bottom-up, create their ALL cells and call the SuffixCoalesce_New Algorithm (Algorithm 7)

Figure 10: Updated CreateDwarfCube Algorithm 6

Algorithm 7 SuffixCoalesce_New Algorithm

Input: inputDwarfs = set of Dwarfs, NSF = safe Views

```
1: if only one Dwarf inputDwarfs then
2:       return Dwarf in inputDwarfs
3: end if
4: while unprocessed cells exist in the top nodes of inputDwarfs do
5:       find unprocessed cells exist in the top nodes of inputDwarfs;
6:       toMerge ← set of Cells of top nodes of inputDwarfs having keys with values
         equal to Key_min ;
7:       if NSF.contains(currentLevel +1) then
8:             if already in the last level of structure then
9:             curAggr ← calculateAggregate(toMerge.aggregateValues)
10:            write cell [Key_min, curAggr ]
11:            else
12:            newAggr← calculateAggregate for corresponging leafNode aggValues
               by traversing the path down to the last level, and copy their values
13:            write cell [Key_min, suffixCoalesce_New(toMerge.sub-dwarfs),
               newAggr]
14:            end if
15:      else
16:            write cell[Key_min, suffixCoalesce_New(toMerge.sub-dwarfs), null]
17:      end if
18: end while
14: create the ALL cell for this node either by aggregation or by calling
    SuffixCoalesce_New, with the sub-Dwarfs of the node's normal cells as input;
15: return position in disk where resulting dwarf starts
```

Figure 11: SuffixCoalesce New Algorithm 7

Chapter 4

Efficient Cube Storage

4.1 Input Mapper Utility

The data cube operator described in [15] performs the computation of one or more aggregate functions, for all possible combinations of grouping attributes. The data source may contain strings or other scalar data types (dates, lengths of time, days of the week, GPS coordinates, etc). The authors of [15] also provided some useful hints for cube computations including the use of mapping string dimension types to integers for reducing the storage. In our implementation we used this idea, creating a *TupleReader* generating integer ID's to map each different dimension attributes (strings). A string-to-integer mapping class creates an extra mapping file containing all the mappings for each one generated dwarf cube, stored together with the cube. This simplifies implementation, as it does not store different data types in nodes, and build and query speed times are improved (native data type faster to move around and compare). Later on, when we query the cubes, the mappings are read and used to translate strings to unique IDs. There is a warm-up time each time a cube is opened to read those maps and keep them in memory for fast translation. Each query has to look up in those maps prior to exploring the cube to look up the result. The experiments showed a dramatic improvement on both built and query times, with 1000 queries to be answered in under 1 second on hot cache experiments, even with 25 dimensions of dwarf cube index.

This extends the input possibilities, as it is not depending on what type of input a user decides to use. New applications include an increasing number of dimensions, and the explosion on the size of the cube is a real problem. We

decided to use double as aggregate data values. Float would halve the size of the data portion of the nodes, but would also loose precision, which is important for a sufficient number of OLAP application tools.

4.1 Memory-Mapped I/O

My search in dwarf algorithm started by constructing a memory-based version of Dwarf. After testing the performance, it soon became inherent the need to build a disk version of the cube as it was impossible to create cubes of large datasets. Finally, my advisor suggested continuing building a memory-mapping version, in order to execute buffered writes to disk and improve performance. The original paper mentions a considerable amount of tuning in the implementation, and as I coded in java (Eclipse IDE), I decide to use NIO collection (new I/O) of Java programming language APIs, which offers features and libraries for intensive I/O operations. It was introduced with the J2SE 1.4 release of Java by Sun Microsystems to complement the existing standard I/O. The APIs of NIO were designed to provide features such as:

- Buffers for data of primitive types,
- Channels, a new primitive I/O abstraction,
- A File interface that supports locks and memory mapping of files up to `INTEGER.Max_Value` Bytes (2 GiB).

The String-Integer mapper works entirely in memory with hash tables, and it was not the source of any performance difference. The main focus of my work was on the *FileManager* class. It provides support for the Dwarf cube file format. The format is a series of chunks that are memory mapped. Each chunk is as close in size as possible to `Integer.MAX_VALUE-4`. To avoid nodes crossing maps boundaries, sometimes it can be a bit less. The file

begins with the length of the chunk, followed by the chunk data, etc. At the beginning of the first chunk is the cube meta data, containing

- Location of root node
- Number of dimensions
- Number of Facts
- G_{min} factor, minimum number of leaf cells necessary to store the total.

Data is appended always at the end of the last byte buffer. When the last buffer capacity is going to be exhausted, a new one is added to the list, and the file offset is increased by the size of an Integer, to store the size of the next buffer. The next buffer always starts following this Integer, and the size of the buffer does not take into account this Integer. We use a *byteBuffer* to read the next chunk of the file into a pre-allocated *byteBuffer*, update the size of the last chunk and close the last chunk.

Also, we used bitwise operator (AND, XOR) and bit masking technique, in order to encode the headers of each node, containing:

1. node type: LeafNode / NonLeafNode
2. number of keys of each node
3. position index of each offset

Chapter 5

Experiments and performance analysis

5.1 Outline of Experiments

This thesis experimental work will examine three principal features of Dwarf indexes such as index size, time construction and query performance. Also, we will extend our experiments to study the performance of the Dwarf index, when built for specific Views as described in Chapter 3, randomly generated. We will repeat the most important experiments of [1] and also compare the results with a more recent research conducted in 2008 which is described in [17]. This is due to the fact that researchers in [17] used hardware configuration for their experiments similar to ours, also implemented in Java using memory-mapped I/O. Our attempt is to give the best understanding on Dwarf index scaling for certain scenarios, and of course, analyze our thesis work performance

5.2 Hardware Experimental Setup

We performed several experiments with different datasets and sizes to validate our storage and performance expectations. All tests in this section were run on a 2.4 GHz Intel Core 2 Duo processor running on 4 GB of RAM of a 2008 MacBook pro). We used a 500 GB SATA Hard disk rotating at 7200 rpms (Seagate Momentus), able to cache at about 16MB/sec, and a transfer rate of 3GB/sec. The operating system was Macintosh Snow Leopard 10.6.8. All code was implemented using java version 1.6. Still, our hardware is of

course better than the hardware available in 2002. In [1], authors used a 700 MHz Celeron processor running Linux 2.4.12 with 256 MB of RAM. The hard disk used was a 30GB disk rotating at 7200 rpms, able to write at about 8 MB/sec and read at about 12 MB/sec. The average disk seek time was not specified. Obviously, due to improved hardware we expect our runtime measurements for index construction and query response times to be always by a factor better than the results reported by the inventors in [1], conducted in 2002. Also, we compare our tests with a more recent research on [18] conducted in 2008. Researchers used a memory-mapped version of Dwarf, implemented in java, using improved hardware; two 2.4 GHz Dual Core AMD Opteron 280 processors, i.e., four cores in total, and 6 GB of main memory. Their operating system was Linux 2.6.9, and used a 300 GB ATA/133 hard disk with 16 MB Cache (Maxtor 6L300R0).

Other optimizations are:

1) Efficient I/O. As the I/O-classes of java.io are rather inefficient, we used the newer I/O-classes of java.NIO. These classes perform considerably better than the old java.io. In our implementation, non-leaf nodes contain integer offsets (using the mapper utility) pointing to nodes in the memory-mapped array. We map integer arrays than byte arrays.
2) Support Dwarfs. When the algorithm decides to build extra dwarfs to support all given views, we construct the supportDwarfs based only on given dimensions contained in the views. For each useless dimension, amount of size required has major impact on size and time reduction

5.3 Data Sets

Synthetic. We used synthetic data sets. Similarly to [1] we used data dimensions values follow either a uniform, a self-similar 80-20 [18], or a zipfian distribution over a given cardinality, and did not use any correlations among the dimensions. Random numbers were generated using the same generators as the authors in [1]. Authors in [18] mention using FastMersenneTwister for random numbers.

5.4 Index Construction Experiments

The following index construction experiments are based on scaling:

- number of Dimensions,
- number of Tuples.

5.4.1 Scaling Dimensions

In this section, our goal is to examine Dwarf index construction times, and index sizes using (1) synthetic datasets and (2) constructing Dwarf cubes for the whole plan. Our first set of experiments is based on scaling the number of dimensions from a range of 10 to 30. We used synthetic data of 100,000 tuples.

D	F. Table Size [MB]	Uniform						
		Size [MB]			Time [sec]			
		Dwarf[1]	Dwarf[18]	**NEW**	Dwarf[1]	Dwarf[18]	**NEW**	
							2core	1core
10	4.9	62	55	85	26	11	5.6	6.4
15	6.8	153	141	219	68	22	10.7	12.5
20	8.8	300	283	441	142	42	19	21.5
25	10.7	516	495	774	258	66	30	35
30	12.7	812	790	1236	424	102	46	50.6

Table 8: Storage and Creation time vs #Dimensions for Uniform

D	F. Table Size [MB]	80-20						
		Size [MB]			Time [sec]			
		Dwarf[1]	Dwarf[18]	**NEW**	Dwarf[1]	Dwarf[18]	**NEW**	
							2core	1core
10	4.6	115	152	156	46	26	7.2	8.5
15	6.4	366	1293	517	147	195	17.7	19.8
20	8.1	840	7779	1221	351	1123	38	41.3
25	9.9	1788	n/a	2390	866	n/a	85	91.3
30	11.7	3063	n/a	4146	1529	n/a	206	225

Table 9: Storage and Creation time vs #Dimensions for 80-20

D	F. Table Size [MB]	Zipf = 0.95						
		Size [MB]			Time [sec]			
		Dwarf[1]	Dwarf[18]	**NEW**	Dwarf[1]	Dwarf[18]	**NEW**	
							2core	1core
10	4	n/a	118	180	n/a	22	8.4	9.7
15	5.5	n/a	515	773	n/a	85	27.9	30
20	7	n/a	1570	2328	n/a	234	81	84
25	8.5	n/a	3851	5660	n/a	548	242	263
30	10	n/a	8272	11958	n/a	1153	680	776

Table 10: Storage and Creation time vs #Dimensions for Zipf = 0.95

In tables 8, 9, 10 are demonstrated repeats and complements of experiments shown in Table [4] from [1]. Cardinalities for all dimensions are equal to 1,000 and Fact table contained 100,000 tuples. The dimension values were either uniformly distributed, or 80-20 Self-Similar, of zipfian (θ = 0.95) distributions over the same cardinality. As noted in [18], researchers made the experiments using one core of the processor. We show that differences between using 1 or 2 cores in our implementation, does not provoke any stupendous time differences.

Comparing the above results, we observe that in construction times, there are big differences from [18] and [1]. We did not impose any correlation among the dimensions.

We observe the following:

- **Dwarf Index for Uniform Data**.

For a uniform distribution, our results have greatly improved constructions times against to [1] and [18]. Similar to [18] we used one core processor, and hardware was almost identical. Prefix redundancy elimination is obvious, having a Dwarf index always bigger than the fact table. For $D = 10$, our Dwarf takes 85 MB, while the fact table takes 4.9 MB, a 17:1 expansion ratio. For $D = 30$, Dwarf takes 1236 MB for a 12.7 MB fact table, resulting in an expansion ratio of 97:1. Uniform distribution posts the highest savings. For example for $D = 30$ we have 1236 MB, in comparison to Zipfian distribution where we have 11958 MB of dwarf index. In general, our implementation achieves 2x faster time construction against [18], and 9x against [1] at conditions of $D = 30$. From size comparison, for $D = 20$ or $D = 25$ we observe a factor of 1.5 larger size index than reported by [1] and [18].

- **Dwarf Index for Self-Similar Data.**

Using a 80-20 distribution, we have great savings when compared to [18]. For $D = 20$, in [18] is reported an index size of 7779 MB while our implementation outputs 1221 MB index size, obviously a factor of 6 compressed index. In [18] authors also mention a factor 3 to 9 larger size than the original dwarf, while our implementation maintains a factor 1.5 larger than reported both for all dimension scaling. From time savings, it is again our implementation a winner as for $D = 20$ we complete dwarf index built in

38 seconds, while [1] in 351 seconds (9:1), and [18] is unexpectedly slow reporting 1123 seconds (29:1). It is certain that suffix redundancy is clearly the dominant factor in the overall performance, and creation time is proportional to the Dwarf size. In addition, [18] suffers from a major disadvantage by not supporting any index size over 16 GB.

- **Dwarf Index for Zipf Data.**

Following [18] comparison tests, we generated datasets given of zipf factor theta $\theta = 0.95$. This was chosen by the authors [18] in order to generate a distribution which presents a significant skew while still having less skew than the self-similar 80-20 distribution. In table 4 of [1] authors do not imply any results, so we only compare only with [18]. Although we obtain faster construction times again, (for $D = 10$ we achieve construction time of 9.7 second while [18] post 180 seconds [20:1]), size elimination is more efficient in [18], as their savings in size is by a factor of 1.4 smaller.

In general, Dwarf produces smaller indexes for less skewed distributions. The denser areas benefit from prefix elimination, which is smaller, and sparser areas have less suffix redundancy to eliminate.

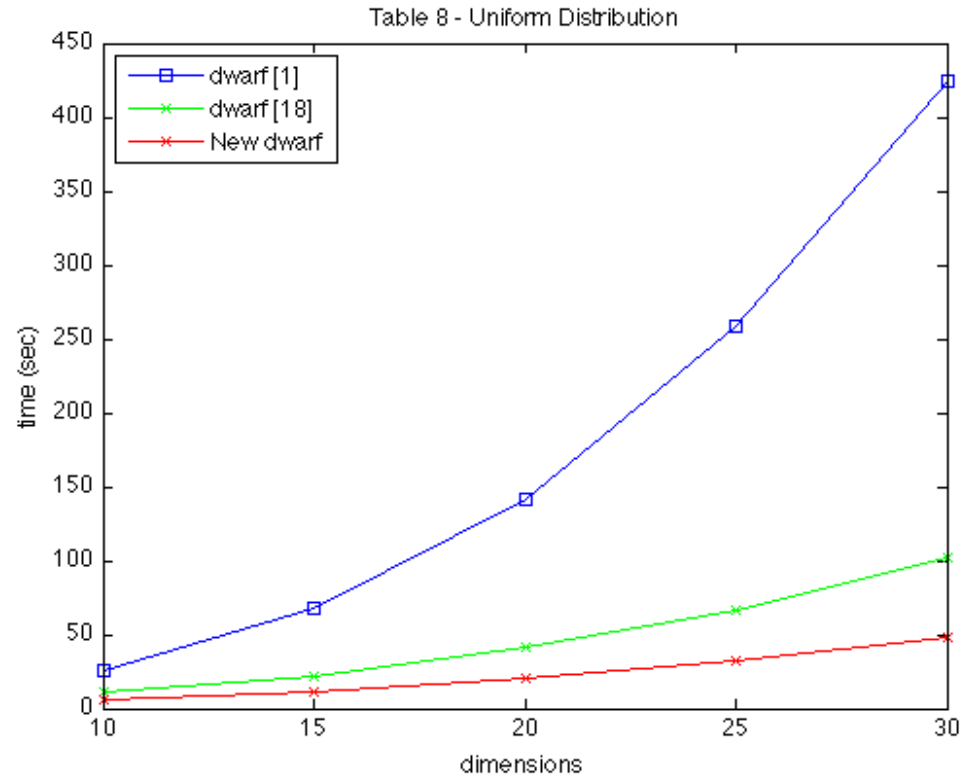


Figure 12: Results of Table 8 for #dimensions vs Time (Uniform)

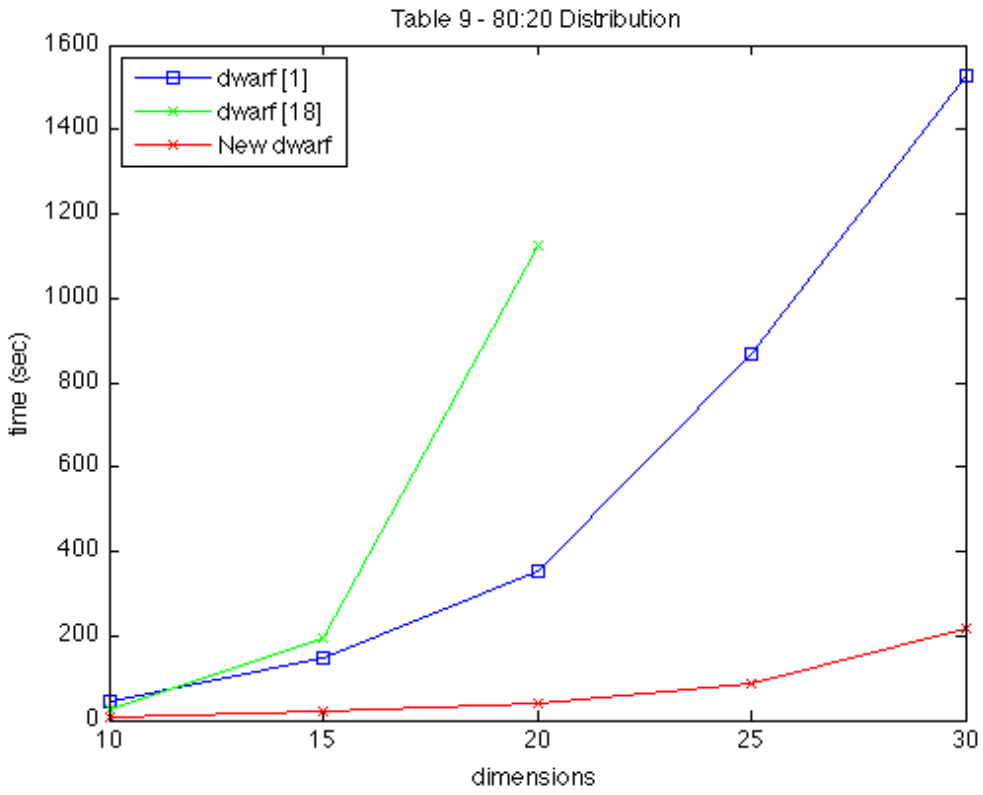


Figure 13: Results of Table 9 for #dimensions vs Time (80-20)

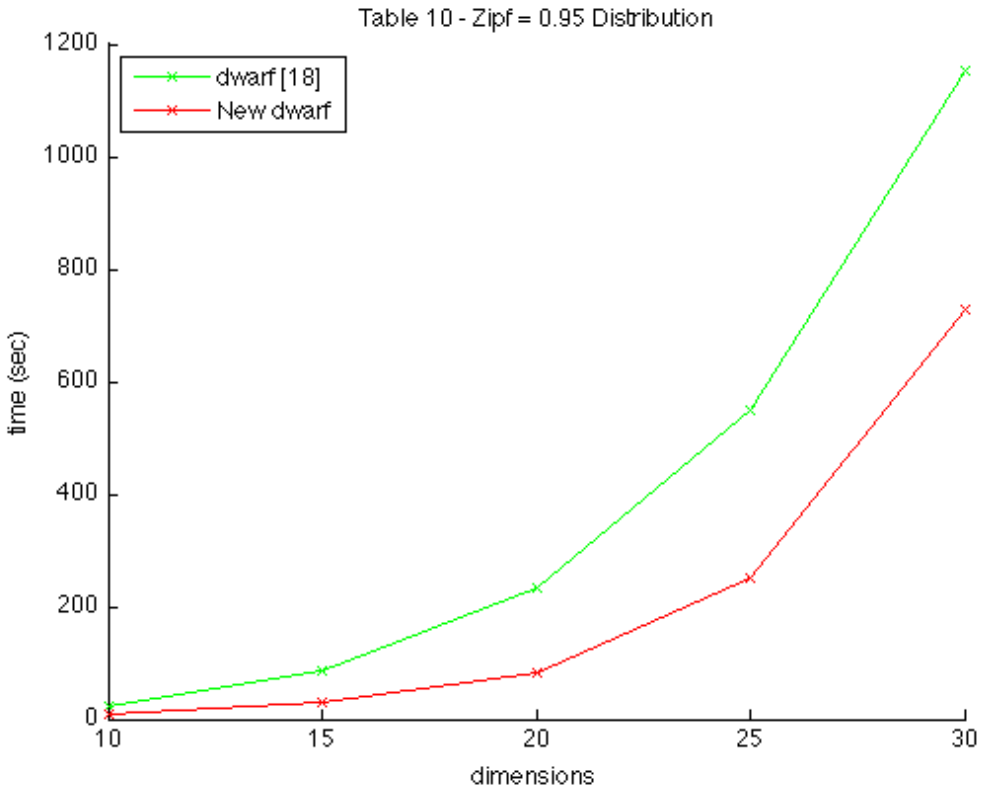


Figure 14: Results of Table 10 for #dimensions vs time (Zipfian)

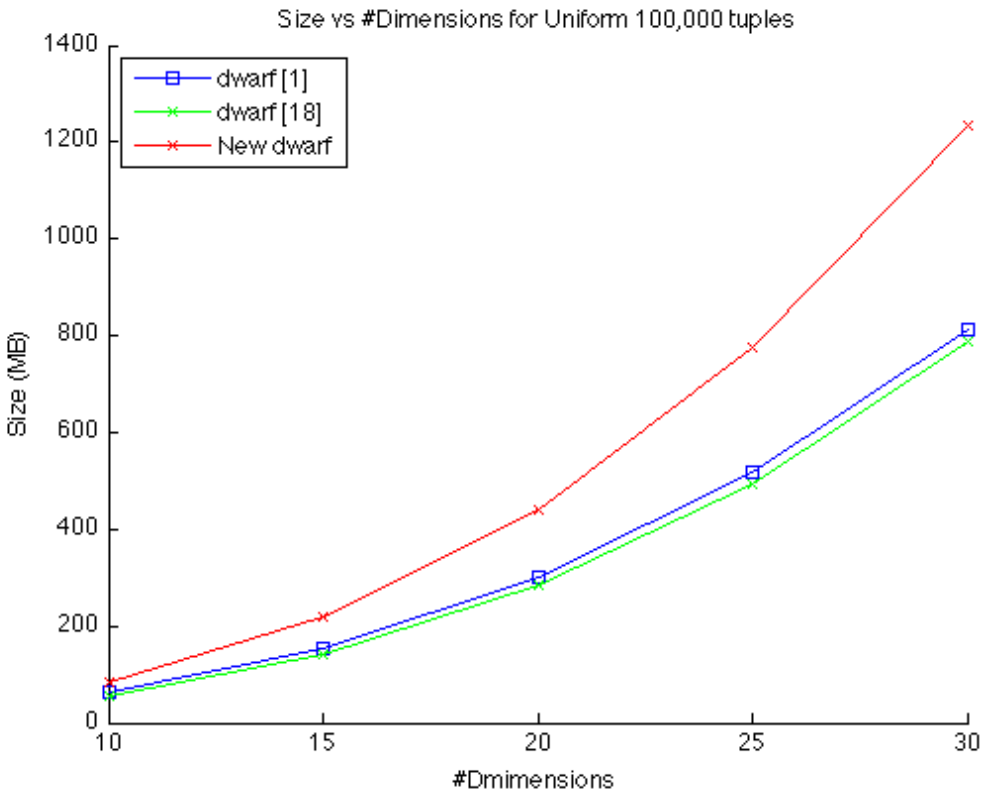


Figure 15: Results of #dimensions vs size for Uniform

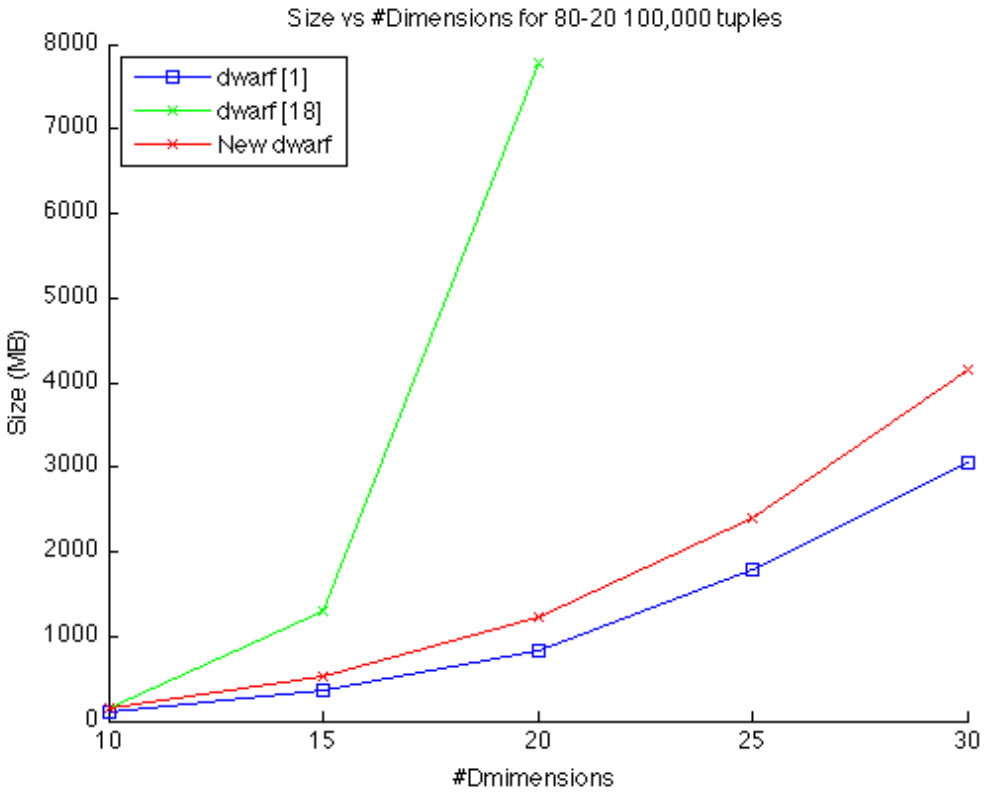


Figure 16: Results of #dimensions vs size for 80-20

For 80-20 self-similar data distributions, our implementation wins [18] and seems to be in sync with [1] regarding size reduction as shown in figures (15, 16). Figures (12,13,14) show fast runtime performance of our algorithm.

The next experiment examines the behavior of having more than one aggregate value in a data set. We generated 100,000 tuples, cardinality = 1,000 for each dimension, both for uniform and self-similar distribution. The results are shown below.

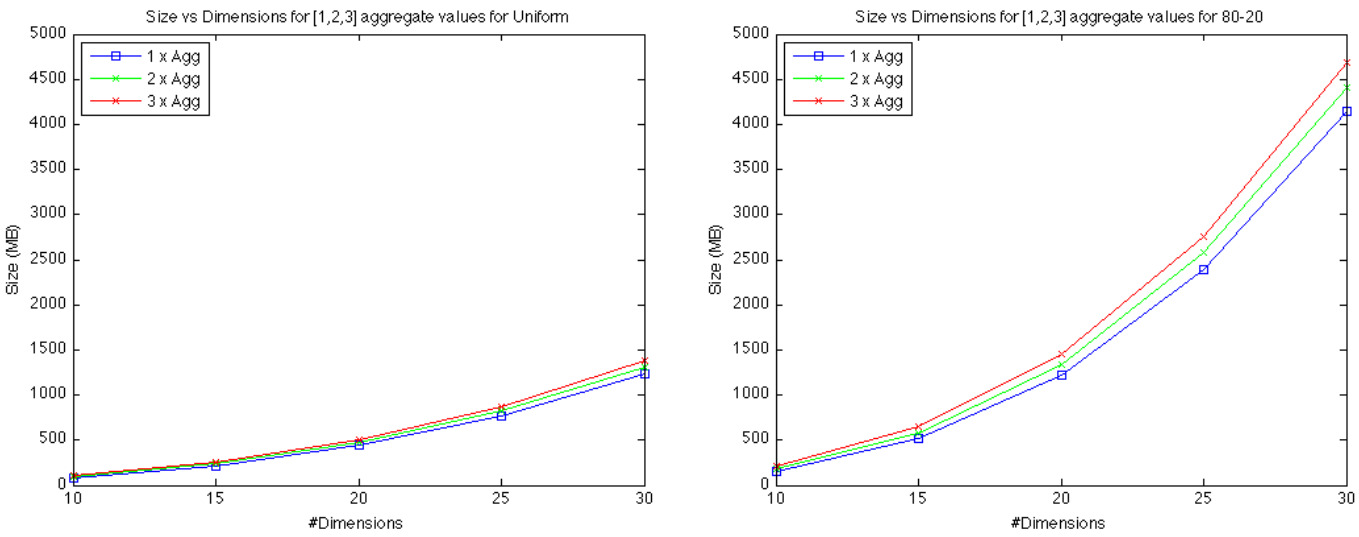


Figure 17: Scaling #dimensions and aggregate values

Increasing the number of aggregate values does not have any "interesting" impact on the generated dwarf size index. From our tests, scaling the number of dimensions and the number of aggregates, adds about 30-60 Mbytes while for self – similar data, we observe an additional 30 - 270 MBytes of extra size cost.

Our next experiment is reproducing the results of scaling Dwarfs as found in Figures 3 and 4 in [1]. A fact table containing 250,000 tuples created by either a uniform or a 80-20 self similar distribution. The dimensions are ranging from 4 to 10 in [1], and we extend the experiment to 30 dimensions. As the authors do not imply any cardinality for each dimension, we will use a cardinality of 1000. In figure 5 from [18] (a), (b), (c) authors compare storage space, construction time and expansion ratio towards the number of Dimensions.

#Dimensions	Cardinality for each Dimension Attribute = 1000, Tuples = 250,000			
	Size [MB] [f. Table : Dwarf Index]		Time [sec]	
	Uniform	80-20	Uniform	80-20
4	6.4 : 27	6.1 : 24	3	2.8
6	8.3 : 75	7.9 : 88	5	5
8	10.3 : 152	9.6 : 211	9	10
10	12.2 : 268	11.4 : 416	14	17
12	14.2 : 430	13.2 : 728	20	27
14	16.1 : 646	15 : 1168	27	41
16	18.1 : 924	16.8 : 1763	38	62
18	20 : 1271	18.6 : 2535	50	87
20	22 : 1696	20.3 : 3513	71	128
22	23.9 : 2205	22.1 : 4719	103	220
24	25.9 : 2808	23.9 : 6184	156	342
26	27.8 : 3512	25.7 : 7934	180	518
28	29.8 : 4328	27.5 : 9994	224	667
30	31.7 : 5255	29.3 : 12390	362	932

Table 11

This experiment shows that, as the original evaluation, Dwarf index ratio expansion remains modest when compared to the size of the fact table. As we can see below in Figures (18, 19, 20), and under 10 dimensions, both uniform and self-similar times, storage and construction ratio, are fluctuated on same levels. As the dimensions grow, for skewed distributions, both index sizes and construction times grow exponentially, reaching a ratio of 169:1 for Uniform and 427:1 for Self-Similar 80-20. Again, in comparison to [18] the authors could not scale further than 17 dimensions, reporting 473:1 for 80-20, which is greater than our expansion ration in 30 dimensions.

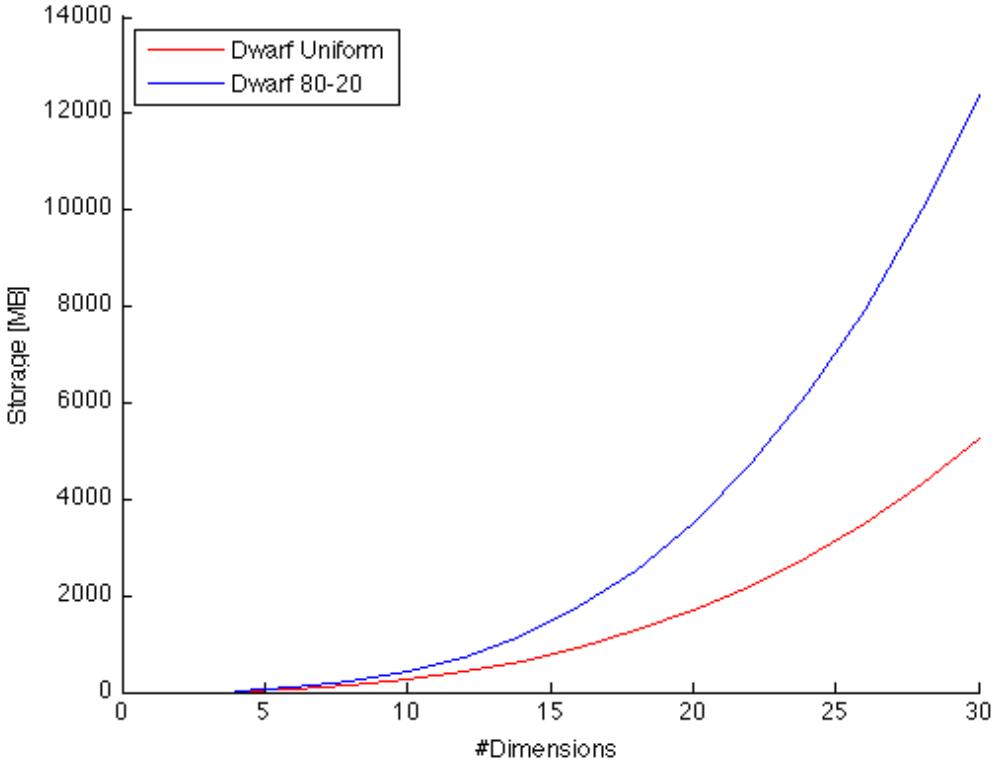


Figure 18: #Dimensions vs Storage [MB]

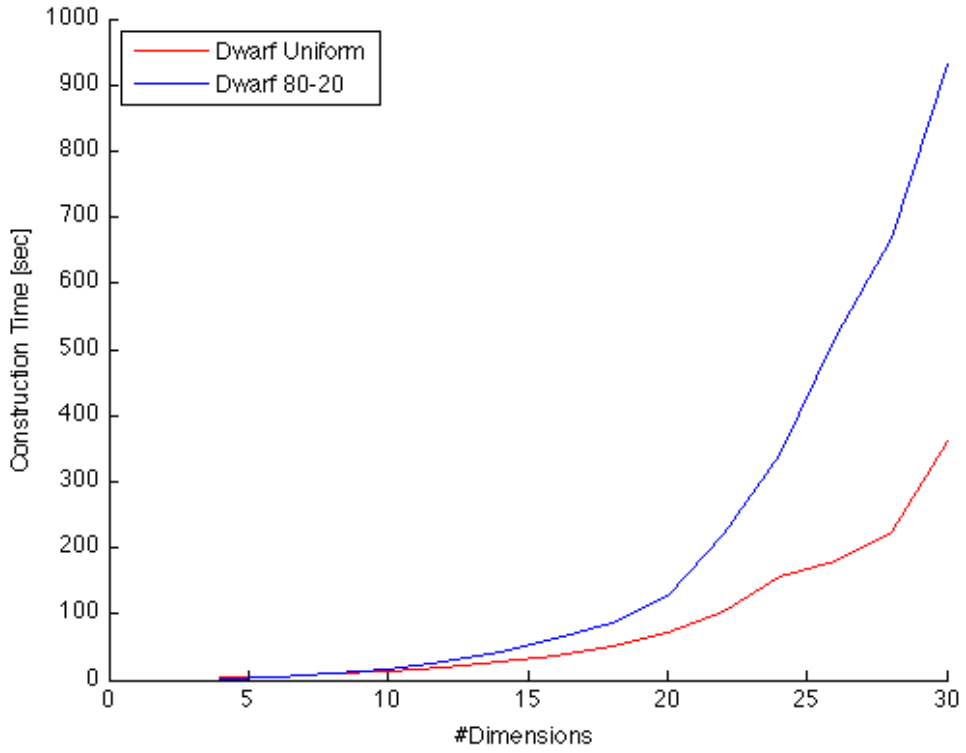


Figure 19: #Dimensions vs Construction Time [sec]

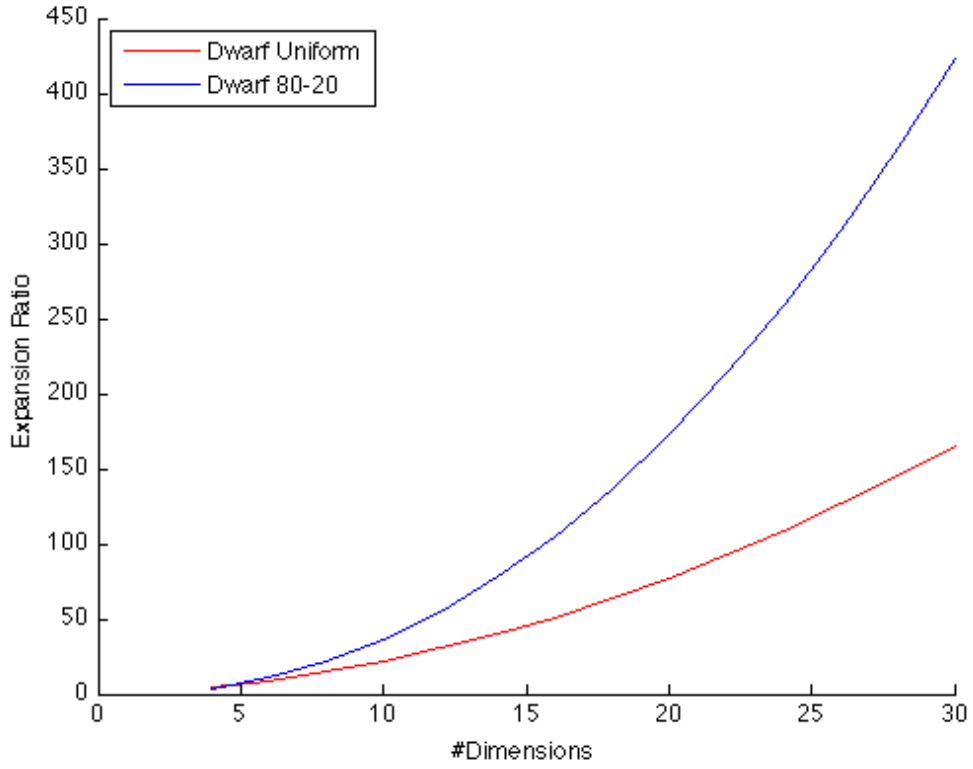


Figure 20: #Dimensions vs Expansion ratio

5.4.2 Scaling Tuples

The next experiments to follow are based on scaling the number of tuples. In Table 5 of [1] authors post Dwarf storage and computation time for a 10-dimensional cube. The number of tuples of each dimension varies from 100,000 to 1,000,000. More specific, we have:

	Dim1	Dim2	Dim3	Dim4	Dim5	Dim6	Dim7	Dim8	Dim9	Dim10
#card	30,000	5,000	5,000	2,000	1,000	1,000	100	100	100	10

#Tuples	F. Table Size [MB]	Uniform					
		Size [MB]			Time [sec]		
		Dwarf[1]	Dwarf[18]	NEW	Dwarf[1]	Dwarf[18]	NEW
100,000	4	62	53	66	27	8	5
200,000	8	133	113	142	58	19	9
400,000	17	287	239	324	127	43	17
600,000	25	451	372	491	202	64	26
800,000	34	622	509	684	289	90	35
1,000,000	42	798	651	873	387	114	45

Table 12

#Tuples	F. Table Size [MB]	Self-similar 80-20					
		Size [MB]			Time [sec]		
		Dwarf[1]	Dwarf[18]	NEW	Dwarf[1]	Dwarf[18]	NEW
100,000	4	72	94	83	31	16	6
200,000	8	159	195	182	69	35	10
400,000	17	351	400	432	156	70	21
600,000	25	553	608	669	250	109	32
800,000	34	762	818	932	357	146	43
1,000,000	42	975	1031	1173	457	188	56

Table 13

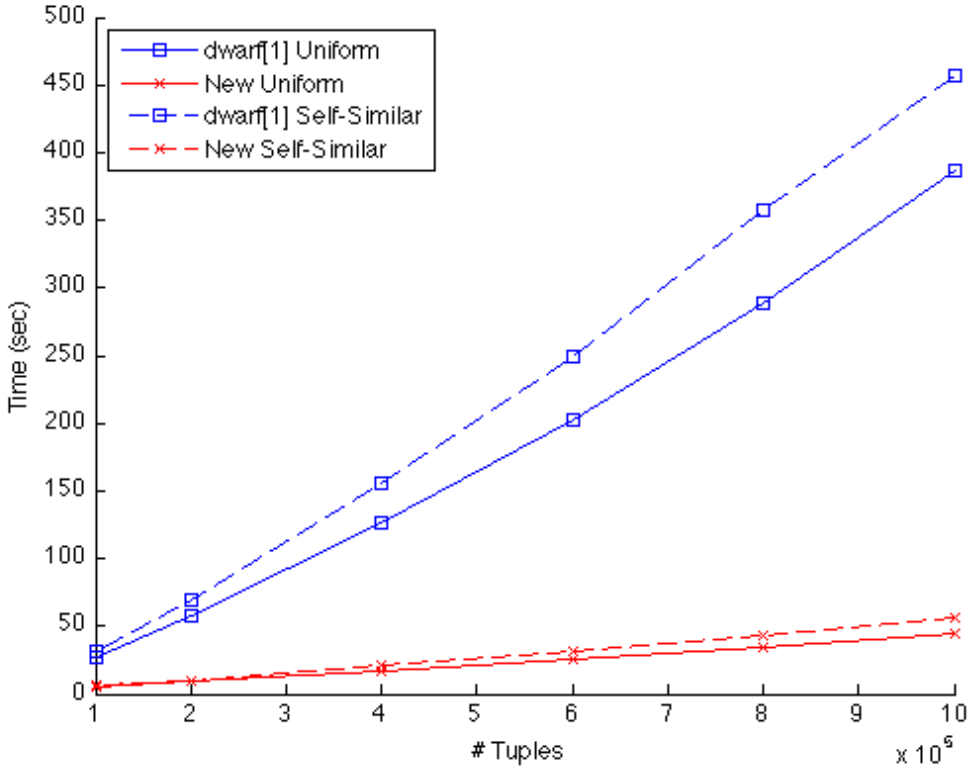


Figure 21: #Tuples vs Time [sec] for [1] and NEW

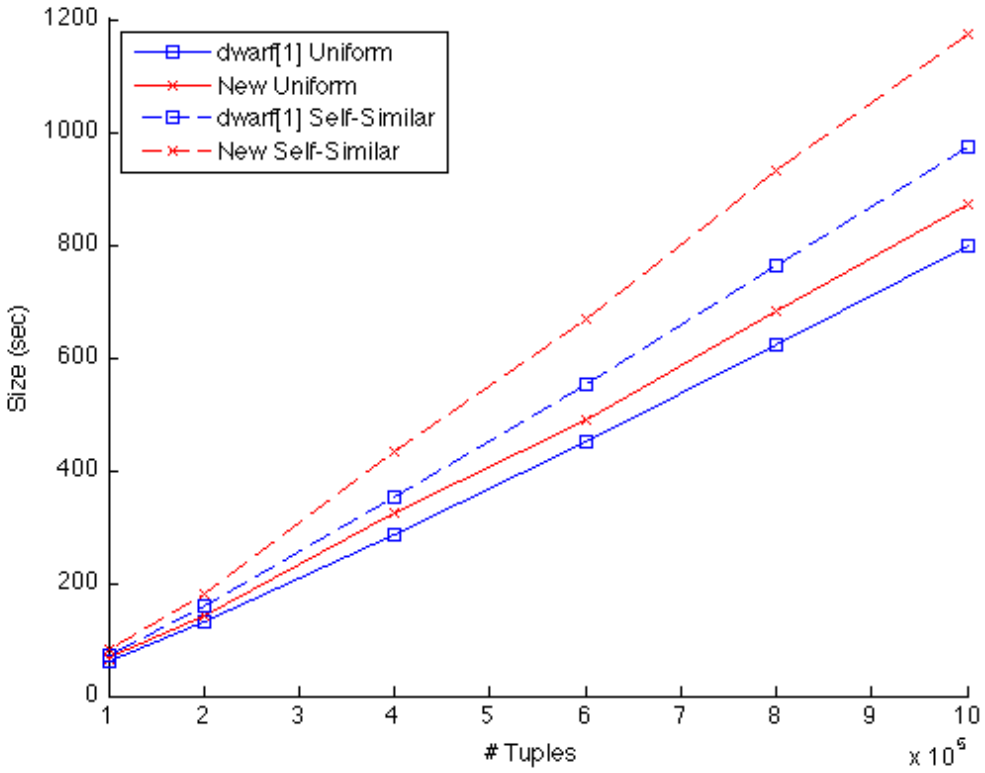


Figure 22: #Tuples vs Size [MB] for [1] and NEW

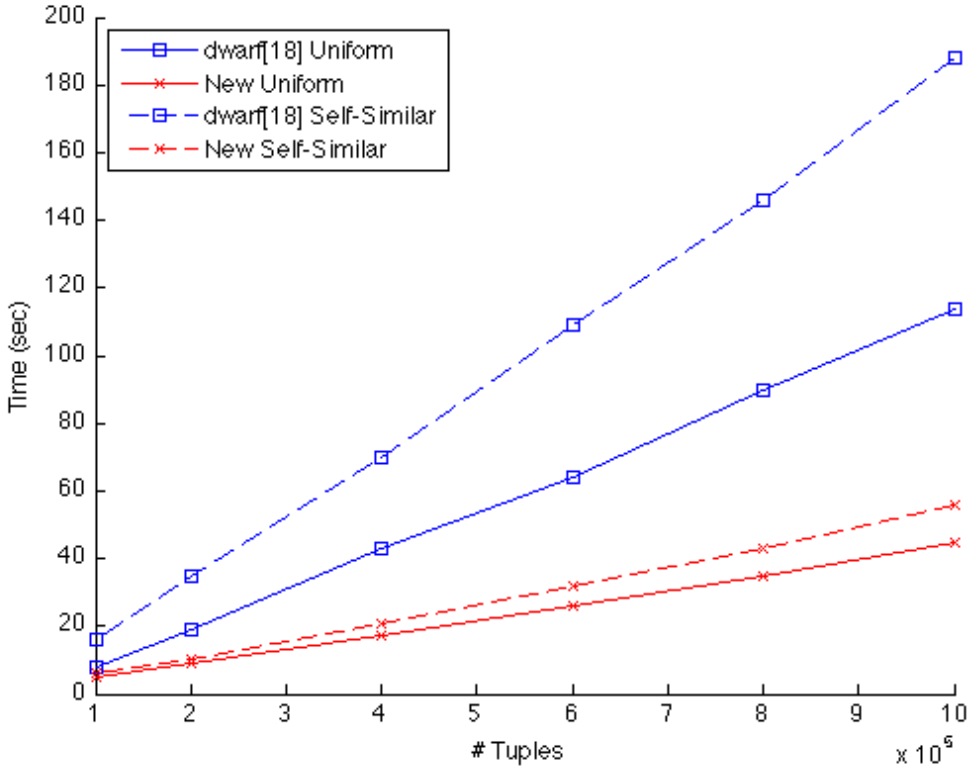


Figure 23: #Tuples vs Time [sec] for [18] and NEW

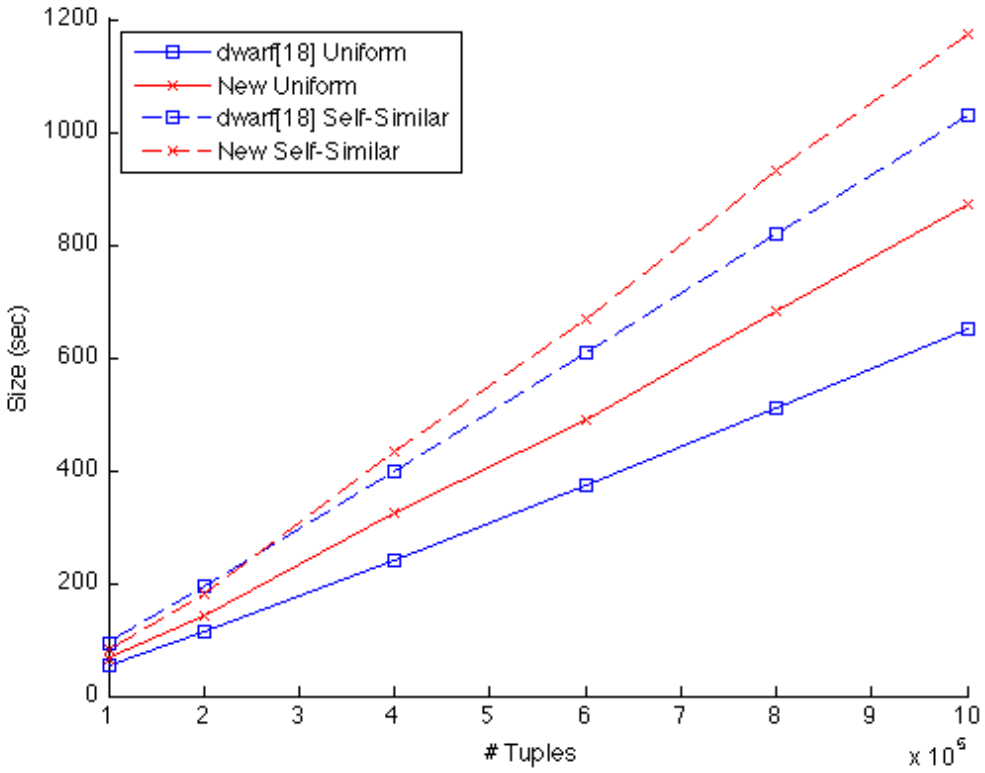


Figure 24: #Tuples vs Size [MB] for [18] and NEW

Graphical Comparison: we observe impressive savings in time (figures 21, 23). We manage to obtain similar scaling behavior in size as [18], but in comparison to [1] both our index sizes and expansion ratios are greater than those posted in [1], and much similar to [18] (figures 22, 24). One aspect of this result is the fact that our generators, using cardinalities [30000, 5000, 5000, 2000, 1000, 1000, 100, 100, 100, 10], and uniform parameters, in fact, produced larger data files than those stated in Figure 7 of [18]. Below, we show these variances in size. In [18] they report smaller sizes in fact tables against those that we used. Thus, we decided to compare expansion ratios as shown below:

#Tuples	Fact Table Size [MB]		
	[18]	Uniform	80-20
100,000	4	4.8	4.6
200,000	8	9.7	9.1
400,000	17	19.3	18.2
600,000	25	29	27.3
800,000	34	38.6	36.5
1,000,000	42	48.3	45.6

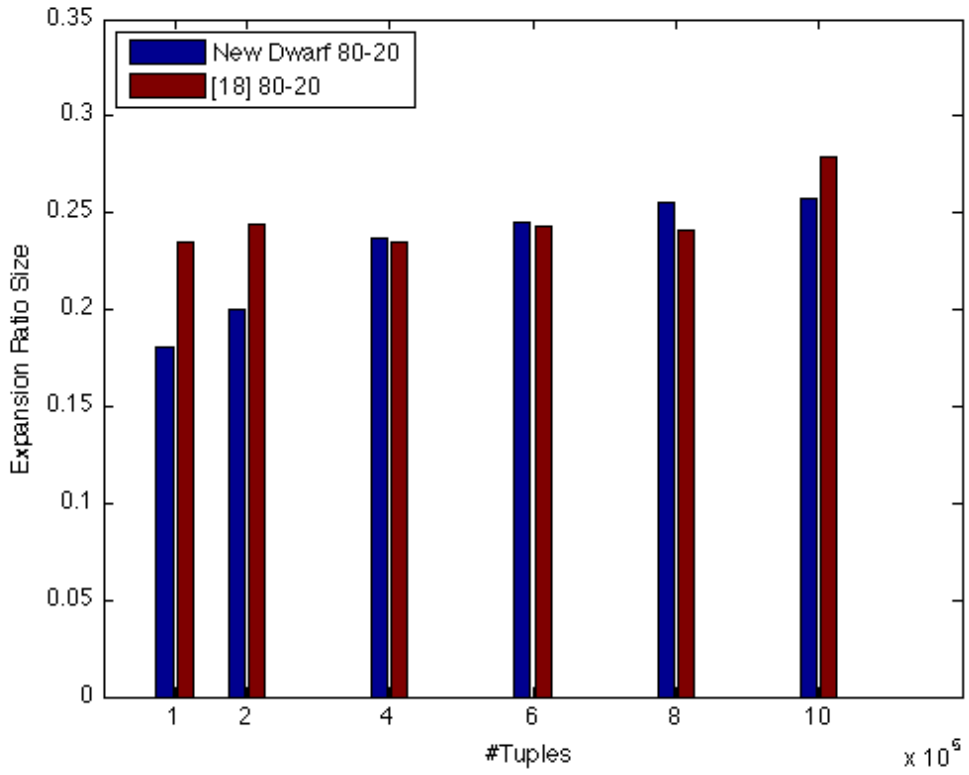


Figure 25: #Tuples vs Expansion Ratio Size for [18] and NEW

From the above (figure 25) it is clear that for 80-20 distributions, we achieve at most cases smaller size expansion and of course better size reduction. Tables 12 and 13 show results as reported by [1], [18] and our results (NEW). The results for uniform data show that the index sizes obtained by NEW are slightly bigger than the other two. This might be caused because our fact table sizes for the reported number of tuples, were larger than those described. However, index creation times are shown in figures 16 and 18 to be the fastest compared to the other two methods, as NEW is by a factor 5-8 faster than [1], 2 - 3.5 runtimes faster than [18] (figure 26, 27).

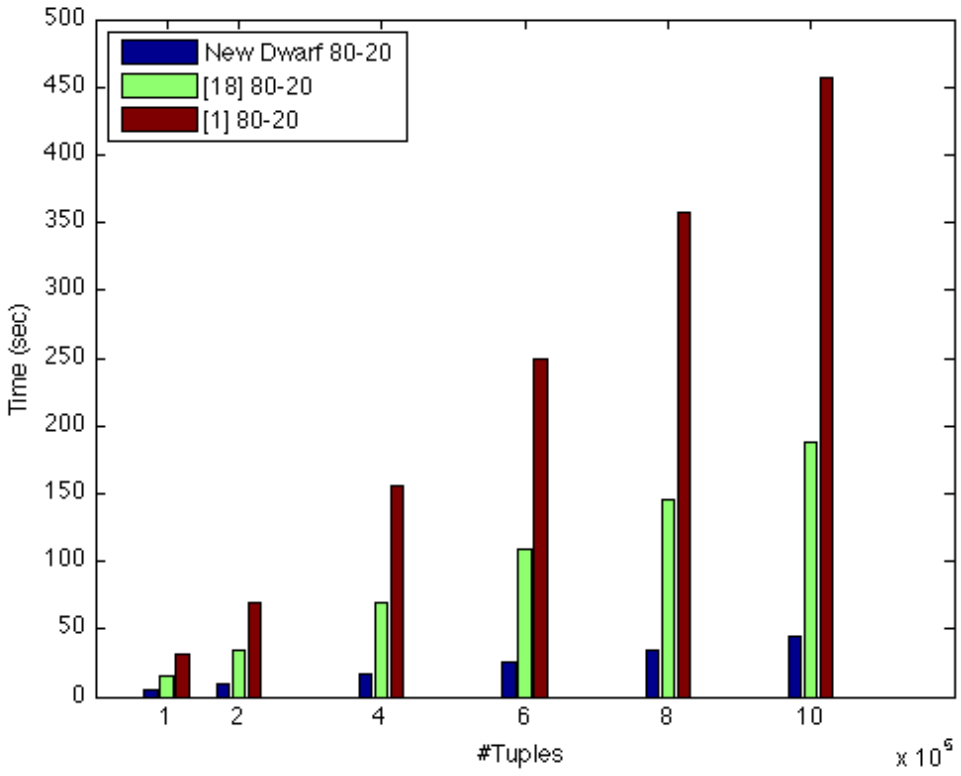


Figure 26: #Tuples vs Time [sec] for [1], [18] and NEW

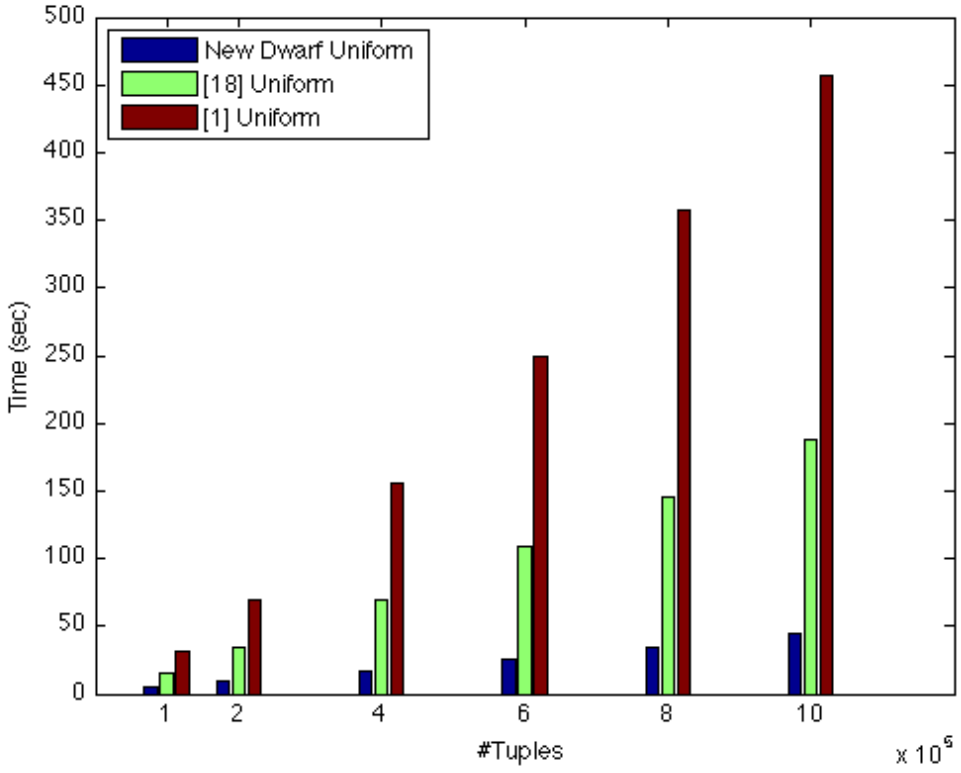


Figure 27: #Tuples vs Time [sec] for [1], [18] and NEW

For the next experiment will keep a cardinality of 1,000 over 15 dimensions, and repeat the experiments scaling the tuples from 1,000,000 to 10,000,000. We will compare the results by repeating the experiment of various cardinalities, (30000, 20000, 5000, 5000, 5000, 2000, 2000, 2000, 2000, 1000, 1000, 100, 100, 100, 10)

#Tuples	F. Table Size [MB]	Uniform				80-20			
		Size [MB]		Time [sec]		Size [MB]		Time [sec]	
		Scaling card.	1000 card	Scaling card.	1000 card	Scaling card.	1000 card	Scaling card.	1000 card
100,000	7.2	182	219	11	12	202	300	11	13
200,000	14.4	392	564	20.5	24	448	758	23	29
400,000	28.7	862	1553	42	62	1010	1893	47	71
600,000	43.1	1387	2835	67	113	1640	3186	76	174
800,000	57.5	1932	4307	114	296	2318	4568	121	444
1,000,000	71.8	2520	5897	176	539	3036	6004	265	711

Table 14: Experiments scaling #Tuples for Uniform and 80-20 self similar

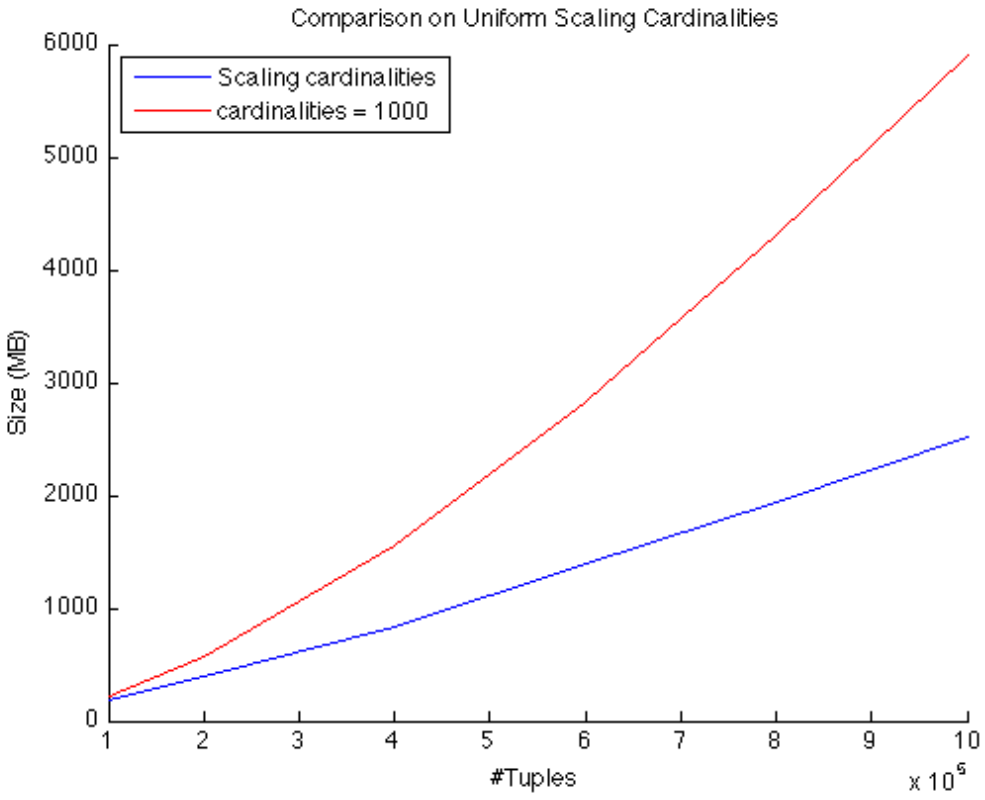


Figure 28: #Tuples vs Size [MB] for Table 14 (Uniform data)

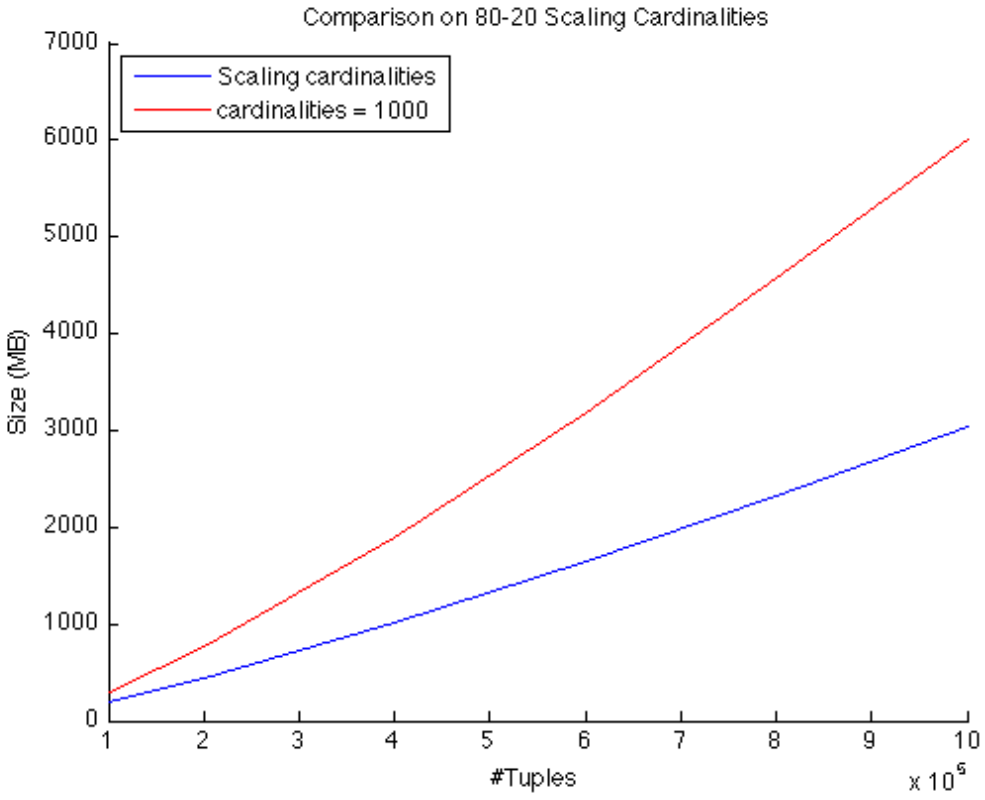


Figure 29: #Tuples vs Size [MB] for Table 14 (80-20 data)

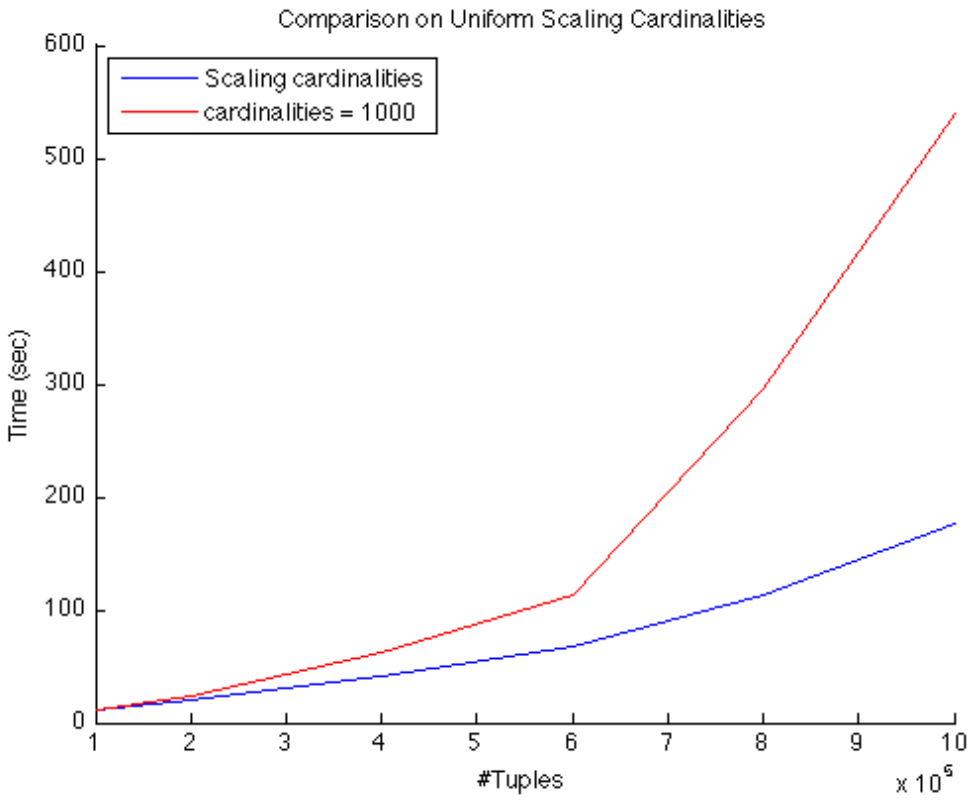


Figure 30: #Tuples vs Time [sec] for Table 14 (Uniform data)

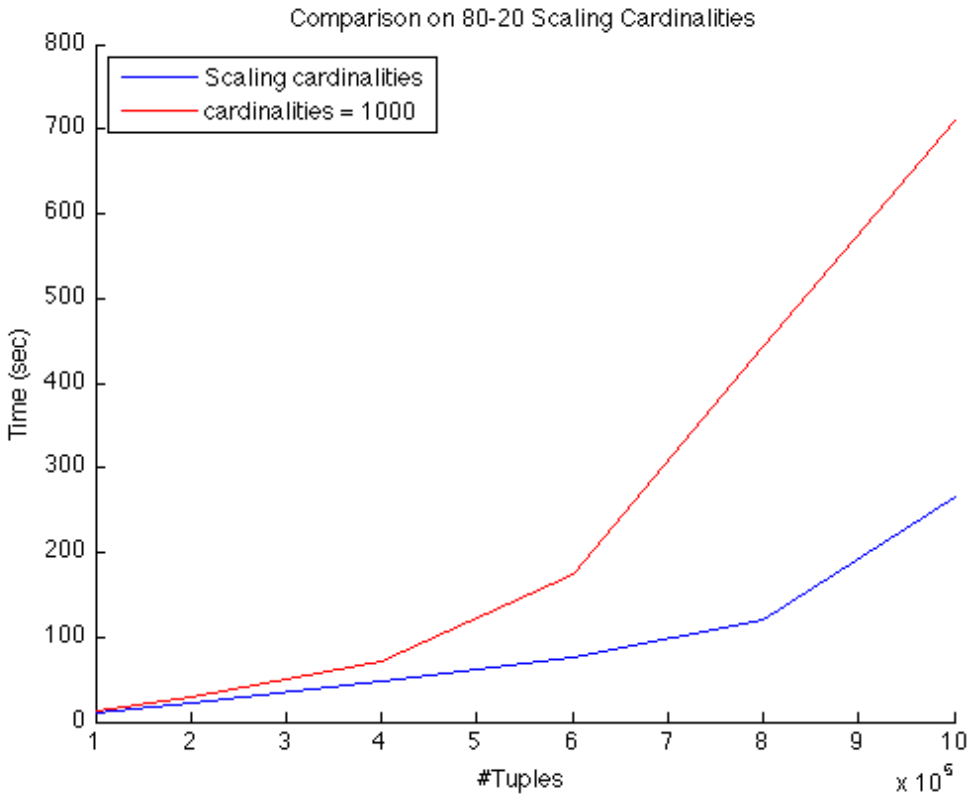


Figure 31: #Tuples vs Time[sec] for Table 14 (80-20 data)

From the above figures, we observe that keeping cardinalities at 1,000 each, Dwarf index sizes as well as index construction times are considerably larger than having a various number of cardinalities on each dimension. This clearly is in total sync with the fact that cardinalities have major impact on construction size and times. Expansion index ratios are shown below:

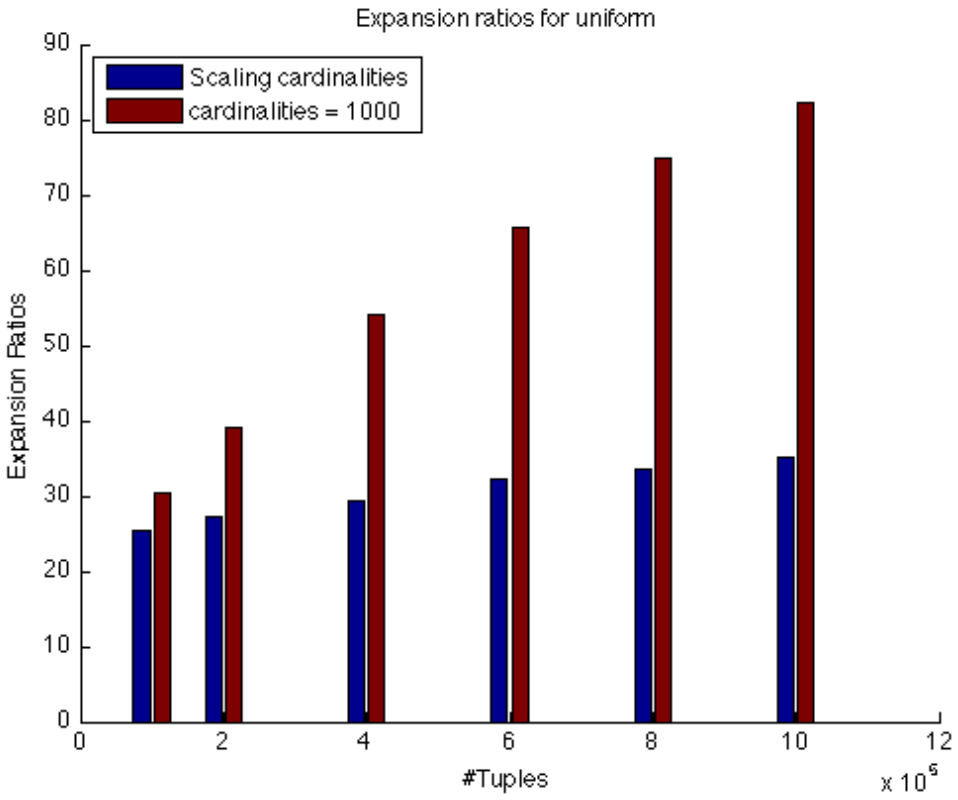


Figure 32: #Tuples vs Expansion Ratios for Uniform Data

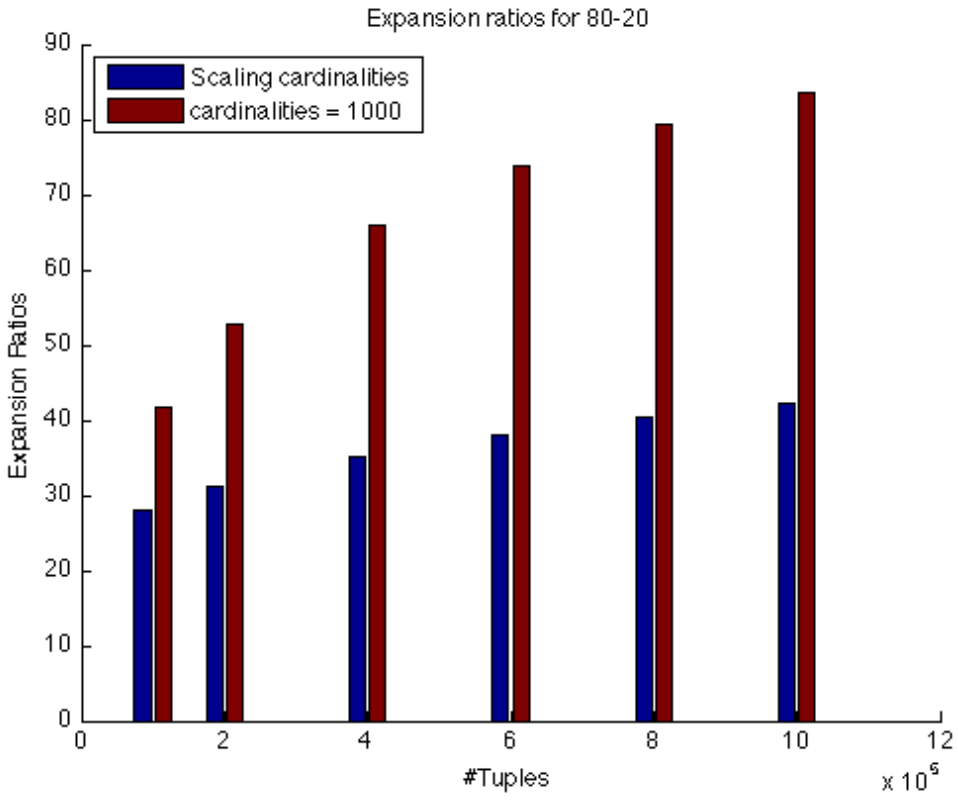


Figure 33: #Tuples vs Expansion ratios for 80-20 Data

In summary, all the experiments showed that both for uniform and 80-20 data, our results have faster time construction, and smaller expansion ratios. The index sizes increase as we produce data sets with denser prefix and suffix areas.

5.5 Safe Views Index Construction Experiments

Our next experiments are performed given a number of views. For a scaling of dimensions from 6 to 30, we repeat the above experiments, in order to evaluate the performance on creating index cubes based on specific views. For this purpose, we use a view's generator to randomly pick dimensions of the fact table. The above experiments will be repeated in order to evaluate size and time construction. Each dimension had exactly 50% probability of appearing in each view. Of course, as the dimensions will grow, the generated views will also grow. We generated 500,000 tuples, over a cardinality of 1000, both for uniform and 80-20 distribution.

One optimization we made, in order to accelerate the process of finding best permutation, was to detect which are useful dimensions contained into the randomly generated views. We know that for a given set n of dimensions, the total permutations are $n!$. Reducing the number of possible dimensions, which contribute on safe views, accelerates the process due to the fact that we have much less number of permutations to generate and check.

As dimensions grow, building support dwarfs becomes more contingent. Again, we used synthetic data as described in 6.3 for uniform and 80-20 self-similar data distribution.

5.5.1 Scaling Views

At first, we will try to investigate the behavior of the algorithm, by scaling the number of views for a number of dimensions). As total views and dimensions grow, the permutation generator and safe view discovery become more and more time–consuming. Thus, we decided to check only a specific random number of permutations, for each experiment. Also, as views grow,

we add only new randomly picked to the previously selected, and random permutations to check, remain the same.

For the first experiment, we used an 8-dimensional fact table, followed by uniform distribution, 1,000 cardinalities for each dimension, and 500,000 tuples. For our experiments, we used 1,000 random permutations to search for safe views and the results are shown in table 15.

#Views	Perm Time [sec]	#Not Safe Views	Size Dwarf [MB]	Time Dwarf [sec]	Size Support Dwarf [MB]	Time Support Dwarf [sec]
15	4.6	13	13.3	4.8	49.5	5
30	9.4	24	42.2	7	42.2	4.8
45	13	29	49.5	5.1	42.2	4.6
60	14	30	121.6	9.7	35	5.6
75	16.6	35	175	10	175	10
90	23	40	350	21	175	17
105	26	39	416	25	350	23
120	29	37	416	28	416	27.5
135	34	34	416	21	416	25
150	32	34	416	22	416	21
165	47	32	416	22	416	20
180	43	31	416	24	416	21
195	62	28	416	25	121	14.3
210	57	21	416	23	49.9	12.6
225	69	9	416	25	121	9.9
240	85	52	416	24	37	4.9
254	1	0	416	21.5		
ORIG	**-**	**-**	**416 : 20.4 (MB)**	**18.5 (sec)**	**-**	**-**

Table 15

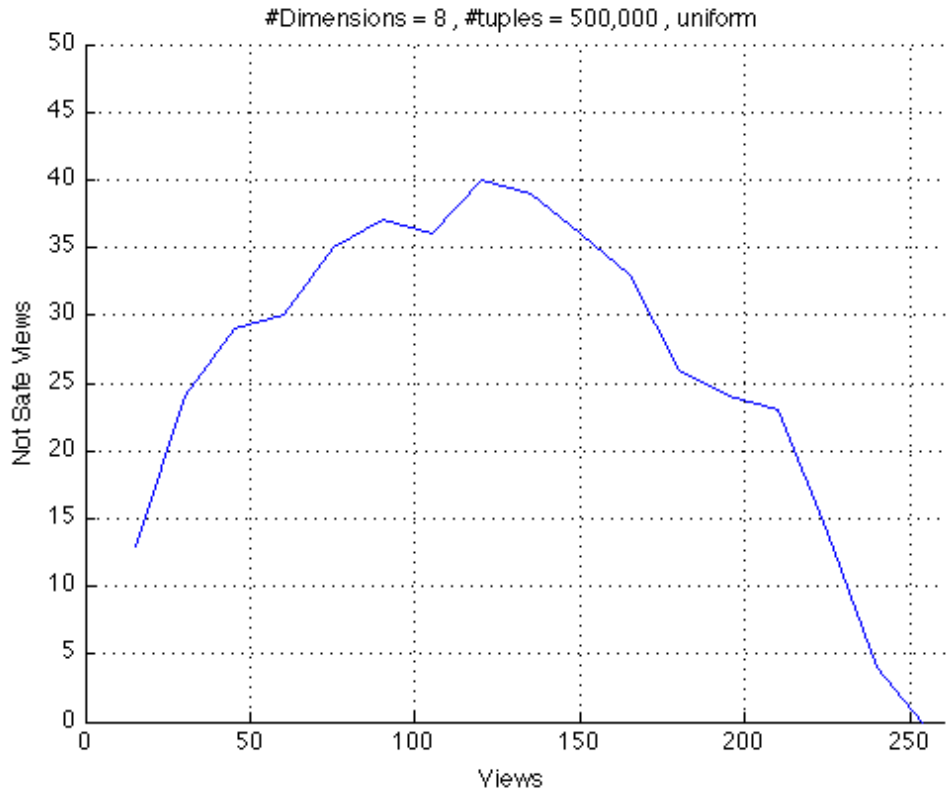


Figure 34: #Views vs NotSafe Views for D=500,000 , c = 1000, Uniform

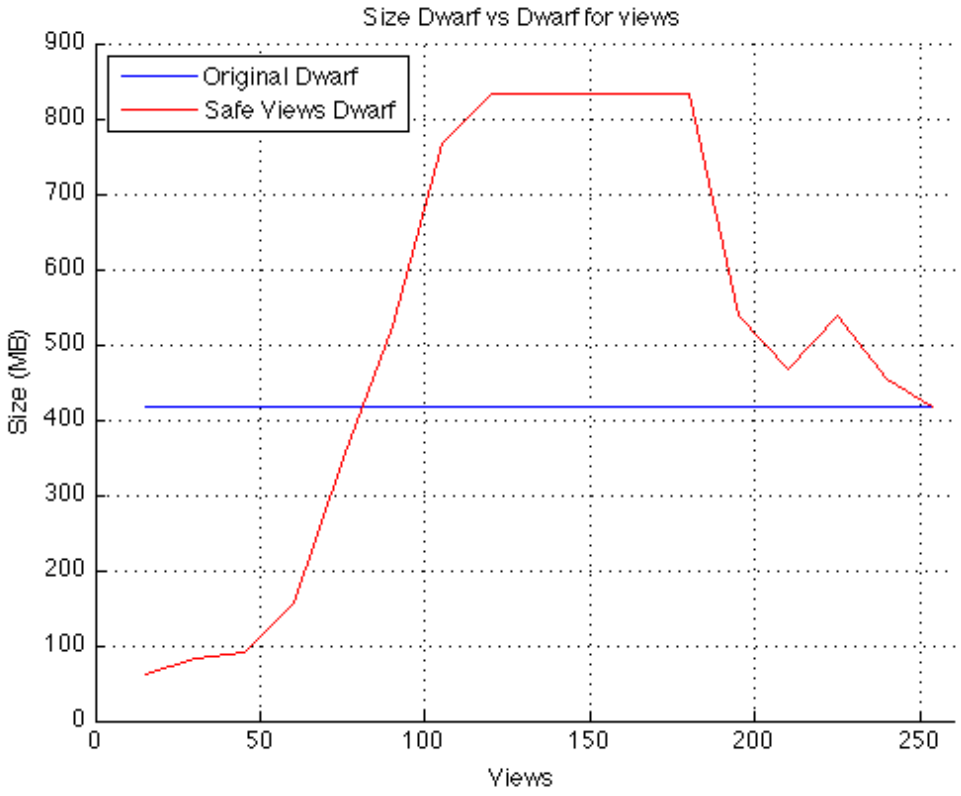


Figure 35: Views vs Size[MB] for D = 500,000, c = 1000, Uniform

Our next experiment contains nine dimensions using 500,000 tuples at uniform distribution:

#Views	Perm Time [sec]	#Not Safe Views	Size Dwarf [MB]	Time Dwarf [sec]	Size Support Dwarf [MB]	Time Support Dwarf [sec]
30	9.9	25	92	7.6	66	6.9
60	21	39	226	14	90	8
90	25	52	226	12	226	14
120	36	66	226	14	280	15.6
150	52	75	260	15	395	17.3
180	62	80	226	11.8	395	16.7
210	97	86	395	17.4	226	15
240	111	81	600	31	600	30
270	128	81	600	29	600	30
300	153	91	600	28	600	30
330	187	72	600	30	600	30
360	198	61	600	32	470	25
390	238	54	600	31	600	30
420	257	41	600	32	240	13.6
450	272	30	600	31	226	11
480	295	10	600	31	90	9
510	1.7	0	600	28.7		
ORIG	**-**	**-**	**600 : 28 (MB)**	**18.5**	**-**	**-**

Table 16

We observe that for more than 100 views, we are constructing two full cubes, which is totally expected, due to the "strict" rules for safeness and construction. As views grow, more and more dwarf chunks are forced to be build. If we decide to extend the rules and offer more flexibility, would lead to produce more safe views per dwarf. *Iff* the dwarf index size could be predicted, we would be able to rule out cases of not computing two full dwarfs, instead overcome the safe views and build a full dwarf containing all possible views.

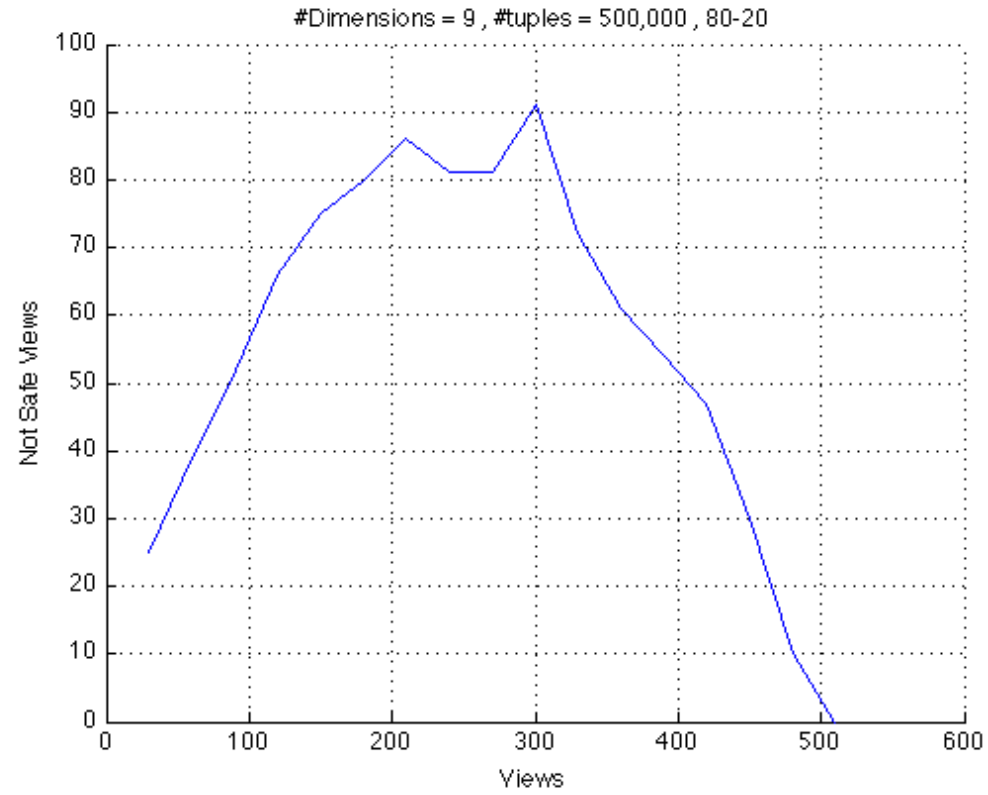


Figure 36: #Views vs NotSafe Views for D=500,000 , c = 1000, 80-20

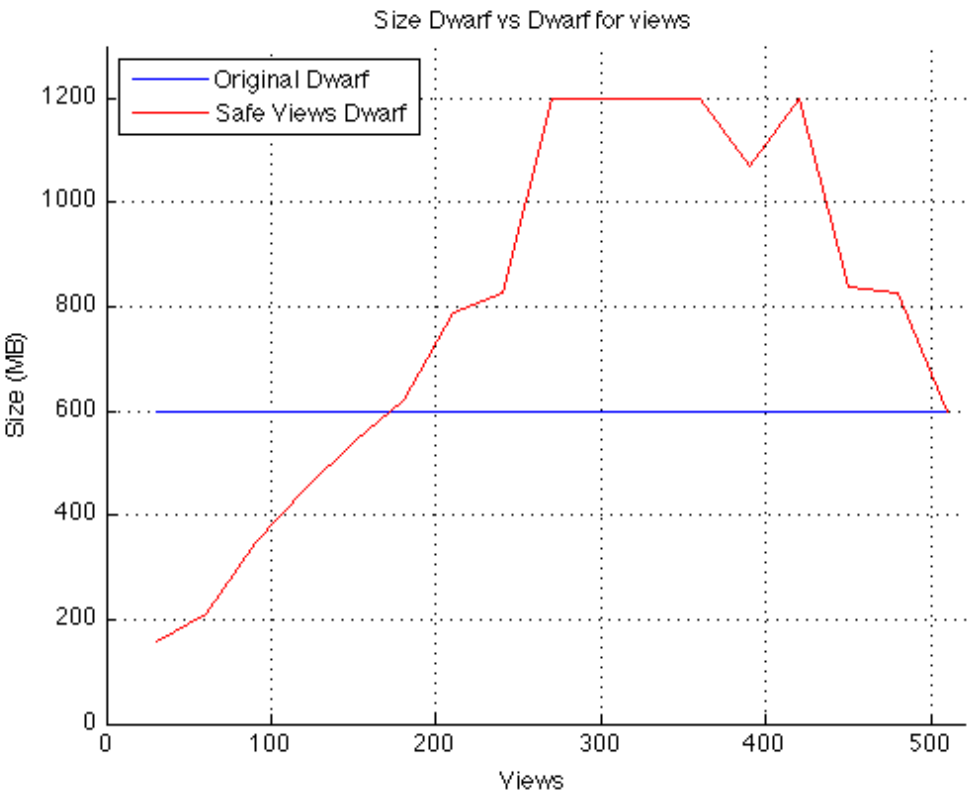


Figure 37: Views vs Size[MB] for D = 500,000, c = 1000, 80-20

As it can be clear from the above figures (35, 37), we have been able to reduce size of index cube for a various number of views. As more views are added, the more partitions of dwarf tree index need to be built, leading to a full cube. At peak, we unfortunately, as expected, are forced to create two full dwarf cubes. The most size reduction is reported when we input randomly a small number of views (max 15 – 90 for the above experiments figures 35, 37). This is due to the fact that, the less views are inserted, it is more unlikely that a full index cube will be requested to be built to support them.

Again, we create the sub-dwarfs of the "ALL" nodes following each path of input views. We have managed to store *aggregateValues* on levels, as well as be able to overcome the coalescing issues we described above. Avoiding writing to disk the outcomes of *suffixCoalescing* on level paths where we do not want "ALL" values has been a major factor. Thus, we have reduced the size of Dwarf, while we maintain the path of the nodes through the tree index.

5.5.2 Scaling Dimensions

For our next experiments, we decided to use a more realistic number of views, and scale the number of dimensions. Thus, we will use 30-60-90 views, for a scale of dimensions from 10 – 14 to see the results. We generated 300,000 tuples uniform distribution, over a cardinality of 1,000. Each experiment will be repeated 3 times, using different random views and permutations each time.

	# Dimensions - Uniform distribution, # Tuples = 300,000, Card = 1000								
#Views	10			12			14		
30	69	30	79	63	61	60	88	98	98
60	144	60	82	123	98	72	93	95	132
90	162	90	228	312	216	216	112	252	113
ORIG (MB)	**290 : 14.7**			**360 : 17**			**840 : 19.4**		

Table 17

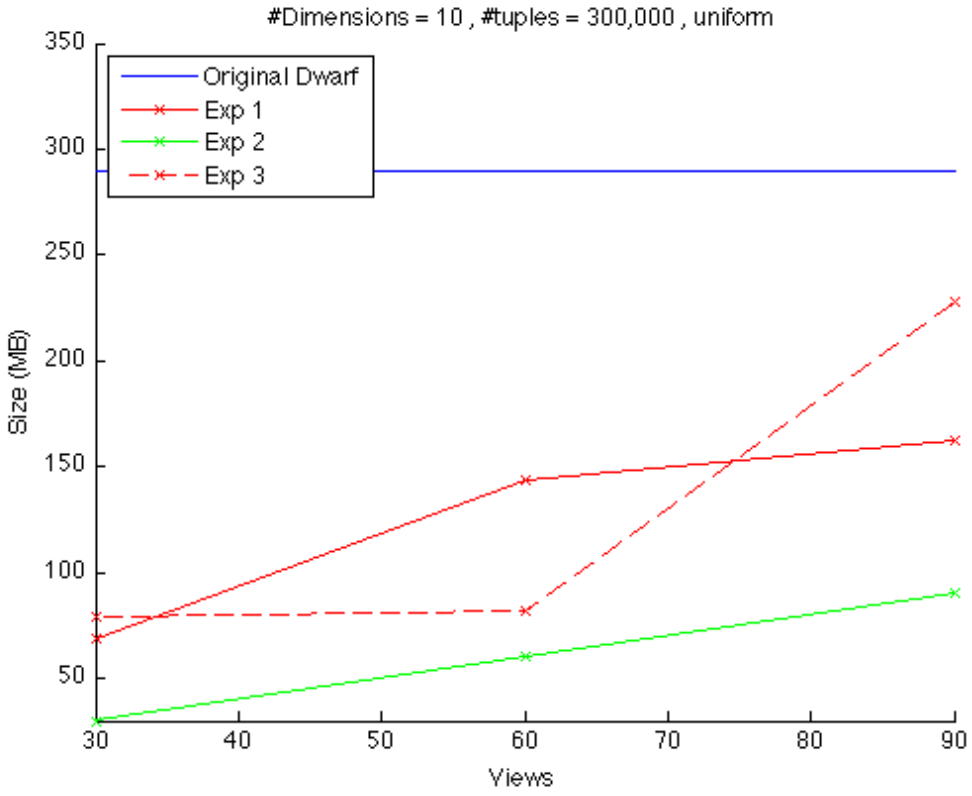


Figure 38

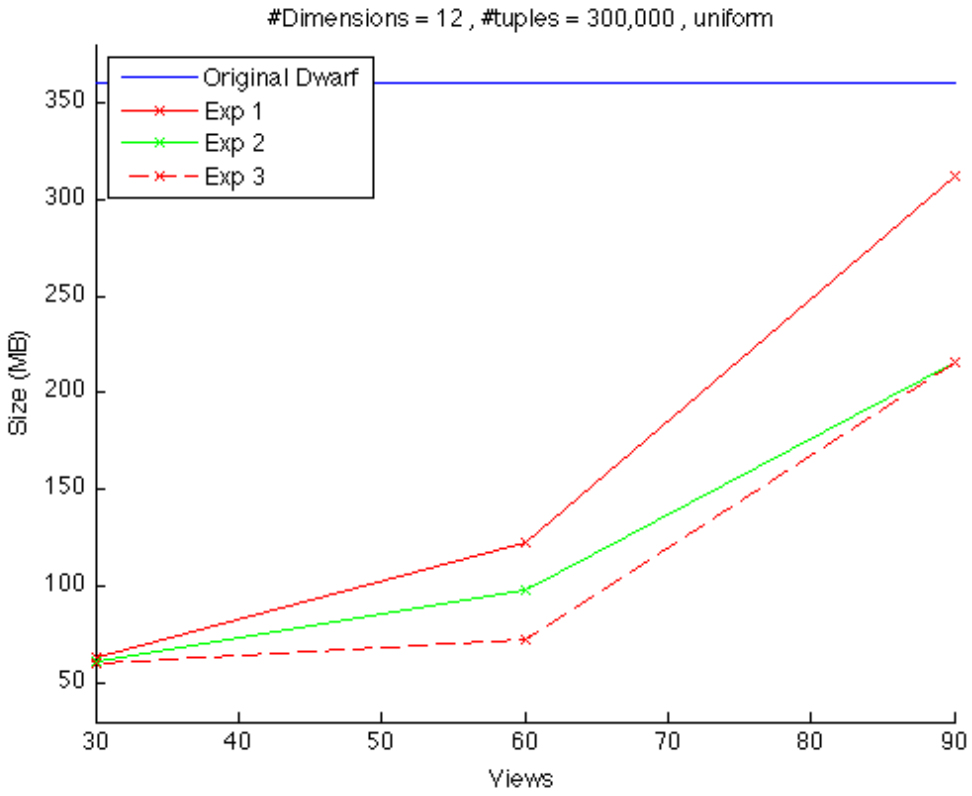


Figure 39

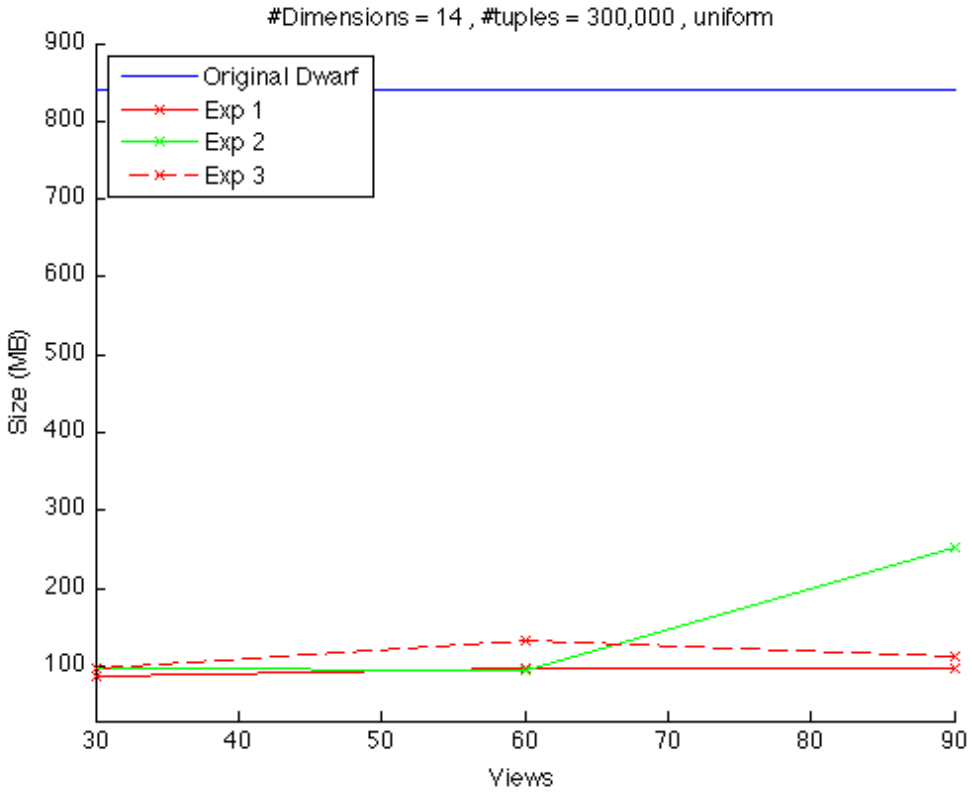


Figure 40

It can be clearly shown that we succeed to reduce the size of the dwarf (figures 38, 39, 40). Of course, those index do not support every possible view. Thus, for those views that finally safe, the requested dwarf index is smaller than the actual cube. We manage to reduce the index cube by a factor of 4 to 9. The percentage of size reduction relies on a various number of factors for the above experiments such as:

- #Views vs #Dimensions
- Amount of not safe views
- Distribution of the fact table

Chapter 6

Query Performance

In this section we will reproduce the experiments both in [1] and [18] whose purpose is to evaluate the query performance of the cube. We created two workloads of 1000 queries followed by the following probabilities: P_{newQ}, P_{dim}, P_{pointQ}. We range the dimensions from 4 to 30.

- P_{newQ} refers to the probability that a new query will not be related to the previous one. Thus, in [1] mention the authors assign a $(1-P_{newQ})/2$ for a roll-up query, $(1-P_{newQ})/2$ for a drill-down query, and P_{newQ} for a new query. Workload A is generated having roll-up or drill-down queries equal to 0.33. Workload B, contains only unrelated queries
- P_{dim} is the probability that each dimension will be selected to participate in the new query. A p = 0.4 means 4 dimensions on average for a 10-dimensional cube, or 12 on average for a 30-dimensional cube.
- P_{pointQ} refers on how selective the queries will be: a value of 1 produces only point queries, while a value of 0 produces queries with range query on every dimension. Using low parameter p = 0.2, we achieve for most queries to return more tuples, while a high value of this parameter would lead to return very few tuples.

Range specifies the range of values contained in the queries. If the data set is generated with a cardinality of 1000, a range of 20% forces the queries to hold only 200 values.

		Probabilities			Range	
Workload	#Queries	P_{newQ}	P_{dim}	P_{pointQ}	Max	Min
A	1000	0.34	0.4	0.2	20%	1
B	1000	1	0.4	0.2	20%	1

Table 18 : Workload Characteristics for "Dwarfs vs Full Table Scans" Query Experiment (see Table 7 in [1])

As described in 6.1 [18], we generated a 250,000 tuples fact table, using uniform distribution, with cardinalities in each dimension = 1,000. For each experiment, in order to avoid pre-caching of the dwarf index cube on memory, for each dimension, we followed the following experiment procedure:

1. Create a new data set for each dimension
2. Build the full cube for this data set
3. Create workload A
4. Run the query test
5. Delete the full cube and rebuilt it for the same data set
6. Create workload B
7. Run the query test

Dimensions	Uniform Data				80-20 data			
	Workload A		Workload B		Workload A		Workload B	
	Hot	Cold	Hot	Cold	Hot	Cold	Hot	Cold
4	0.36	0.4	0.3	0.24	0.3	0.7	0.3	0.9
5	0.35	1.3	0.32	0.8	0.35	0.87	0.32	1.49
6	0.39	0.6	0.35	2.3	0.37	2.1	0.35	3.7
7	0.4	2.2	0.38	2.5	0.41	2.8	0.38	5
8	0.43	1.5	0.39	4.8	0.43	3.5	0.38	6.6
9	0.43	3.6	0.4	4.26	0.43	4.3	0.4	7.3
10	0.48	2.8	0.43	6.9	0.46	5.3	0.44	8.1
15	0.5	6.6	0.5	8.9	0.67	9.7	0.52	11.8
20	0.6	8	0.58	10.5	0.62	13.2	0.58	15.6
25	0.64	9.9	0.60	12.3	0.65	14.1	0.65	17.5
30	0.7	12.5	0.65	14.6	0.7	15.9	0.69	18.7

Table 19: Runtime performance for Workload A/B

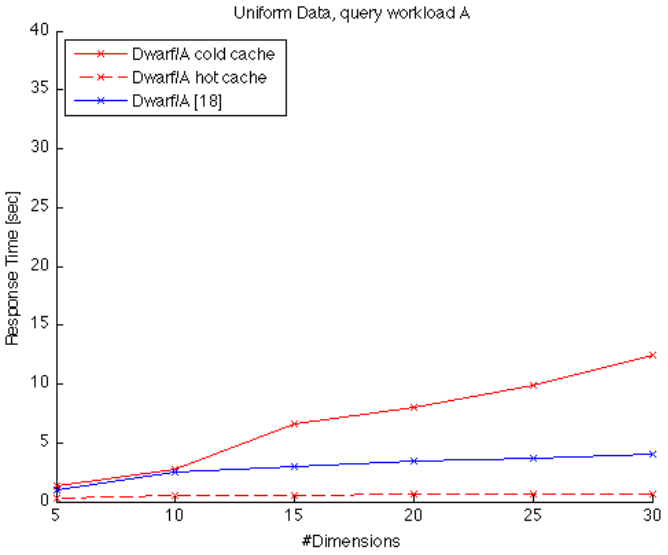


Figure 41: Response Times for Uniform Data on Queries Workload A vs [18]

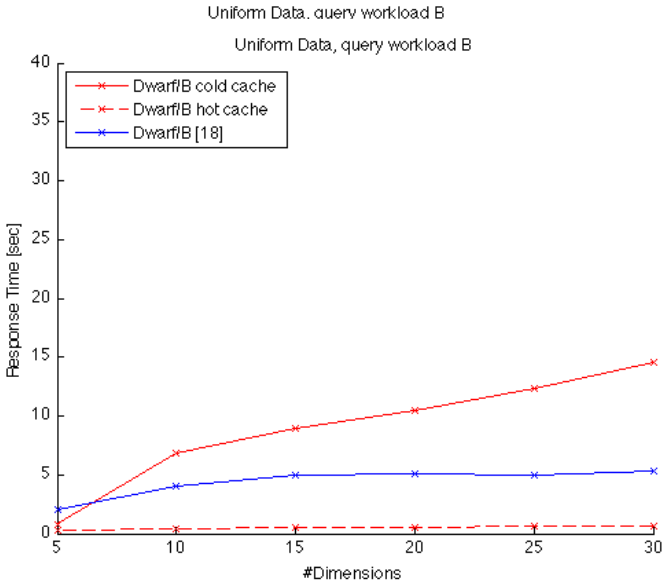


Figure 42: Response Times for Uniform Data on Queries Workload B vs [18]

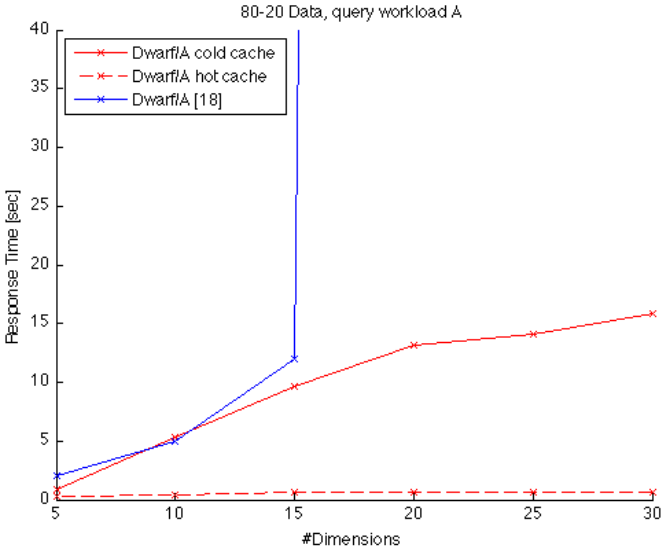


Figure 43: Response Times for 80-20 self similar Data on Queries Workload A vs [18]

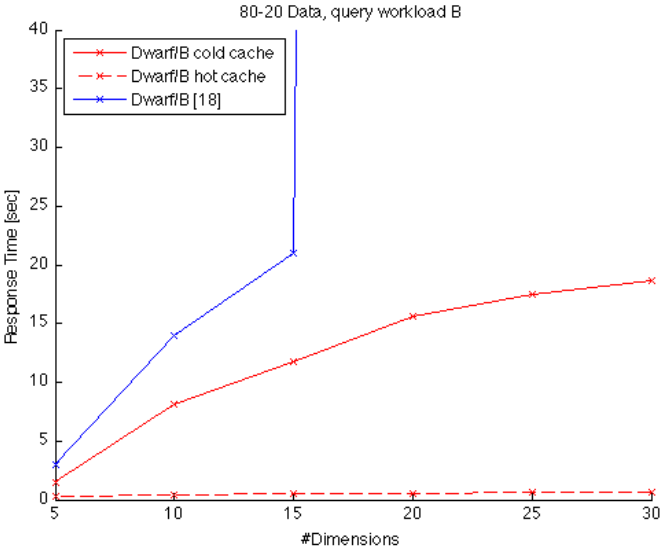


Figure 44: Response Times for 80-20 self similar Data on Queries Workload B vs [18]

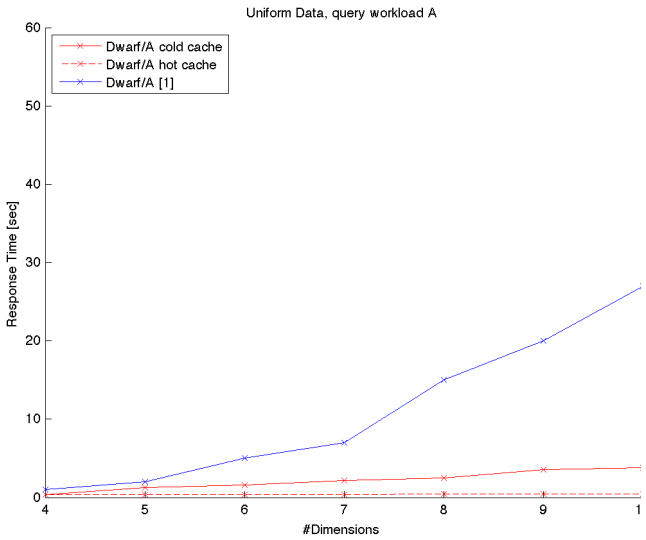


Figure 45: Response Times for Uniform Data on Queries Workload A vs [1]

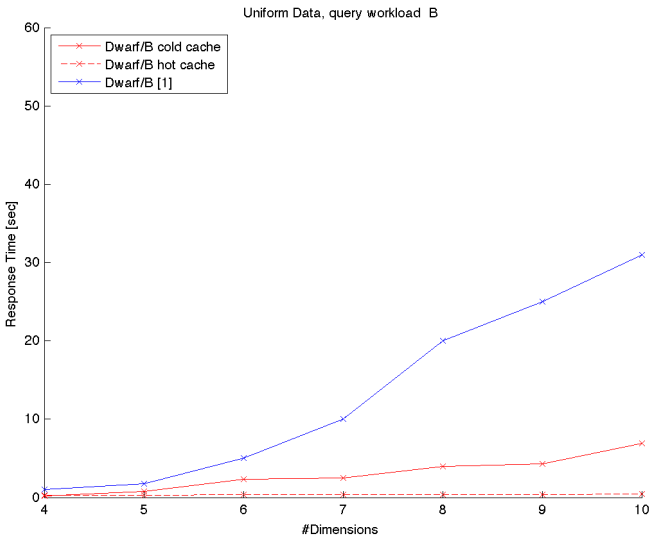


Figure 46: Response Times for Uniform Data on Queries Workload B vs [1]

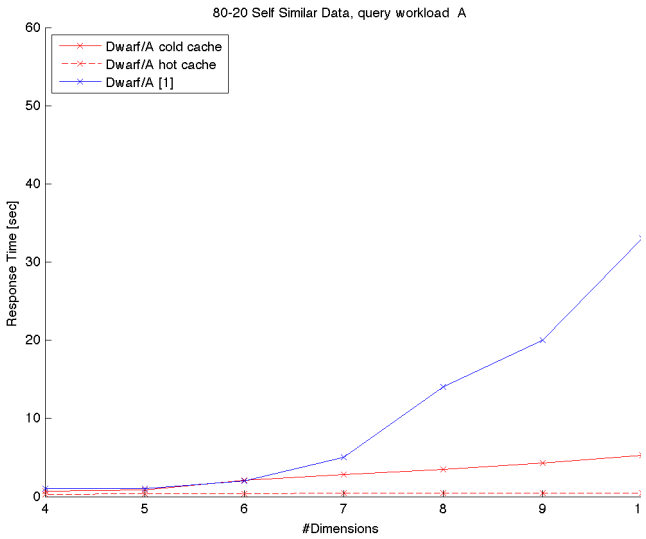


Figure 47: Response Times for 80-20 Self Similar Data on Queries Workload A vs [1]

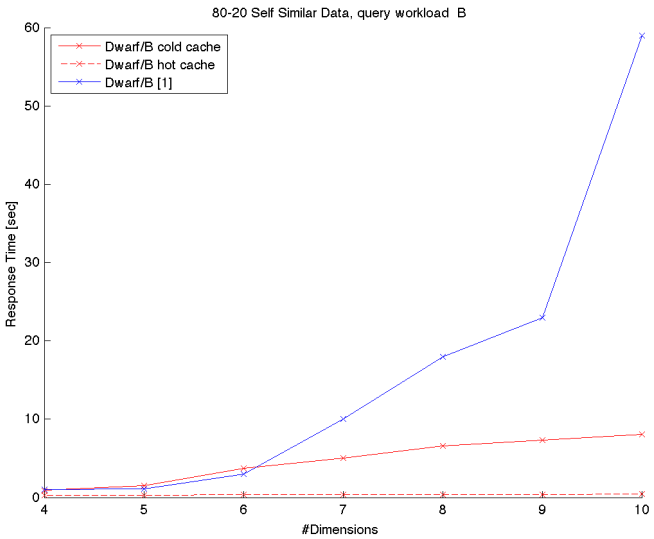


Figure 48: Response Times for 80-20 Self Similar Data on Queries Workload B vs [1]

Effect on Uniform Data: Figures 41, 42 show the results for uniform data to process workload A and B. The above results reveal that [18] was faster in query performance in contrast to our results. The authors do not mention if the results they demonstrate are based on "hot" or "cold" caching of the dwarf index. As a consequence, we decided to plot both query process times. The results show that Dwarf performs well: for workload A and 30 dimensions, it needs up to 12.5 seconds to process 1000 queries (while in [18] mention 4 seconds). For workload B it needs up to 14.6 seconds to process workload B ([18] mention 5.3 seconds). This is because workload B contains only new unrelated queries, while workload A contains roll-up and/or drill-down queries. Comparing our implementation to the original Dwarf algorithm [1], from figures 45, 46, and for dimensions range 4 to 10, our results much faster. For example, to process workload A [1] needs up to 27 seconds for 10 dimensions, while our tests process in 3.8 seconds, and to process workload B, up to 31 seconds against our 6.9 seconds of our implementation. In summary, our tests show a factor of 4 - 7 faster than [1], and (if the authors demonstrate cold cache times, we are by a factor of 3 slower than [18]).

Effect on Self-Similar Data: Figures 43, 44 show results for 80-20 self-similar data. Here, although for uniform data [18] report quite good performance runtimes, for 80-20, the performance characteristics are unpredictable. According to [18], their algorithm is not able to respond to queries for more than 15 dimensions. In our opinion, this is a huge disadvantage of [18] algorithm, which suggests that they might have not implemented correctly dwarf algorithm. They report that for more than 15 dimensions, the Dwarf index built for the 16 MB size fact table does not fit into the 5 GB available main memory anymore and has to perform considerable I/O. Our assumption is that they probably have not completely managed to implement a "correct"

memory-disk based dwarf as they have previously claimed. In general, our implementation has not any major memory requirements, as every I/O is memory-mapped buffered on disk. Although in previous experiments (Figure 7(a) from [18]), they report building dwarf index cube for 1,000,000 tuples over 80-20 self similar distribution, we do not clearly understand how it can be possible not to be able to answer queries from a dwarf index supporting 250,000 for more than 15 dimensions. For 15 dimensions and workload A, our algorithm takes up to 9.7 seconds against 13.2 of [18], and for workload B, our algorithm takes up to 11.8 against 21 seconds of [18]. Comparing with original Dwarf, for 10 dimensions our runtimes are 5.3 for workload A, and 8.1 for workload B, while authors in [1] report 33 and 59 seconds respectively (figures 47, 48). This is by a factor of 6 – 7 faster. In general, for self – similar data, we observe slightly slower performance due to the fact that the index dwarf cubes are larger, and the query utility needs more time to serve the workloads. Especially for workload B, performance is slower, again, due to the fact that each query is not related to the previous queried.

Chapter 7

Conclusions and Future Work

In this thesis, we focus on work related to Dwarf indexes. An in-depth discussion of the differences among the performance results reported by the inventors [1] and [18] is provided in Chapters 5 and 6. In general, our contribution was to achieve high performance build and query times both for dwarf index, as well as for the built based on specific views subsets. We tried to study the comportments of prefix and suffix coalescing, which are major factors that induce the size of the cube. Also, we created decision rules and additional algorithms for organizing the input views. During the conduction of this dissertation, we decided to encapsulate a set of rules in order to synchronize with our implementation of original Dwarf. Finally, we extended input possibilities, supporting text and sql-based input of any type of data (strings) through a mapping process. Our experiments for uniform and 80-20 self similar data indicate that the behavior of Dwarf index seem to be in sync with [1] and [18], while for skewed data our implementation outperformed [18] performance. Current results in Chapters 5 and 6, reveal an impressive achievement to maintain Dwarf expansion ratios and runtimes in low levels, while scaling the number of dimensions even up to 30.

Even in the area of constructing Dwarf for views, the size reduction is quite interesting, as soon as the users tend to query only a limited number of dimensions on each dataset. Future work would include more flexibility of efficiently sorting safe views. In cases of building more than the actual dwarf cube, future work would include a decision criteria based on the upper limit of not safe views vs the size of index to be constructed.

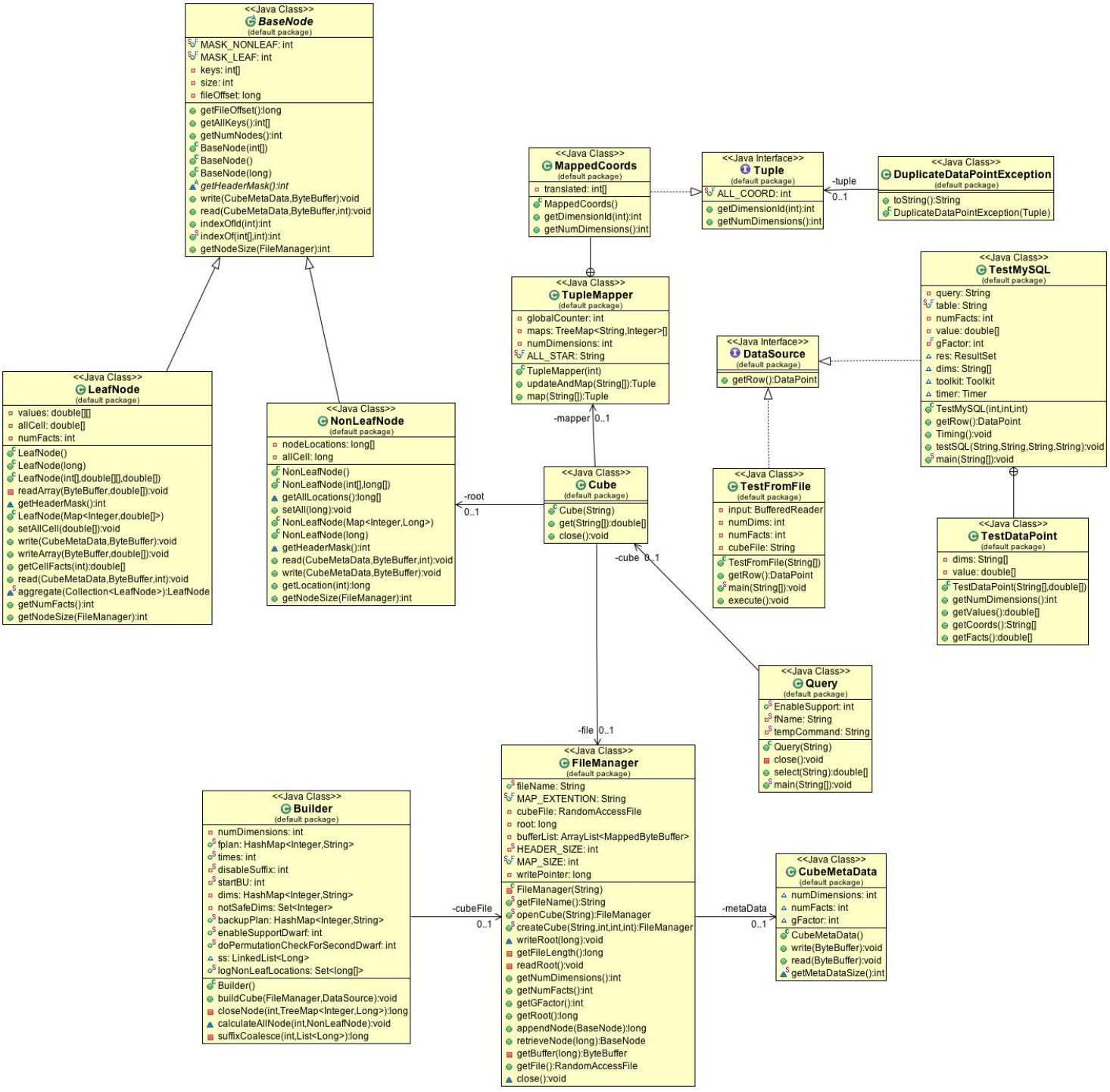
Further improvements would be to extend the possibilities of dwarf algorithm for supporting more aggregate functions. Also, we would like to support the GMIN parameter as described in 4.1 [1]. This could improve size reduction both for the original dwarf implementation, and *safeView* cube built, as in sparse cubes, storage-space and query performance are critical. Also, we could extend this work, by encapsulating extra decision rules. Authors mention that, during coalescing of nodes to build an {ALL∗} node, if it's size exceeded a size limit (larger than main memory), then they executed partial sorting from fact table, to avoid reading big chunk from disk. Using this, we could improve size reduction when building for dwarf for subset of views. Supporting range queries was beyond the limits of this thesis, and would be interesting to be supported. Also, a future work would include the ability to compute any dismissed {ALL∗} nodes when not existing during query time, by copying the relevant tree data on-the-fly from the fact table (lazy-updates).

As far as today's applications demand tremendous space and time benefits from OLAP applications, we believe that Dwarf can be extended to support an extensive scale of applications, as far as it is generated over a single pass over the data, and requires no deep knowledge of the value distributions. It offers 0% loss of data, providing both storage and indexing mechanism for high dimensional data. Also, a more recent analytical and algorithmic framework [2] derived the surprising result that the coalesced cube grows polynomially w.r.t the dimensionality. This result has changed the establish state that the cube is exponential on the number of dimensions and. In terms of updating performance, dwarf by far outperforms the closes competitor for storing the full data cube, and is good for periodic, large updates, as well as good for online 1-1 updates due to the fast built dimes we achieved.

In general, supporting specific view subset we extend the applicability of DWARF to a much wider area.

Appendix A

Class Diagram of Dwarf implementation



References

[1] Sismanis Yannis, Deligiannakis Antonios, Roussopoulos Nick, and Kotidis Yannis . Dwarf: Shrinking the PetaCube. In *ACM SIGMOD*, 2002.

[2] Sismanis Yannis and Roussopoulos Nick . "The Dwarf Data Cube Eliminates the high Dimensionality Curse." *University of Maryland* (TR-CS4552), 2003.

[3] Harinarayan, V, A. Rajaraman, and J. Ullman. "Implementing Data Cubes Efficiently." *In SIGMOD*, 1996.

[4] Rosen, Kenneth H. "Discrete Mathematics and Its Applications, 2nd edition." 282-284. McGraw-Hill, 1991.

[5] Xiaolei Li, Jiawei Han, Hector Gonzalez. "High-Dimensional OLAP: A Minimal Cubing Approach." *In VLDB*, Toronto, 2004.

[6] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD record*, 26:65-74, 1997.

[7] K. Beyer and R. Ramakrishnan. “Bottom-up computation of sparse and iceberg cubes.” *In SIGMOD*, 1999.

[8] J.han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measuers. *In SIGMOD*, 2001.

[9] D. Xin, J. Han, X. Li, and B. W. Wah. “Star-cubing: Computing iceberg cubes by top-down and bottom-up integration.” *In VLDB*, 2003.

[10] Y. Zhao, P. M. Deshpande, and J. F. Naughton. “An array-based algorithm for simultaneous multidimensional aggregates.” *In SIGMOD*, 1997

[11] j. Widom. “Research Prblems in Data Warehousing”. *In CIKM*, 1995

[12] Model 204. http://www.cca-int.com/prodinfo/m204.html

[13] C.D.French.“One Size Fits All “, Database Architectures Do Not Work For DSS. In *SIGMOD*, 1995.

[14] C. D. French. Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. In *ICDE*, 1997.

[15] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. “Data Cube: A relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *In ICDE*, 1996, New Orleans IEEE

[16] Sismanis John, "DWARF: A complete system for analyzing high-dimensional data sets", dep. Computer Science, Maryland, 2004

[17] J. Gray et al. Data Cube: A Relational aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *ICDE*, 1996.

[18] J. Dittrich, L. Blunschi, M.A. Vaz Salles, "Dwarfs in the Rearview Mirror: How Big are they Really?". *In VLDB*, 2008, ETH Zurich.

[19] C. Hahn, S. Warren, and J. London. Edited Synoptic Cloud Reports from Ships and Land Stations over the Globe.
http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html

[20] K. Morfonios, Y. Ioannidis, "Cure for Cubes: Cubing Using ROLAP Engine". *In VLDB*, 2006 Endowment, Dept. of Informatics and Telecom. Univ. of Athens

[21] X. Longgang, F. Yucai, "Fast Computation of Iceberg Dwarf". *In proceedings of the 16th International SSDBM*, 2004 Huazhong Univ. of Sci. and Technol. Wuhan, China