ABSTRACT

Title of dissertation: DWARF: A COMPLETE SYSTEM FOR ANALYZING HIGH-DIMENSIONAL DATA SETS

John (Yannis) Sismanis,
Doctor of Philosophy, 2004

Dissertation directed by: Professor Nick Roussopoulos
Department of Computer Science

The need for data analysis by different industries, including telecommunications, retail, manufacturing and financial services, has generated a flurry of research, highly sophisticated methods and commercial products. However, all of the current attempts are haunted by the so-called "high-dimensionality curse"; the complexity of space and time increases exponentially with the number of analysis "dimensions". This means that all existing approaches are limited only to coarse levels of analysis and/or to approximate answers with reduced precision. As the need for detailed analysis keeps increasing, along with the volume and the detail of the data that is stored, these approaches are very quickly rendered unusable. I have developed a unique method for efficiently performing analysis that is not affected by the high-dimensionality of data and scales only polynomially -and almost linearly- with the dimensions without sacrificing any accuracy in the returned results. I have implemented a complete system (called "Dwarf") and performed an extensive experimental evaluation that demonstrated tremendous improvements over existing methods

for all aspects of performing analysis -initial computation, storing, querying and updating it.

I have extended my research to the data-streaming model where updates are performed on-line, exacerbating any concurrent analysis but has a very high impact on applications like security, network management/monitoring router traffic control and sensor networks. I have devised streaming algorithms that provide complex statistics within user-specified relative-error bounds over a data stream. I introduced the class of "distinct implicated statistics", which is much more general than the established class of distinct count statistics. The latter has been proved invaluable in applications such as analyzing and monitoring the distinct count of species in a population or even in query optimization. The "distinct implicated statistics" class provides invaluable information about the correlations in the stream and is necessary for applications such as security. My algorithms are designed to use bounded amounts of memory and processing -so that they can even be implemented in hardware for resource-limited environments such as network-routers or sensors- and also to work in noisy environments, where some data may be flawed, either implicitly due to the extraction process, or explicitly.

DWARF: A COMPLETE SYSTEM FOR ANALYZING HIGH-DIMENSIONAL DATA SETS

by

John (Yannis) Sismanis

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2004

Advisory Committee:

Professor Nick Roussopoulos, Chair/Advisor
Professor Sudarshan Chawathe
Professor Lise Getoor
Professor Louiqa Raschid
Professor Virgil D. Gligor

DEDICATION

To my Family

ACKNOWLEDGMENTS

I want to express my gratitude to all those people who have made this thesis possible and made my graduate experience one that I will cherish forever.

First and foremost I'd like to thank my advisor, Professor Nick Roussopoulos for giving me an invaluable opportunity to work on challenging problems over the past six years. He believed very strongly in my work and showed amazing confidence in my abilities. He helped me concentrate my efforts and without him none of this work would have been possible. It has been a pleasure to work with and learn from such an extraordinary individual.

I would also like to thank the members of the committee, Professor Sudarshan Chawathe, Professor Lise Getoor, Professor Louiqa Raschid and Professor Virgil Gligor for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing the manuscript and providing comments for the thesis.

My talented colleagues and friends have enriched my graduate life in many ways and deserve a special mention. Yannis Kotidis donated invaluable support and helped me kick-off by providing the basics of my research, a platform for comparison and hours of discussions. In certain ways my work builds upon his foundational work. Alexandros Labrinidis is not only a talented colleague and a special friend but also a housemate for two years. Alex helped me in numerous ways and most importantly to organize not only my space and time more efficiently, but also to use

the world-wide-web in a way that I ever thought possible! Antonios Deligiannakis is not only an inspiring thinker but also a vivid developer. He contributed a lot of code for the human-computer interface of the system and he helped with the write-up of many reports and papers. Antonis and I helped each other during the graduate courses and our games of chess proved invaluable breaks during graduate school. Finally I want to thank Dimitrios Tsoumakos not only for the inspiring and insightful discussions we shared, but also for being a friend and helping me out at all times.

I was very fortunate to work in the environment of the Maryland Database group and interact with amazing people like Konstantinos Stathatos, Ugur Cetintemel, Björn T. Jónsson, Flip Korn, Manuel Rodriguez-Martinez, Tolga Urhan, Byoung-Kee Yi, Demet Aksoy, Mehmet Altinel and Fatma Ozcan. My interaction with them has been very fruitful and enjoyable.

I want to thank my homemates for providing a nice and easy going environment and nothing but good memories. First I must thank my long-time homemate Nikolas Stathatos and his parents who have been a crucial factor during my graduate studies. Nikolas is a special friend and helped me by taking the extra time to deal with any problem I had. I also thank Damianos Karakos for trying to be a perfect as possible homemate and for the interesting discussions about information theory and its applications.

I would like to acknowledge financial support from the Alexandros Onassis Public Benefit Foundation for some of the work discussed herein.

I must acknowledge Katerina Potika for her endless love and support through

all those years. I know how difficult it has been for us and the least I can do is be extremely grateful I'm lucky to have her by my side.

Finally, I owe my deepest thanks and gratitude to my family. They taught me to believe in myself and they offered me endless encouragement and support at all times. They've always stood by me and guided me through my career, pulling me through against impossible odds at times. This thesis is dedicated to them.

It is impossible to remember all, and I apologize to those I've inadvertently left out.

Thank you all!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

During the last decade, the need for data analysis by different industries, including telecommunications (call analysis, fraud detection), retail (user profiling, inventory management), manufacturing (customer support, order shipment) and financial services (risk and claims analysis), has generated a flurry of research, highly sophisticated methods and commercial products. However, all of the attempts are haunted by the so-called *"high-dimensionality curse"*; the complexity of space and time increases *exponentially* with the number of analysis "dimensions". In practice, this means that all existing approaches are limited only to coarse levels of analysis and/or to approximate answers with reduced precision. As the need for detailed analysis keeps increasing, along with the volume and the detail of the data that is stored, these approaches are very quickly rendered unusable. In my dissertation I have developed a unique method for efficiently performing analysis that is not affected by the "high-dimensionality" of data and scales only *polynomially* -and almost linearly- with the dimensions without sacrificing any accuracy in the returned results. I have implemented a complete system (called dwarf cube) and performed an extensive experimental evaluation that demonstrated tremendous improvements over existing methods for all aspects of performing analysis -initial computation, storing, querying and updating it. My method has already been used by the academic community

as a point of reference for similar "coalesced" methods of analysis.

1.1 Multidimensional Analysis

When performing analysis, data is typically modeled multidimensionally, where data attributes are divided into *dimensions* and *measures*. For example, when processing data in the telecommunication industry, the source phone number, the target phone number, and the time -when the phone call took place- define dimensions of interest and each specific call can be visualized as a point in this multidimensional space. Furthermore, the dimensions can be hierarchical; i.e. time can be organized in a second→minute→hour→ ... hierarchy and source/target phone number in a street→area→zip code hierarchy. Other attributes of phone calls like their durations or their costs constitute measures. Exploratory analysis requires efficient performance for aggregating the measure attributes over any possible combination (grouping) of the dimensions. For example, requesting the average cost of all phone calls per residential customer, or the maximum duration for each target phone number. The grouping may be at several hierarchy levels such as in the average duration grouped by the zip code of the source phone number and the week of the month. Undoubtedly the core for such analysis is the data cube operator which encapsulates all such possible groupings and provides the formulation for queries over categories, rollup/drilldown operations which allow the user to query from highly detailed groupings to less detailed or vice-versa, and perform cross tabulation. However the introduction of the cube operator and its expressive power comes at a big

cost. The number of groupings in the cube increases exponentially with the number of dimensions and the number of hierarchy levels per dimension. As a consequence, all attempts by previous researchers and commercial products are deemed unable to compute and store but small low-dimensional data cubes.

I developed a revolutionary storage system called the "dwarf cube". It is based on the discovery of and elimination of *prefix* and *suffix redundancies* in the groupings of data cubes. I introduced a specialized directed acyclic graph (called dwarf) to represent the cube, where all groupings are "overlapped" together. In dwarf, unique prefixes of the cube are stored only once through the prefix expansion operation. This operation is very effective in dense areas of the cube, where a lot of points share the same dimensions. However, *by far the most important operation is the suffix coalescing operation* that complements the prefix expansion and is orders of magnitudes more effective in sparse areas of the cube. A cube has areas that are totally empty or sparse, where there are no -or very few- points for the corresponding groupings. In addition, certain dimension values appear, most of the time, together in such a way that a dimensional value *implies* another. This implicit *sparsity of the cube* and the *implications between dimension values* generate a huge amount of "redundant" groupings, in the sense that another grouping provides the exact same aggregates to the cube. The suffix coalescing operation stores such redundant groupings just once, reducing by *orders of magnitudes* the storage requirements while maintaining the indexing for the redundant groupings.

The construction of the dwarf cube is done by two interleaved processes, one that does the *prefix expansion* and one that does *suffix coalescing*. These two pro-

cesses are performed in just a *single pass* over the data. Equally, or even more important, than the discovery and elimination of prefix and suffix redundancies, is that the interleaving of prefix expansion and suffix coalescing has the unique property of identifying these redundancies before computing any aggregates in the redundant areas of the data cube. This avoidance of recomputation of redundant aggregates results in a remarkable savings in computation time. As an example of such savings of the dwarf cube computation, a cube of twenty-five dimensions, that needed a petabyte of storage when stored in a conventional data cube, reduces to a dwarf cube of only a couple of gigabytes in under 20 minutes of time. This is for a full data cube in which every unique grouping was computed and stored with no loss in precision.

The analysis of the interleaved process revealed the surprising result that, even if one considers only the sparsity of the cube, *the complexity of the dwarf storage is polynomial to the number of dimensions*, unlike all previous methods that scale exponentially. This striking result reformulates the context of cube management and extends its applicability into a much wider area of applications than ever before. In real data, the complexity of my approach was even smaller -almost linear- due to the implications between dimensions. With respect to the volume of the data, my analysis proves that the dwarf storage scales again polynomially and very close to linear (the polynomial power is very close to 1).

Establishing the theory is one thing, and actually developing a complete system and applying it to real applications is another. I am the main author and maintainer of the dwarf storage that was developed from scratch for this purpose

and is the current state-of-art tool for analyzing huge and highly complex data. During the development and experimentation of the dwarf system, I discovered a rather counter-intuitive fact. Aggregates have a wide variety of computation cost and therefore, it is much more efficient to aggregate and store those groupings that are "hard" to compute, and postpone the "easy" groupings to be performed on the fly during request time. This kind of *partial materialization* benefits not only the initial computation, but also the overall response performance because of the big gap between raw computation power and fetching data from secondary memory. For example, when a data cube query asks for specific groupings, it is much more efficient to perform the aggregation on the fly using either other groupings or raw data than fetching precomputed aggregates from secondary memory. For that reason, I introduced the *granularity parameter* for sparse areas, and the *knob materialization* for dense areas. Both approaches control the amount of materialization by avoiding storing and handling groupings that are "easy" to perform during request time. The granularity parameter and knob materialization control how much initial computation and storage are saved. An extensive experimental evaluation for the granularity and knob parameter is covered in the thesis. This demonstrated that although a large number of groupings are not computed nor stored, the response performance increases four times, while the computation/storage requirements are reduced ten-fold with comparison to fully materialized dwarf cubes.

In addition, the effect of the knob materialization is so profound that, in one of our experiments with a real data set given to us by a principal database management system vendor, the query performance increased two times, while the storage

requirements dropped eight times and the computation time dropped six times with respect to the partial materialized dwarf cubes with only the granularity approach -which are already two times faster and smaller than fully materialized dwarf cubes-. Due to our non-disclosure agreement I can only reveal that the principal vendor used a partial materialization technique that needed over three days just to select what to materialize, while my knob/granularity approach required about 15 minutes to compute/store and index the corresponding cube on the same hardware achieving sub-second performance for analytical requests.

1.2 Streaming Analysis

My research is not limited to the "data-warehousing" model, where data from different operational databases, perhaps over very large periods of time, are consolidated in a single place resulting in historical and summarized data that is orders of magnitudes larger than the source operational databases. In this model, analysis is performed in an off-line mode and updates are performed in batches, when no analysis can take place. I have extended my research to the "data-streaming" model where updates are performed on-line, exacerbating any concurrent analysis but has a very high impact on applications like security (intrusion detection, denial of service attacks), network management/monitoring (network and user statistics) router traffic control (pricing, alternate queuing, route selection) and sensor networks. The data stream is potentially unbounded in size and one must assume that neither the memory, required to store the whole stream, nor the processing power, required to

keep up in pace with the stream, is available.

I have devised streaming algorithms that provide complex statistics within user-specified relative-error bounds over a data stream. I introduced the class of "distinct implicated statistics", which is much more general than the established class of "distinct count" statistics. The latter has been proved invaluable in applications such as analyzing and monitoring the distinct count of species in a population -like the distinct number of sources seen at any moment by a network router- or even in query optimization -selecting a good query plan-. The "distinct implicated statistics" class provides invaluable information about the correlations in the stream and is necessary for applications such as security. For example, an online intrusion detection system, running on a network router, could be interested in the "number of distinct destinations that 90% of the time are contacted by at least ten different sources for the FTP service over a sliding window of 1h". Such statistics capture interesting correlations in an online fashion and can be used as a monitoring or analysis tool and provide the framework for triggering mechanisms. My algorithms are designed to use bounded amounts of memory and processing -so that they can even be implemented in hardware for resource-limited environments such as network-routers or sensors- and also to work in "noisy" environments, where some data may be flawed either implicitly due to the extraction process (i.e. acceptable errors in sensors) or explicitly (i.e. due to an attack in a secured environment).

Chapter 2

Preliminaries & Contributions

2.1 Preliminaries

2.1.1 Computing Data Cubes

The data cube operator [GBLP96] performs the computation of one or more aggregate functions for all possible combinations of grouping attributes (called *views*). The inherent difficulty with the cube operator is its size, both for computing and storing it. The number of all possible group-bys increases exponentially with the number of the cube's dimensions and a naive store of the cube behaves in a similar way. The authors of [GBLP96] provided some useful hints for cube computation including the use of parallelism, and mapping string dimension types to integers for reducing the storage. The problem is exacerbated by the fact that new applications include an increasing number of dimensions and, thus, the explosion on the size of the cube is a real problem. All methods proposed in the literature try to deal with the space problem, either by precomputing a subset of the possible group-bys [HRU96a, GHRU97a, Gup97, BPT97a, SDN98], by estimating the values of the group-bys using approximation [GM98, VWI98, SFB99, AGP00] or by using online aggregation [HHW97] techniques or finally by pre-computing only those group-bys

that have a minimum support (membership) of at least *minimum support* C_{min} tuples [BR99]. The larger the value for C_{min}, the smaller the number of group-bys that satisfy the minimum support condition and, therefore, the smaller the size of the resulting sub-cubes.

2.1.2 Storing/Indexing/Updating Data Cubes

Although computing is a major aspect of the problem of computing data cubes, of equal importance is the problem of efficiently storing, indexing, querying and updating the precomputed aggregates. The "easiest" way is to store each view as an independent relation in a relational database system. Such systems are called Relational OLAP (ROLAP) and the major benefit is that they are build upon well-known and understood technology. However the major drawback is not only the need of extra indexes for efficiently querying the views, but the tremendous amount of space required to store all the views of high-dimensional data sets. Multidimensional-OLAP (MOLAP) tries to address the issue of indexing by using multidimensional array storage techniques that provide an implicit indexing mechanism. Still however these systems suffer from the "dimensionality curse" and cannot scale to high-dimensional datasets.

2.2 Contributions — The Dwarf Solution

To address these issues, we propose Dwarf, a highly compressed structure[1] for computing, storing, and querying data cubes. Dwarf solves the storage space problem, by identifying prefix and suffix redundancies in the structure of the cube and factoring them out of the store.

Prefix redundancy can be easily understood by considering a sample cube with dimensions a, b and c. Each value of dimension a appears in 4 group-bys (a, ab, ac, abc), and possibly many times in each group-by. For example, for the fact table shown in Table 2.1 (to which we will keep referring throughout this chapter) the value $S1$ will appear a total of 7 times in the corresponding cube, and more specifically in the group-bys: $\langle S1, C2, P2\rangle$, $\langle S1, C3, P1\rangle$, $\langle S1, C2\rangle$, $\langle S1, C3\rangle$, $\langle S1, P2\rangle$, $\langle S1, P1\rangle$ and $\langle S1\rangle$. The same also happens with prefixes of size greater than one -note that each pair of a, b values will appear not only in the ab group-by, but also in the abc group-by. Dwarf recognizes this kind of redundancy, and stores every unique prefix just once.

Store	Customer	Product	Price
S1	C2	P2	\$70
S1	C3	P1	\$40
S2	C1	P1	\$90
S2	C1	P2	\$50

Table 2.1: Fact Table for cube Sales

Suffix redundancy occurs when two or more group-bys share a common suffix (like abc and bc). For example, consider a value b_j of dimension b that appears

[1]The name comes after Dwarf stars that have a very large condensed mass, but occupy very small space. They are so dense, that their mass is about *one* ton/cm^3

in the fact table with a single value a_i of dimension a (such an example exists in Table 2.1, where the value $C1$ appears with only the value $S2$). Then, the group-bys $\langle a_i, b_j, x\rangle$ and $\langle b_j, x\rangle$ always have the same value, for any value x of dimension c. This happens because the second group-by aggregates all the tuples of the fact table that contain the combinations of any value of the a dimension (which here is just the value a_i) with b_j and x. Since x is generally a set of values, this suffix redundancy has a multiplicative effect. Suffix redundancies are even more apparent in cases of correlated dimension values. Such correlations are often in real datasets, like the Weather dataset used in one of our experiments. Suffix redundancy is identified during the construction of the Dwarf cube and eliminated by *coalescing* their space.

What makes Dwarf practical is the automatic discovery of the prefix and suffix redundancies without requiring knowledge of the value distributions and without having to use sophisticated sampling techniques to figure them out. The Dwarf storage savings are spectacular for both dense and sparse cubes. We show that in most cases of very dense cubes, the size of the Dwarf cube is much less than the size of the fact table. However, while for dense cubes the savings are almost entirely from prefix redundancies, as the cubes get sparser, the savings from the suffix redundancy elimination increases, and quickly becomes the dominant factor of the total savings.

Equally, or even more, significant is the reduction of the computation cost. Each redundant suffix is identified *prior* to its computation, which results in substantial computational savings during creation. Furthermore, because of the condensed size of the Dwarf cube, the time needed to query and update is also reduced. Inherently, the Dwarf structure provides an index mechanism and needs no addi-

tional indexing for querying it. It is also self-sufficient in the sense that it does not need to access or reference the fact table in answering any of the views stored in it.

An additional optimization that we have implemented is to avoid precomputation of certain group-bys that can be calculated on-the-fly by using fewer than a given constant amount of tuples. The information needed to calculate these group-bys is stored inside the Dwarf structure in a very compact and clustered way. By modifying the value of the above constant, the user is able to trade query performance for storage space and creation time. This optimization was motivated from iceberg cubes [BR99] and may be enabled by the user if a very limited amount of disk space and/or limited time for computing the Dwarf is available.

2.3 Contributions — Dwarf Polynomial Complexity

The whole problem of managing data cubes is further complicated by the fact that since precomputation may result in an nonintuitive large increase in the amount of storage required by the database, the database administrator would like an accurate and fast estimate of the required storage *without* actually performing the aggregations. The Dwarf solution, by factoring out structural redundancies exacerbates the problem rendering previous approaches (i.e. [SDNR96]) inapplicable. We investigate the problem of efficiently and accurately estimating the size of the Dwarf.

The complexity analysis for the time and space requirements of the Dwarf construction algorithm shows that *unlike all previous methods* Dwarf scales polyno-

mially with respect to the dimensionality and therefore the problem of cube management is not inherently exponential in nature. Additionally we show that real datasets scale even better because of theirs implicit correlations. Finally, we were able to devise an efficient algorithm for estimating the size of the Dwarf before actually computing it.

2.4 Contributions — Implication Counts

The spectacular savings achieved by the Dwarf for managing data cubes, naturally leads to the question of defining a metric that encapsulates the "volume" of structural redundancies of the data cube. We propose the use of *Implication Counts* as a set of sufficient aggregates that capture that information. For a data set logically divided into two sets of attributes `A` and `B`, we define as the implication count C_a of itemsets a_i of `A`, the number of a_i that *imply* some itemsets of `B`. For example, in the fact table in Table 2.1 we observe that the group-bys $\langle S2, C1, x\rangle$ and $\langle C1, x\rangle$, where x is any value of the dimension `Product`, have always the same aggregate values. This happens because the $\langle C1, x\rangle$ group-by aggregates all the tuples of customer $C1$ for any combination of stores. However, in this case, there is only one store ($S2$) and the aggregation "degenerates" to the aggregation already performed for the group-by $\langle S2, C1, x\rangle$. In this sense, customer $C1$ *implies* store $S2$ and the count of such implications directly translates to savings due to suffix redundancy.

We have addressed the problem of efficiently estimating such implications counts and our contribution can be summarized as follows:

1. We describe a generalization of implication aggregate queries that frequently arise in the data stream model of data processing and in many other fields of database research.

2. We provide memory and processing efficient algorithms for estimating such aggregates, within small error bounds (typically less than 10% relative error).

3. We prove that the complement problem of estimating non-implication counts can be (ϵ, δ)-approximated under most conditions.

4. We demonstrate the accuracy of our methods, through an extensive set of experiments on both synthetic and real datasets.

Chapter 3

The Dwarf Cube

3.1 Introduction

In the following we define Dwarf, a highly compressed data structure for managing (computing, storing, indexing, updating) data cubes. To demonstrate the storage savings provided by Dwarf (and what fraction of the savings can be attributed to prefix and suffix redundancies), we first compare the Dwarf cube sizes against a binary storage footprint (BSF), i.e. as if all the views of the cube were stored in unindexed binary summary tables. Although this is not an efficient (or sometimes feasible) store for a cube or sub-cubes, it provides a well understood point of reference and it is useful when comparing different stores.

We also compared the Dwarf cubes with *Cubetrees* which were shown in [RKR97, KR98] to exhibit at least a 10:1 better query response time, a 100:1 better update performance and 1:2 the storage of indexed relations. Our experiments show that Dwarfs consistently outperform the Cubetrees on all counts: storage space, creation time, query response time, and updates of full cubes. Dwarf cubes achieve comparable update performance on partial cubes stored on Cubetrees having the same size with Dwarf cubes. However, byte per byte, Dwarf stores many more materialized views than the corresponding Cubetree structures and, therefore, can

answer a much wider class of queries for the same footprint.

We used several data sets to compute Dwarf cubes. One of them was a cube of 20 dimensions, each having a cardinality of 1000, and a fact table containing 100000 tuples. The BSF for its cube is 4.4TB.[1] Eliminating the prefix redundancy, resulted in a Dwarf cube of 1.4 TB (31.8% of the original size). Eliminating the suffix redundancy reduced the size of the Dwarf cube to just 300 MB[2], a 1:14666 reduction over BSF. Following Jim Gray's spirit of pushing every idea to its limits, we decided to create a Petacube of 25-dimensions with BSF equal to one Petabyte. The Dwarf cube for the Petacube is just 2.3 GB and took less than 20 minutes to create. This is a 1:400000 storage reduction ratio.

3.2 Formal Dwarf Description

We first describe the Dwarf structure with an example. Then we define the properties of Dwarf formally.

3.2.1 A Dwarf example

Figure 3.1 shows the Dwarf Cube for the fact table shown in Table 2.1 . It is a full cube using the aggregate function *sum*. The nodes are numbered according to the order of their creation. The height of the Dwarf is equal to the number of dimensions, each of which is mapped onto one of the levels shown in the figure.

[1] The BSF sizes, and the size of Dwarf cubes without enabling suffix coalescing were accurately measured by first constructing the Dwarf cube, and then traversing it appropriately.

[2] All the sizes of Dwarf cubes, unless stated otherwise, correspond to the full Dwarf cubes

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. The node pointed by a cell and all the cells inside it are *dominated* by the cell. For example the cell $S1$ of the root dominates the node containing the keys $C2, C3$. Each non-leaf node has a special ALL cell, shown as a small gray area to the right of the node, holding a pointer and corresponding to all the values of the node.

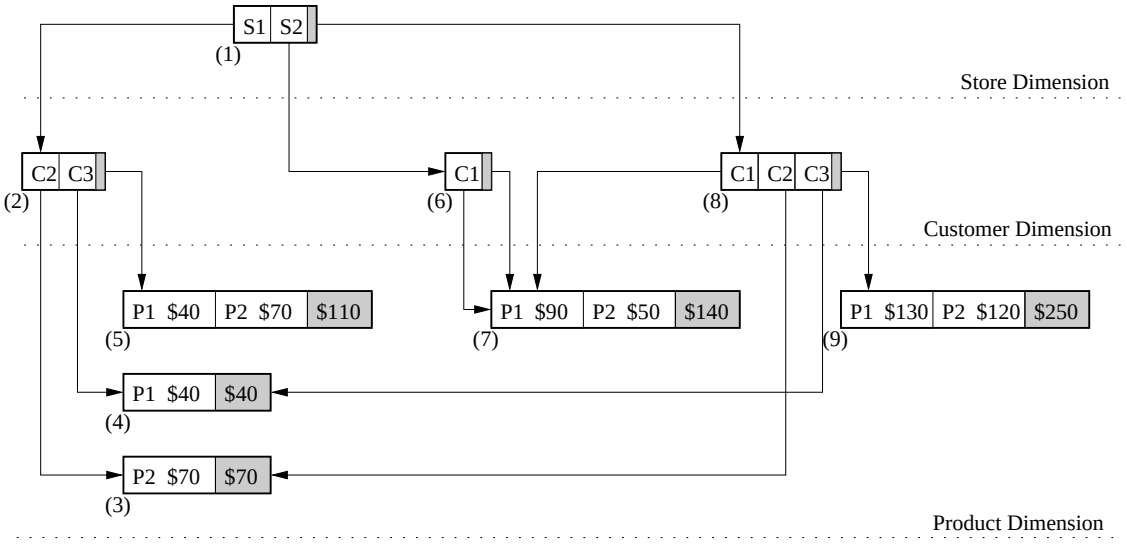


Figure 3.1: The Dwarf Cube for Table 2.1

A path from the root to a leaf such as $\langle S1, C3, P1\rangle$ corresponds to an instance of the group-by Store, Customer, Product and leads to a cell $[P1\ \$40]$ which stores the aggregate of that instance. Some of the path cells can be open using the ALL cell. For example, $\langle S2, ALL, P2\rangle$ leads to the cell $[P2\ \$50]$, and corresponds to the sum of the Prices paid by any Customer for Product P2 at Store S2. At the leaf level, each cell is of the form [key, aggregate] and holds the aggregate of all tuples that match a path from the root to it. Each leaf node also has an ALL cell that stores the

aggregates for all the cells in the entire node. $\langle ALL, ALL, ALL \rangle$ leads to the total Prices (group-by NONE). The reader can observe that the three paths $\langle S2, C1, P2 \rangle$, $\langle S2, ALL, P2 \rangle$, and $\langle ALL, C1, P2 \rangle$, whose values are extracted from processing just the last tuple of the fact-table, all lead to the same cell [$P2$ \$50], which, if stored in different nodes, would introduce *suffix redundancies*. By *coalescing* these nodes, we avoid such redundancies. In Figure 3.1 all nodes pointed by more than one pointer are coalesced nodes.

3.2.2 Properties of Dwarf

In Dwarf, like previous algorithms proposed for cube computation, we require the dimension attributes to be of integer type (thus mapping other types, like strings, to integers in required) but, unlike other methods, we do not require packing the domain of values between zero and the cardinality of the dimension. Any group-by of a D-dimensional cube can be expressed by a sequence of D values (one for each dimension), to which we will refer as the *coordinates* of the group-by in a multidimensional space. In SQL queries, the coordinates are typically specified in the WHERE clause. The group-by's j-th coordinate can either be a value of the cube's j-th dimension, or left open to correspond to the ALL pseudo-value.

The Dwarf data structure has the following properties:

1. It is a directed acyclic graph (DAG) with just one root node and has exactly D levels, where D is the number of cube's dimensions.

2. Nodes at the D-th level (*leaf nodes*) contain cells of the form: $[key,\ aggrValues]$.

3. Nodes in levels other that the D-th level (*non-leaf nodes*) contain cells of the form: [*key*, *pointer*]. A cell C in a 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 aggrValues 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 paths from the root, all of which will share a common suffix. For example, in the node at the bottom of the Product level of Figure 3.1, the first cell of

the node corresponds to the sequences $\langle S1, C2, P2\rangle$ and $\langle ALL, C2, P2\rangle$, which share the common suffix $\langle C2, P2\rangle$. If a node N is a coalesced node, then any node X which is a descendant of N will also be a coalesced node, since it can be reached from multiple paths from the root.

A traversal in the Dwarf structure always follows a path of length D, starting from the root to a leaf node. It has the form $\langle [Node_1.val|ALL]$, $[Node_2.val|ALL]$, ... , $[Node_D.val|ALL]\rangle$, meaning that the i-th key found in the path will either be a value $Node_i.val$ of the i-th dimension, or the pseudo-value ALL. The Dwarf structure itself constitutes an efficient interlevel indexing method and requires no additional external indexing.

We now define some terms which will help in the description of the algorithms. The *dwarf of a node* N is defined to be the node itself and all the dwarfs of the nodes dominated by the cells of N. The dwarf of a node X that is dominated by some cell of N is called a *sub-dwarf* of N. Since leaf node cells dominate no other nodes, the dwarf of a leaf node is the node itself. The number of cells in the node N_j, which a cell C_i dominates, is called the *branching factor* of C_i.

A sequence of i keys, followed in any path from the root to a node N at level $i+1$ of the Dwarf structure, is called the *leading prefix* of N. A *leading prefix* of N, which contains no coordinate with ALL, is called the *primary leading prefix* of N.

The *content* of a cell C_i, belonging to a node N, is either the aggrValues of C_i if N is a leaf node, or the sub-dwarf of C_i if N is a non-leaf node.

3.2.3 Evidence of Structural Redundancy

Prefix Redundancy

A path from the root of the Dwarf structure to a leaf, corresponds to an instance of some group-by. Dwarf creates the minimum number of cells to accommodate all paths. In the cube presented in Figure 3.1, for the first level of the structure (Store), the maximum number of cells required is equal to the cardinality of the Store dimension $Card_{store}$ plus 1 (for the ALL cell).

For the second level (Customer), if the cube was completely dense, we would need a number of cells equal to the product: $(Card_{store} + 1) \times (Card_{customer} + 1)$. Since most cubes are sparse, there is no need to create so many cells.

However, even in the case of dense cubes, the storage required to hold all cells of the structure (including the ALL cells) is comparable to that required to hold the fact table. A Dwarf for a saturated cube of D dimensions and the same cardinality N for each dimension, is actually a tree with a constant branching factor equal to: $bf = N + 1$. Therefore, the number of leaf nodes and non-leaf nodes required to represent this tree is:

$$nonLeafNodes = \frac{(N+1)^{D-1} - 1}{N}\ ,\ \ LeafNodes = (N+1)^{D-1} \qquad (3.1)$$

Each non-leaf node contains N non-leaf cells and one pointer and each leaf node contains N leaf cells and the aggregates. The size of a non-leaf cell is two units (one for the key and one for the pointer), while the size of a leaf-cell is $A+1$ (A units for

the aggregates and one for the key). The fact table of the saturated cube has N^D tuples. The size for each tuple is $D + A$. The ratio of the size of the Dwarf over the fact table is then approximated[3] by:

$$ratio \approx \frac{A(N+1)^D + N(N+1)^{D-1}}{(D+A)N^D} \tag{3.2}$$

For example for a full dense cube with $D = 10$ dimension, a cardinality of $N = 1000$ for each dimension, and one aggregate ($A = 1$), we have a ratio of: 0.18, i.e. the Dwarf representation needs less than 20% of the storage that the fact table requires. This proves that the fact table itself (and, therefore, certainly the cube) contains redundancy in its structure.

The above discussion serves to demonstrate that Dwarf provides space savings even in the case of very sparse cubes. Of course, for such a case a MOLAP representation of the cube would provide a larger cube compression. However, MOLAP methods for storing the cube require knowledge (or the discovery) of the dense areas of the cube, and do not perform well for sparse, high-dimensional cubes. On the other hand, Dwarf provides an automatic method for highly compressing the cube independently of the characteristics (distribution, density, dimensionality...) of the data.

[3]The size of all the leaf nodes is much larger than the size of all the non-leaf nodes

Suffix Redundancy

Since Dwarf does not store cells that correspond to empty regions of the cube, each node contains at least one cell with a key value, plus the pointer of the ALL cell. Therefore, the minimum branching factor is 2, while the maximum value of the branching factor of a cell at level j is $1+Card_{j+1}$, where $Card_{j+1}$ is the cardinality of dimension $j+1$. The branching factor decreases as we descend to lower levels of the structure. An approximation of the branching factor at level j of the structure, assuming uniform distribution for the values of each dimension for the tuples in the fact table, is:

$$branch(j) = 1 + \min\left(Card_{j+1},\ \max\left(1,\ T/\prod_{i=1}^{j} Card_i\right)\right) \tag{3.3}$$

where T is the number of tuples in the fact table. If the cube is not very dense, the branching factor will become equal to 2 at the k-th level, where k is the lowest number such that $T/\prod_{i=1}^{k} Card_i \leq 1$. For example, for a sparse cube with the same cardinality $N = 1000$ for all dimensions, $D = 10$ dimensions and $T = 10000000 (\ll N^D)$ tuples, the branching factor will reach the value 2 at level $k = \lceil \log_N T \rceil = 3$. This means that in very sparse cubes, the branching factor close to the root levels deteriorates to 2. A branching factor of 2 guarantees (as we will see in Section 3.2) that suffix redundancy exists at this level. Therefore, the smaller the value of k, the larger the benefits from eliminating suffix redundancy, since the storage of larger dwarfs is avoided.

Correlated areas of the fact table can also be coalesced. Assume for example,

that a set of certain customers C_s shop *only* at a specific store S. The views $\langle Store, Customer, ...\rangle$ and $\langle ALL, Customer, ...\rangle$ share the suffix that corresponds to the set C_s. In Table 2.1, customers $C2$ and $C3$ shop only at store $S1$ and in Figure 3.1 we see that the nodes 3 and 4 of the dwarf of node 2 are also coalesced from node 8.

3.3 Constructing the Dwarf Cube

The Dwarf construction is governed by two processes: the *prefix expansion*, and the *suffix coalescing*. A non-interleaved two-pass process would first construct a cube with the prefix redundancy eliminated, and then check in it for nodes that can be coalesced. However, such an approach would require an enormous amount of temporary space and time, due to the size of the intermediate cube. It is thus imperative to be able to determine when a node can be coalesced with another node before actually creating it. By imposing a certain order in the creation of the nodes, suffix coalescing and prefix expansion can be performed at the same time, without requiring two passes over the structure.

Before we present the algorithm for constructing the Dwarf cube, we present some terms that will be frequently used in the algorithm's description. A node N_{ans} is called an *ancestor* of N iff N is a descendant node of N_{ans}. During the construction of the Dwarf Cube, a node N at level j of the Dwarf structure is *closed* if there does not exist an unprocessed tuple of the fact-table that contains a prefix equal to the primary leading prefix of N. An existing node of the Dwarf structure

which is not *closed* is considered *open*.

The construction of a Dwarf cube is preceded by a single sort on the fact table using one of the cube's dimensions as the primary key, and collating the other dimensions in a specific order. The choice of the dimensions' ordering has an effect on the total size of the Dwarf Cube. Dimensions with higher cardinalities are more beneficial if they are placed on the higher levels of the Dwarf cube. This will cause the branching factor to decrease faster, and coalescing will happen in higher levels of the structure. The ordering used will either be the one given by the user (if one has been specified), or will be automatically chosen by Dwarf after performing a scan on a sample of the fact table and collecting statistics on the cardinalities of the dimensions.

3.3.1 Prefix Expansion algorithm

The Dwarf construction algorithm *PrefixExpansion* is presented in Algorithm 1. The construction requires just a single sequential scan over the sorted fact table. For the first tuple of the fact table, the corresponding nodes and cells are created on all levels of the Dwarf structure. As the scan continues, tuples with common prefixes with the last tuple will be read. We create the necessary cells to accommodate new key values as we progress through the fact table. At each step of the algorithm, the common prefix P of the current and the previous tuple is computed. Consider the path we need to follow to store the aggregates of the current tuple. The first $|P|+1$ nodes (where $|P|$ is the size of the common prefix) of the path up to a node N have

already been created because of the previous tuple. Thus, for a D-dimensional cube, $D - |P| - 1$ new nodes need to be created by expanding the structure downwards from node N (and thus the name *Prefix Expansion*), and an equal number of nodes have now become closed. When a leaf node is closed, the ALL cell is produced by aggregating the contents (aggregate values) of the other cells in the node. When a non-leaf node is closed, the ALL cell is created and the SuffixCoalesce algorithm is called to create the sub-dwarf for this cell.

Algorithm 1 PrefixExpansion Algorithm

Input: sorted fact table, D : number of dimensions

1: Create all nodes and cells for the first tuple
2: last_tuple = first tuple of fact table
3: **while** more tuples exist unprocessed **do**
4: current_tuple = extract next tuple from sorted fact table
5: P = common prefix of current_tuple , last_tuple
6: **if** new closed nodes exist **then**
7: write special cell for the leaf node homeNode where last_tuple was stored
8: 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
9: **end if**
10: Create necessary nodes and cells for current_tuple { $D - |P| - 1$ new nodes created}
11: last_tuple = current_tuple
12: **end while**
13: write special cell for the leaf node homeNode where last_tuple was stored
14: 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)

For example consider the fact table of Table 2.1 and the corresponding Dwarf cube of Figure 3.1. The nodes in the figure are numbered according to the order

of their creation. The first tuple $\langle S1, C2, P2\rangle$ creates three nodes (Nodes 1, 2 and 3) for the three dimensions (Store, Customer and Product) and inserts one cell to each node. Then the second tuple $\langle S1, C3, P1\rangle$ is read, which shares only the prefix $S1$ with the previous tuple. This means that cell $C3$ needs to be inserted to the same node as $C2$ (Node 2) and that the node containing $P2$ (Node 3) is now *closed*. The ALL cell for Node 3 is now created (the aggregation here is trivial, since only one other cell exists in the node). The third tuple $\langle S2, C1, P1\rangle$ is then read and contains no common prefix with the second tuple. Finally, we create the ALL cell for Node 4 and call SuffixCoalesce for Node 2 to create the sub-dwarf of the node's ALL cell.

3.3.2 Suffix Coalesce algorithm

Suffix Coalesce creates the sub-dwarfs for the ALL *cell* of a node. Suffix Coalesce tries to identify *identical* dwarfs and coalesce their storage. Two, or more, dwarfs are *identical* if they are constructed by the same subset of the fact table's tuples. Prefix expansion would create a tree if it were not for Suffix Coalescing.

The SuffixCoalesce algorithm is presented in Algorithm 2. It requires as input a set of Dwarfs (*inputDwarfs*) and merges them to construct the resulting dwarf. The algorithm makes use of the helping function calculateAggregate, which aggregates the values passed as its parameter.

SuffixCoalesce is a recursive algorithm that tries to detect at each stage whether

Algorithm 2 SuffixCoalesce Algorithm

Input: inputDwarfs = set of Dwarfs

```
1: if only one dwarf in inputDwarfs then
2:    return dwarf in inputDwarfs {coalescing happens here}
3: end if
4: while unprocessed cells exist in the top nodes of inputDwarfs do
5:    find unprocessed key Key_min with minimum value in the top nodes of input-
      Dwarfs
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:       write cell [Key_min  calculateAggregate(toMerge.aggregateValues)]
9:    else
10:      write cell [Key_min  SuffixCoalesce(toMerge.sub-dwarfs)]
11:   end if
12: end while
13: create the ALL cell for this node either by aggregation or by calling SuffixCoa-
    lesce
14: return position in disk where resulting dwarf starts
```

some sub-dwarf of the resulting dwarf can be coalesced with some sub-dwarf of *inputDwarfs*. If there is just one dwarf in *inputDwarfs*, then coalescing happens immediately, since the result of merging one dwarf will obviously be the dwarf itself. The algorithm then repeatedly locates the cells *toMerge* in the top nodes of inputDwarfs with the smallest key Key_{min} which has not been processed yet. A cell in the resulting dwarf with the same key Key_{min} needs to be created, and its *content* (sub-dwarf or aggregateValues) will be produced by merging the *contents* of all the cells in the *toMerge* set. There are two cases:

1. If we are at a leaf node we call the function calculateAggregate to produce the aggregate values for the resulting cell.

2. Otherwise, coalescing cannot happen at this level. We call SuffixCoalesce recursively to create the dwarf of the current cell, and check if parts of the

structure can be coalesced at one level lower.

At the end, the ALL cell for the resulting node is created, either by aggregating the values of the node's cells (if this is a leaf node) or by calling SuffixCoallesce, with the sub-dwarfs of the node's cells as input.

As an example, consider again the Dwarf cube presented in Figure 3.1. We will move to the step of the algorithm after all the tuples of Table 2.1 have been processed, and the ALL cell for Node 7 has been calculated. SuffixCoalesce is called to create the sub-dwarf of the ALL cell of Node 6. Since only one sub-dwarf exists in inputDwarfs (the one where C1 points to), immediate coalescing happens (case in Line 1) and the ALL cell points to Node 7, where C1 points to. Now, the sub-dwarf of the ALL cell for Node 1 must be created. The cell C1 will be added to the resulting node, and its sub-dwarf will be created by recursively calling SuffixCoalesce, where the only input dwarf will be the one that has Node 7 as its top node. Therefore, coalescing will happen there. Similarly, cells C2 and C3 will be added to the resulting node one by one, and coalescing will happen in the next level in both cases, because just one of the inputDwarfs contains each of these keys. Then the ALL cell for Node 8 must be created (Line 13). The key P1 is included in the nodes pointed by C1 and C3 (Nodes 7,4), and since we are at a leaf node, we must aggregate the values in the two cells (Line 8).

3.3.3 Memory Requirements

The PrefixExpansion algorithm has no major requirements, since it only needs to remember which was the previously read tuple. For the SuffixCoalesce algorithm, the priority queue (used to locate in Line 5 the cells with the minimum key), contains at each step one key from the top node of each dwarf in inputDwarfs. Since in the worst case we will descend all D levels of the structure when creating the ALL cell for the root node, the memory requirements for the priority queue (which are the only memory requirements for the algorithm) in the worst case of a fully dense Dwarf cube are equal to:

$$MaxMemoryNeeded = c \cdot \sum_{i=1}^{D} Card_i \tag{3.4}$$

where c is the size of the cell. However, since the cube is "always" sparse, the number of cells that must be kept in main memory will be *much* smaller than the sum of the dimensions' cardinalities, and the exact number depends on the branching factor at each level of the structure.

3.3.4 Proof of Correctness

In this section we prove by induction on the number of dimensions that the construction process described by the two interleaved algorithms PrefixExpansion and SuffixCoalesce constructs a correct representation of the data cube. Let's assume that the number of dimensions is represented with the symbol d. First we prove that the dwarf for $d = 1$ is a correct representation the we assume that the process

works for $d = k$ and finally we prove that the construction is correct for $d = k + 1$.

Dwarf with $d = 1$

The PrefixExpansion algorithm creates one node with as many cells as unique tuples in the fact table aggregating the measures of identical tuples. The SuffixCoalesce algorithm then aggregates the cell in the node and creates the special ALL cell. This results in a correct one-dimensional cube representation.

Dwarf with $d = k$

We assume that the process is correct when the number of dimensions $d = k$. In other words we assume that the two interleaved algorithms PrefixExpansion and SuffixCoalesce construct a correct cube representation for a k-dimensional data cube.

Dwarf with $d = k + 1$

In this case the PrefixExpansion algorithm creates a node that points to sub-dwarfs of k dimensions. By our assumption in the second step of the induction these dwarfs are correct representations of the corresponding k-dimensional data cubes. The suffix coalesce algorithm merges all the sub-dwarfs by calling recursively the suffix coalesce for k-dimensional dwarfs. Again by our assumption this step creates correct representations of the corresponding k-dimensional data cubes. The interleaved process in this case create a node that points to correct representations of k-dimensional data cubes and therefore it is a correct representation of the $(k + 1)$-dimensional data cube.

3.3.5 Incremental Updates

The ability to refresh data in a modern data warehouse environment is currently more important than ever. As the data stored increases in complexity, the possibility of incrementally updating the data warehouse/data-mart becomes essential. The "recompute everything" strategy cannot keep up the pace with the needs of a modern business. The most common strategy is using semi-periodic bulk updates of the warehouse, at specific intervals or whenever up-to-date information is essential.

In this section we describe how the Dwarf structure is incrementally updated, given a set of delta tuples from the data sources and an earlier version of the Dwarf cube. We assume that the delta updates are much smaller in size compared to the information already stored. Otherwise, a bulk incremental technique that merges [KR98] the stored aggregates with the new updates and stores the result in a new Dwarf might be preferable than the in-place method.

The incremental update procedure starts from the root of the structure and recursively updates the underlying nodes and finishes with the incremental update of the node that corresponds to the special ALL cell. By cross-checking the keys stored in the cells of the node with the attributes in the delta tuples, the procedure skips cells that do not need to be updated, expands nodes to accommodate new cells for new attribute values (by using overflow pointers), and recursively updates those sub-dwarfs which might be affected by one or more of the delta tuples.

Since the delta information is much less compared to the information already stored, the number of the cells that are skipped is much larger than the number of

cells that need to be updated. One case requires special attention: by descending the structure, we can reach a coalesced node from different paths. Once we get to the coalesced node we have to check if the coalesced path is still valid, since the insertion of one or more tuples might have caused the coalesced pointer to become invalid. In this case, the corresponding subdwarf has to be re-evaluated, and any new nodes have to be written to a different area of the disk. However, it is important to realize that an invalid coalesced pointer does not mean that the entire subdwarf needs to be copied again. Coalescing to nodes of the old dwarf will most likely happen just a few levels below in the structure, since only a small fraction of all the aggregate values calculated is influenced by the update.

An important observation is that frequent incremental update operations slowly deteriorate the original clustering of the Dwarf structure [4], mainly because of the overflow nodes created. This is an expected effect, encountered by all dynamic data structures as a result to online modifications. Since Dwarf is targeted for data warehousing applications that typically perform updates in scheduled periodic intervals, we envision running an process in the background periodically for reorganizing the Dwarf and transferring it into a new file with its clustering restored.

[4]The query performance of Dwarf still remains far ahead of the closest competitor as shown in our experiments section.

3.4 Performance issues

3.4.1 Query Execution

A point query is a simple traversal on the Dwarf structure from the root to a leaf. At level i, we search for the cell having as key the i-th coordinate value in the query and descend to the next level. If the i-th coordinate value is ALL, we follow the pointer of the ALL cell. A point query is fast simply because it involves exactly D node visits (where D is the number of dimensions).

Range queries differ from point queries in that they contain at least one dimension with a range of values. If a range is specified for the i-th coordinate, for each key satisfying the specified range we recursively descend to the corresponding sub-dwarf in a depth-first manner. As a result, queries on the Dwarf structure have trivial memory requirements (one pointer for each level of the structure).

According to the algorithms for constructing the Dwarf cube, certain views may span large areas of the disk. For example, for a 4-dimensional cube with dimensions a, b, c, d, view $abcd$ is not clustered, since all views containing dimension a (views $a, ab, ac, ad, abc, abd, acd$) are all interleaved in the disk area that view $abcd$ occupies. Therefore, a query with multiple large ranges on any of these views would fetch nodes that contain data for *all* these views. For this reason, we deviate from the construction algorithm, in order to cluster the Dwarf cube more efficiently. This is described in the following section.

3.4.2 Clustering Dwarf Cubes

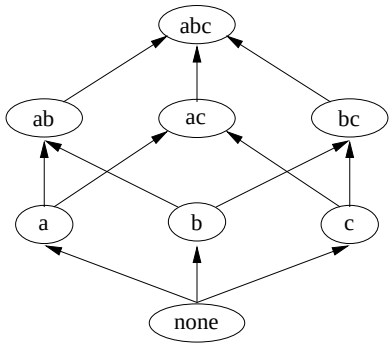


Figure 3.2: Data Cube Lattice for dimensions a, b and c

The algorithms described in section 3.3 present the general principles for constructing Dwarf structures. However there is a lot of room for improvement as far as the clustering of the structure is concerned. As we mentioned, the algorithms do not cluster views of the cube together and therefore accessing one view requires accessing nodes that are probably on different disk pages that are too far apart from each other. In this section we describe how the Dwarf structure can be created in a very clustered manner. Typically, the clustered version of the dwarfs decreased the query response time in real datasets by a factor of 2 to 3.

The lattice representation [HRU96a] of the Data Cube is used to represent the computational dependencies between the group-bys of the cube. An example for three dimensions is illustrated in Figure 3.2. Each node in the lattice corresponds to a group-by (view) over the node's dimensions. For example, node *ab* represents the group-by *ab* view. The computational dependencies among group-bys are represented in the lattice using directed edges. For example, group-by *a* can be computed from the *ab* group-by, while group-by *abc* can be used to compute any other group-

by. In Figure 3.2 we show only dependencies between adjacent group-bys, but we refer to the transitive closure of this lattice.

In Table 3.1 we illustrate an ordering of the views for a three dimensional cube. The second column of the table contains a binary representation of the view with as many bits as the cube's dimensions. An aggregated dimension has the corresponding bit set to true(1). For example view ab corresponds to 001 since the dimension c is aggregated. The views are sorted in increasing order based on their binary representation.

View	Binary Rep	Parents w/ Coalesce
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 3.1: View ordering example

This ordering has the property that whenever a view w is about to be computed, all the candidate ancestor views v_i with potential for suffix coalescing have already been computed. Note that the binary representation for v_i can be derived from the binary representation of w by resetting any one true bit (1) to false (0). This essentially means that the binary representation of v_i is arithmetically less than the binary representation of w and therefore precedes that in the sorted ordering. For example, in Table 3.1, view $w = a(011)$ has ancestors $v_1 = ab(001)$ and

$v_2 = ac(010)$. Figure 3.3 demonstrates the processing tree for the example in Table 3.1. In this order we have chosen to use the ancestor v_i with the biggest common prefix for w.

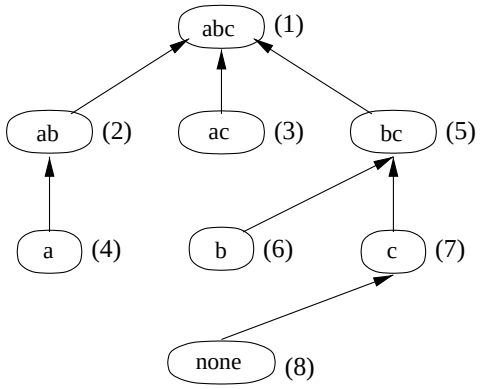


Figure 3.3: Processing Tree

By removing the recursion in the algorithms in Section 3.3 (lines 8,14 in the PrefixExpansion algorithm, and line 13 in the SuffixCoalesce algorithm) we are able to create any one view of the cube. More specifically, the most detailed view (in our example *abc*) can be created with PrefixExpansion, while any other view can be created with the SuffixCoalesce algorithm. Therefore it is easy to iterate through all the views of the cube using the described ordering and create each one of them. This procedure clusters nodes of the same view together and the resulting Dwarf structure behaves much better. For example, consider the structure in Figure 3.1. If this structure is created using the algorithms in Section 3.3 then the nodes will be written in the order: 123456789. Note that node 5 that belongs to view $\langle Store, ALL, Product \rangle$ is written between nodes 4 and 6 that belong to view $\langle Store, Customer, Product \rangle$, therefore destroying the clustering for both views. However, the procedure described here creates the nodes in the order 123467589,

maintaining the clustering of each view. Table 3.2 describes in more detail the procedure.

View	Binary Rep	Nodes
Store,Customer,Product	000	create 1,2,3,4,6,7
Store,Customer	001	close 3,4,7
Store,Product	010	create 5, coalesce to 7
Store	011	close 5,7
Customer,Product	100	create 8, coalesce to 7,4,3
Customer	101	
Product	110	create 9
none	111	close 9

Table 3.2: Example of creating a clustered Dwarf

3.4.3 Optimizing View Iteration

In our implementation we used a hybrid algorithm which does not need to iterate over all views. The hybrid algorithm takes advantage of the situation encountered while creating view $\langle Store, Customer \rangle$ or view $\langle Store \rangle$ as described in Table 3.2. Iterating over these two views did not create any new nodes, but rather closed the nodes by writing the ALL cell.

The situation is more evident in very sparse cubes (usually cubes of high dimensionalities). Assume a five-dimensional cube with ten thousand tuples where each dimension has a cardinality of one hundred. Let us assume that data values are uniformly distributed. The Dwarf representation of view *abcde* (00000) consists of five levels. The first level has only one node with one hundred cells. The second level for every cell of the first one has a node with another one hundred cells. The third level however -since we assumed that the data are uniform and there only ten

thousand tuples- has nodes that consist of only of one cell. Therefore we can close the corresponding cells right away. Thus we avoid iterating on views *abcd*(00001), *abce*(00010), *abc*(00011) and *abde*(00100).

3.4.4 Coarse-grained Dwarfs

Even though the Dwarf structure achieves remarkable compression ratios for calculating the entire cube, the Dwarf size can be, in cases of sparse cubes, quite larger than the fact table. However we can trade query performance for storage-space by using a *granularity* G_{min} parameter. Whenever at some level of the Dwarf structure (during the Dwarf construction) the number of tuples that contributes to the subdwarf beneath the currently constructed node N of level L is less than G_{min}, then for that subdwarf we do not compute any ALL cells. All the tuples contributing to this *coarse-grained* area below node N can be stored either in a tree-like fashion (thus exploiting prefix redundancy), or as plain tuples (which is useful if the number of dimensions D is much larger than L, to avoid the pointers overhead). Notice that for all these tuples we need to store only the last $D - L$ coordinates, since the path to the collapsed area gives as the missing information. Each query accessing the coarse-grained area below node N will require to aggregate at most G_{min} tuples to produce the desired result. The user can modify the G_{min} parameter to get a Dwarf structure according to his/her needs.

3.5 Experiments

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 single 700Mhz Celeron processor running Linux 2.4.12 with 256MB of RAM. We used a 30GB disk rotating at 7200 rpms, able to write at about 8MB/sec and read at about 12MB/sec. We purposely chose to use a low amount of RAM memory to allow for the effect of disk I/O to become evident and demonstrate that the performance of Dwarf does not suffer even when limited memory resources are available.

Our implementation reads a binary representation of the fact table, where all values have been mapped to integer data (4 bytes). Unless specified otherwise, all datasets contained one measure attribute, and the aggregate function used throughout our experiments was the SUM function. The reported times are actual times and contain CPU and I/O times for the total construction of Dwarf cubes including the initial sorting of the fact table.

In the experiments we compared Dwarf to Cubetrees, as far as storage space, creation time, queries and update performance are concerned. In [KR98] Cubetrees were shown to exhibit at least 10 times faster query performance when compared to indexed relations, half the storage a commercial relational system requires and at least 100 times faster update performance. Since no system has been shown to outperform the Cubetrees so far, we concluded that this was the most challenging test for Dwarf.

3.5.1 Cube construction

Prefix redundancy vs Suffix Coalescing

In this experiment we explore the benefits of eliminating prefix redundancy, and using suffix coalescing when computing the CUBE operator. For the first set of experiments, we used a binary storage footprint (BSF) as a means of comparison. The BSF representation models the storage required to store the views of the cube in unindexed binary relations. This representation was also used by [BR99] to estimate the time needed to write out the output of the cube.

	uniform				80-20	
#Dims	BSF	Dwarf Prefix only	Dwarf (MB)	Time (sec)	Dwarf (MB)	Time (sec)
10	2333 MB	1322 MB	62	26	115	46
15	106 GB	42.65 GB	153	68	366	147
20	4400 GB	1400 GB	300	142	840	351
25	173 TB	44.8 TB	516	258	1788	866
30	6.55 PB	1.43 PB	812	424	3063	1529

Table 3.3: Storage and creation time vs #Dimensions

In Table 3.3, we show the storage and the compute time for Dwarf cubes as the number #Dims of dimensions range from 10 to 30. The fact table contained 100000 tuples and the dimension values were either uniformly distributed over a cardinality of 1000 or followed a 80-20 Self-Similar distribution over the same cardinality. We did not impose any correlation among the dimensions. The BSF column shows an estimate of the total size of the cube if its views were stored in unindexed relational tables. The "Dwarf w/ Prefix only" column shows the storage of the Dwarf with the suffix coalescing off, and therefore, without suffix redundancy elimination.

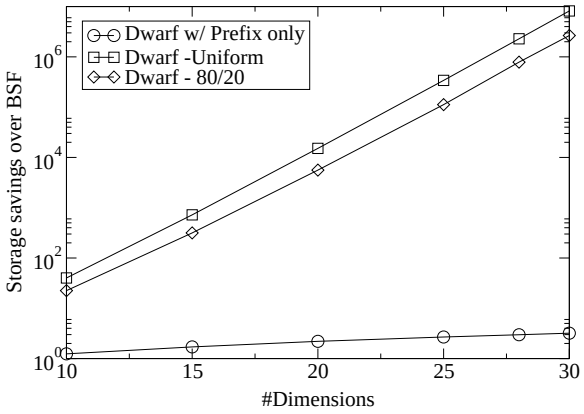


Figure 3.4: Dwarf Compression Ratio over BSF (log scale) vs #Dimensions

To measure the BSF size and the "Dwarf w/ Prefix only" size, we generated the Dwarf with the suffix coalescing turned on, and then traversed the Dwarf structure appropriately. We counted the BSF and the "Dwarf w/ prefix only" storage for both distributions and the results (as far as the savings are concerned) were almost identical -slightly smaller savings for the 80-20 distribution-, so we just present the uniform sizes. The remaining four columns show the Dwarf store footprint and the time to construct it for each of the two distributions. Figure 3.4 shows the compression over the BSF size as the number of dimensions increases. Observe that, as the cube becomes sparser, the savings increase exponentially due to suffix coalescing.

We observe the following:

- Elimination of prefix redundancy saves a great deal, but suffix redundancy is clearly the dominant factor in the overall performance.

- The creation time is proportional to the Dwarf size.

- The uniform distribution posts the highest savings. The effect of skew on the cube is that most tuples from the fact table contribute to a small part

of the whole cube while leaving other parts empty. The denser areas benefit from prefix elimination which is smaller, and sparser areas have less suffix redundancy to eliminate (since fewer tuples exist there).

Table 3.4 gives the Dwarf storage and computation time for a 10-dimensional cube when the number of tuples in the fact table varies from 100000 to 1000000. The cardinalities of each dimension are 30000 , 5000, 5000, 2000, 1000, 1000, 100, 100, 100 and 10. The distribution of the dimension values were either all uniform or all 80-20 self-similar. This set of experiments shows that the store size and computation time grow linearly in the size of the fact table (i.e. doubling the input tuples results in a little more than twice the construction time and storage required).

	uniform		80-20	
#Tuples	Dwarf(MB)	Time(sec)	Dwarf(MB)	Time(sec)
100,000	62	27	72	31
200,000	133	58	159	69
400,000	287	127	351	156
600,000	451	202	553	250
800,000	622	289	762	357
1,000,000	798	387	975	457

Table 3.4: Storage and time requirements vs #Tuples

Comparison with Full Cubetrees

In this experiment we created cubes of fewer dimensions, in order to compare the performance of Dwarf with that of Cubetrees. We created full cubes with the number of dimensions ranging from 4 to 10. In each case, the fact table contained 250000 tuples created by using either a uniform, or a 80-20 self-similar distribution. In

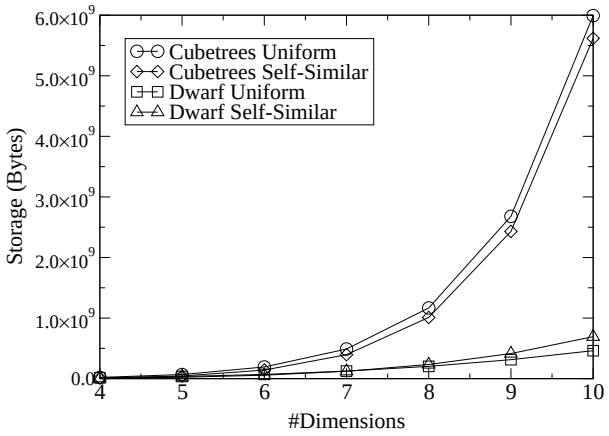


Figure 3.5: Storage Space vs #Dimensions

Figure 3.5 we show the space required for Dwarf and for Cubetrees to store the entire cube. Figure 3.6 shows the corresponding construction times. From these two Figures we can see that:

- Cubetrees do not scale, as far as storage space is concerned, with the number of dimensions. On the contrary, Dwarf requires much less space to store the same amount of information.

- Dwarf requires significantly less time to build the cube. This is because Cubetrees (like other methods that calculate the entire cube) perform multiple sorting operations on the data, and because Dwarf avoids computing large parts of the cube, since suffix coalescing identifies parts that have already been computed.

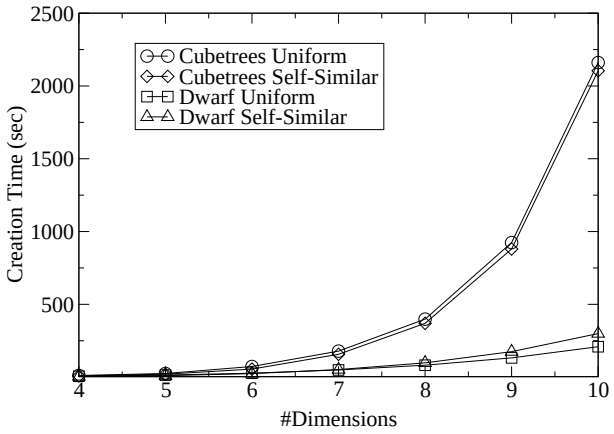


Figure 3.6: Construction Time vs #Dimensions

Comparison to Reduced Cubetrees

This experiment compares the construction time of Dwarf with that of Cubetrees when the Cubetrees size is limited to that of the Dwarf structure. We will refer to this type of Cubetrees as *reduced* Cubetrees. This is useful to examine, since in many cases of high-dimensional data, Cubetrees (and most other competitive structures) may not fit in the available disk space. Since the Cubetrees will not store all the views of the CUBE operator, we have to make a decision of which views to materialize. The PBS algorithm [SDN98] provides a fast algorithm to decide which views to materialize under a given storage constraint, while at the same time guaranteeing good query performance. The PBS algorithm selects the smallest views in size, which are typically the views that have performed the most aggregation. In addition, we have also stored in the *reduced* Cubetrees the fact table, in order for them to be able to answer queries (in the Queries section) on views which are not materialized or cannot be answered from other materialized

views.

Dataset	#Dims	#Tuples	Size (MB)	Cubetrees Time(sec)	Dwarf Time(sec)	PBS Views
Meteo-9	9	348448	66	64	35	63 out of 512
Forest	10	581012	594	349	350	113 out of 1024
Meteo-12	12	348448	358	451	228	310 out of 4096

Table 3.5: Storage and Creation Time for Real Datasets

Table 3.5 gives the Dwarf and reduced Cubetrees storage and creation times for three real datasets. Cubetrees were created having the same size as the corresponding Dwarfs. The construction times of the reduced Cubetrees do not include the running time for the PBS algorithm. The table also shows the number of views contained in the reduced Cubetrees. The first real dataset contains weather conditions at various weather stations on land for September 1985 [HWL]. From this dataset we created two sets - Meteo-9 and Meteo-12 - of input data: one which contained 9 dimensions, and one with 12 dimensions. The second real data-set contains "Forest Cover Type" data [Bla98] which includes cartographic variable that are used to estimate the forest cover type of land areas. In all data sets some of the attributes were skewed and among some dimensions there was substantial correlation.

Even though the reduced Cubetrees calculate significantly fewer views that Dwarf does, Dwarf cubes are significantly faster at their creation for the two Weather datasets, and took the same amount of time as the Cubetrees for the Forest dataset. One important observation is that the Dwarf structure for the Weather dataset with 12 dimensions is smaller, and faster to compute than the Dwarf for the Forest data, which had 10 dimensions. The top three dimensions in the Weather data were highly

correlated and suffix coalescing happened at the top levels of the Dwarf structure in many cases, thus providing substantial space and computational savings.

3.5.2 Query Performance

In this section we study the query performance of Dwarf when compared to full and reduced Cubetrees. We also give a detailed analysis of how range queries, applied to different levels of the Dwarf structure, are treated by both the clustered and unclustered structure.

Dwarfs vs Full Cubetrees

We created two workloads of 1000 queries, and queried the cubes created in the previous experiment (full cubes of 4-10 dimensions with 250000 tuples). The description of the workloads is presented in Table 3.6.

		Probabilities			Range	
Workload	#Queries	P_{newQ}	P_{dim}	P_{pointQ}	Max	Min
A	1000	0.34	0.4	0.2	20%	1
B	1000	1.00	0.4	0.2	20%	1

Table 3.6: "Dwarfs vs Full Cubetrees" Query Workload

Since other query workloads will also be given in tables similar to Table 3.6, we give below a description on the notation used. An important thing to consider is that in query workloads to either real data, or synthetic data produced by using the uniform distribution, the values specified in the queries (either point values, or the endpoints of ranges) are selected by using a uniform distribution. Otherwise, we use the 80/20 Self-Similar distribution to produce these values. This is more

suitable, since we suspect that the user will typically be more interested in querying the denser areas of the cube.

P_{newQ} The probability that the new query will not be related to the previous query. In OLAP applications, users typically perform a query, and then often execute a series of roll-up or drill-down queries. When our query generator produces a query, it produces a roll-up query with probability $(1 - P_{newQ})/2$, a drill-down query with the same probability or a new query with probability P_{newQ}. For example, Workload B creates only new (unrelated) queries, while workload A creates a roll-up or a drill-down with a probability of 0.33 each.

P_{dim} The probability that each dimension will be selected to participate in a new query. For example, for a 10-dimensional cube, if the above probability is equal to 0.4, then new queries will include $10 \cdot 0.4 = 4$ dimensions on average.

P_{pointQ} The probability that we specify just a single value for each dimension participating in a query. Otherwise, with probability $1 - P_{pointQ}$ we will specify a range of values for that dimension. This way we control how selective our queries will be: a value of 1 produces only point queries, and a value of 0 produces queries with ranges in every dimension participating in the query. In most of our experiments we selected low values for this parameter, since a high value would result in most queries returning very few tuples (usually 0).

Range The range for a dimension is uniformly selected to cover a specified percentage of the cardinality of the dimension. For example, if a dimension a has

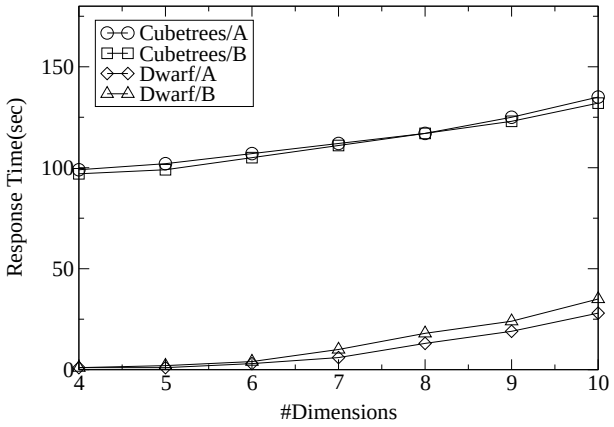


Figure 3.7: Query Performance on Uniform Data

values ranging from 1 to 1000, a 20% value maximum range will force any range of dimension a to be limited to at most 200 values. Each range contains at least one value.

Returning to the experiment, the results for the workloads of Table 3.6 on the cubes created in the previous experiment are shown in Figures 3.7 and 3.8. Dwarf outperforms Cubetrees in all cases, and for small-dimensionality Dwarf cubes are 1-2 orders of magnitude faster. The main advantage of Dwarf cubes is their condensed storage, which allows them to keep in main memory a lot more information than Cubetrees can. Moreover, we can see that Dwarf performs better in workload A, because roll-up and drill-down queries have a common path in the Dwarf structure with the previously executed query, and thus the disk pages corresponding to the common area are already in main memory. For example, for the 10-dimensional cases, in the Uniform dataset the response time drops from 35 to 28 seconds when roll-up and drill-down operations are used (a 20% reduction), while for the Self-

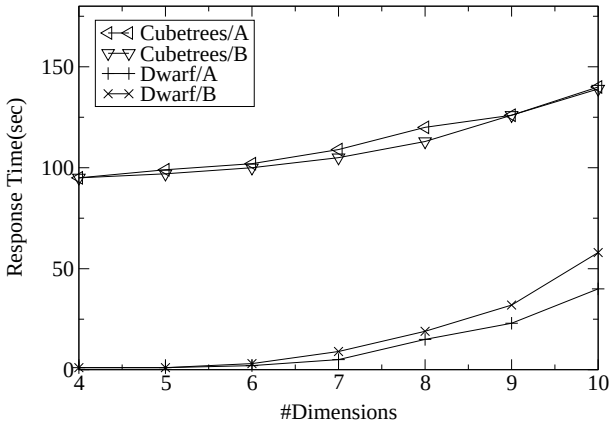


Figure 3.8: Query Performance on Self-Similar Data

Similar case the improvement is even larger: from 58 to 40 seconds. This is a 31% reduction in response time.

Dwarfs vs Reduced Cubetrees

In this set of experiments, we compare the query performance of Dwarfs with that of reduced Cubetrees. The datasets used in this experiment were the real datasets described in Section 3.5.1 (Meteo-9, Meteo-12, Forest). Since Cubetrees in this case did not contain all the views of the cube, we need to explain how we answered queries on non-materialized views.

When a query on a non-materialized view v is issued, the Cubetree optimizer picks the best materialized view w to answer v. If v does not share a common prefix with w, then it uses a hash-based approach to evaluate the query. If however v shares a common prefix with w, then the result is calculated on the fly, taking advantage of the common sort order. The second case is much faster than using a hash-table.

The Cubetree optimizer needs estimates for the size of all views, but in our case we had the exact sizes by issuing appropriate queries to the Dwarf structure.

Workload	#Queries	P_{newQ}	$\overline{\#Dims}$	P_{pointQ}	Range_{max}
A	2000	0.34	4	0.1	15%
B	2000	0.34	4	0.5	25%
C	2000	1.00	4	0.5	25%
D	2000	0.34	3	0.5	25%
E	2000	1.00	3	0.5	25%

Table 3.7: "Dwarfs vs Reduced Cubetrees" Query Workload

For each real dataset we created 5 workloads of 2000 queries, whose characteristics are presented in Table 3.7. Here, the $\overline{\#Dims}$ column denotes the average number of dimensions specified on each query. Notice that workloads C and E are similar to workloads B and D, respectively, but contain no roll-up/drill-down queries.

The query performance of Dwarf and the reduced Cubetrees is presented in Table 3.8. Dwarf is about an order of magnitude faster than the reduced Cubetrees in the Weather datasets Meteo-9, Meteo-12), and $2-3$ times faster in the Forest dataset. Dwarf performs significantly better in the Weather datasets due to the correlation of the attributes in these datasets. Because coalescing happened at the top levels of the structure, a large fraction of nodes at the top levels were cached, thus improving performance dramatically.

An important observation is that Dwarfs are faster when the workload contains roll-up/drill-down queries. For example, for workloads D and E of the forest dataset, Dwarf was 17% faster. Also notice that in this type of workloads the limitation of

the average number of dimensions specified in each query, favors Cubetrees, which typically store views with up to 3 dimensions, because of the PBS algorithm. For workloads with queries containing more dimensions, on average, the performance of the Cubetrees was significantly worse.

	Reduced Cubetrees			Dwarf		
Workload	Meteo-9	Meteo-12	Forest	Meteo-9	Meteo-12	Forest
A	305	331	462	13	34	150
B	292	346	478	13	39	176
C	304	340	483	13	44	208
D	315	301	427	12	47	217
E	305	288	448	15	49	262

Table 3.8: Query Times in Seconds for 2000 Queries on Real Datasets

Evaluating Ranges in Dwarf Cubes

One of our initial concerns when designing Dwarf was to ensure that their query performance would not suffer on queries with large ranges on the top dimensions of the structure. The dimensions on the higher levels of the Dwarf structure have higher cardinalities, and thus a large range on them (for example a range containing 20% of the values) might be expensive because a large number of paths would have to be followed. In the following experiment we study the query behavior of Dwarf, when queries with ranges in different dimensions are issued.

We first created four 8-dimensional datasets, each having a fact table of 800,000 tuples. For the first 2 datasets (A_{uni}, $A_{80/20}$), each dimension had a cardinality of 100. For the last 2 datasets (B_{uni},$B_{80/20}$), the cardinalities of the dimensions were: 1250, 625, 300, 150, 80, 40, 20 and 10. The underlying data used in datasets A_{uni} and

B_{uni} was produced by using a Uniform distribution, while for the other 2 datasets we used the 80-20 Self-Similar distribution. For reference, the sizes of the Dwarf cubes for the A_{uni} and $A_{80/20}$ datasets were 777 and 780 MB, while for the B_{uni} and $B_{80/20}$ datasets the corresponding sizes were 490 and 482 MB.

We decided to create workloads of queries, where three consecutive dimensions would contain ranges on them. For example, if we name the cube's dimensions $a_1, a_2, ..., a_8$, then the first workload would always contain ranges on dimensions a_1, a_2, a_3, the second workload on dimensions a_2, a_3, a_4... We also considered ranges on dimensions a_7, a_8, a_1 and on a_8, a_1, a_2. Each workload contained 1000 queries. Since a set of three dimensions was always queried in each workload, we also issued a point query on the remaining dimensions with probability 30% -otherwise the ALL value is selected-. Having point queries on few dimensions allowed our queries to "hit" different views while the three ranged dimensions remained the same, and the small probability with which a point query on a dimension happens allowed for multiple tuples to be returned for each query. Each range on a dimension contained 5-15% of the dimension's values. The results for datasets A_{uni} and $A_{80/20}$ are presented in Table 3.9, and for datasets B_{uni} and $B_{80/20}$ in Table 3.10. To view the effect of clustering on Dwarf cubes, we also present the query times achieved by Dwarf when we use the original PrefixExpansion and SuffixCoalesce algorithms, without improving clustering the way we described in Section 3.4.2. We refer to the corresponding structure as "Unclust. Dwarfs". For comparison reasons we have also included the corresponding query times for the full Cubetrees[5]. In the

[5]to minimize the effect of online aggregation

following paragraphs we will focus our discussion on how the query performance of Dwarf cubes is influenced by the location of ranges. The behavior of Cubetrees was explained in [KR98].

	A_{uni}				$A_{80/20}$			
Ranged Dims	Cubetrees (sec)	Dwarf (sec)	Unclust. Dwarf (sec)	Result Tuples	Cubetrees (sec)	Dwarf (sec)	Unclust. Dwarf (sec)	Result Tuples
1,2,3	142	8	20	60663	170	13	18	158090
2,3,4	126	9	21	78587	146	15	18	150440
3,4,5	117	10	37	65437	128	15	22	162875
4,5,6	113	13	41	78183	114	16	42	165284
5,6,7	110	18	53	72479	108	17	42	150926
6,7,8	109	22	39	69165	104	13	18	163357
7,8,1	126	28	53	71770	119	23	23	153532
8,1,2	134	17	19	86547	154	19	19	155837

Table 3.9: Time in seconds for 1000 queries on datasets with constant cardinality

In Table 3.9, for the uniform workload A_{uni} and the clustered Dwarf we can observe that the query performance decreases as the ranges move to lower dimensions. In this case, the query values (either point, or ALL), on dimensions above the ranged ones, randomly hit different nodes at the upper levels of the structure. This has the effect that consecutive queries can follow paths in vastly different locations of the Dwarf file. Since the Dwarf does not fit into main memory, the larger the area targeted by queries of each workload, the more swapping that takes place to fetch needed disk pages to main memory, and the worse the query performance. Thus, the performance degrades as the queries move towards the lower levels, because a larger area of the Dwarf file is targeted by the queries, and caching becomes less effective.

Here we need to clarify that a single query with ranges on the top dimensions is

more expensive than a single query with ranges on the lower dimensions. However, consecutive queries involving ranges on the top levels of the structure benefit more from caching, since after a few queries the top level nodes where the ranges are applied will be in main memory. This is why this kind of queries exhibited better performance in the experiment.

The same behavior can be observed for the unclustered Dwarf, with one exception, the 6,7,8 ranges. In this case the benefits of the reduced per-query cost seem to outweigh the cache effects resulting in better overall performance.

Overall, the unclustered Dwarf performs much worse than the clustered one -although still much better compared to Cubetrees-. The reason for the worse behavior of the unclustered Dwarf is (as we have mentioned before) the interleaving of the views. This has the result that most disk pages fetched contain "useless" information for the query, and thus more pages need to be fetched when compared to the clustered Dwarf.

The above concepts can help explain the behavior of the Dwarf structure for the *wrapped queries* on dimensions 8,1,2 and 7,8,1. The ranges at the top dimensions benefit the cache performance but increase the per-query cost. The tradeoff between the two determines the overall performance.

A similar behavior can be observed for the $A_{80/20}$ workload. In this case the queries address denser areas compared to that of the uniform case, as the returned tuples and the overall performance demonstrate. Dwarf performs similarly to the A_{uni} case.

	B_{uni}				$B_{80/20}$			
Ranged Dims	Cubetrees (sec)	Dwarf (sec)	Unclust. Dwarf (sec)	Result Tuples	Cubetrees (sec)	Dwarf (sec)	Unclust. Dwarf (sec)	Result Tuples
1,2,3	206	20	37	96383	271	66	87	2582365
2,3,4	164	11	25	106879	183	29	38	1593427
3,4,5	130	11	22	106073	129	13	17	427202
4,5,6	112	14	19	56261	107	12	16	85862
5,6,7	105	16	15	12327	99	10	8	21808
6,7,8	103	17	12	2773	96	9	8	5173
7,8,1	180	19	73	47436	165	19	39	93139
8,1,2	180	24	37	115291	228	28	34	989998

Table 3.10: Time in seconds for 1000 queries on datasets with varying cardinalities

Table 3.10 presents the query performance of Dwarf for the datasets B_{uni} and $B_{80/20}$. The extra parameter -and the dominating one- to be considered here is the different cardinalities, as a range (i.e. 10%) on the top dimension contains much more values than the same range does in any other dimension. The effect of the different cardinalities is more evident in the $B_{80/20}$ workload. This happens because a given range will be typically satisfied by a lot more values than in the uniform case (recall that 80% of the values exist in 20% of the space). This is evident from both the number of result tuples, and from the query performance which improves when the queries are applied to dimensions with smaller cardinalities. However the basic concepts described for Table 3.9 apply here as well.

Coarse-grained Dwarfs

As described in Section 3.4.4, we can limit the space that Dwarf occupies and subsequently computation time, by appropriately setting the minimum *granularity* (G_{min}) parameter. In this set of experiments we investigate how the construction time, space, and query performance of Dwarfs are influenced when increasing the G_{min}

threshold. We created the Dwarf structure for the 8-dimension cubes of the B_{uni} and $B_{80/20}$ datasets (see previous experiment) for different values of the G_{min} parameter and then issued 8,000 queries on each of the resulting Dwarf cubes. The description of the queries was the same as in the case of the rotated dimensions. Table 3.11 presents the creation times, the required storage, and the time required to execute all 8,000 queries for each Dwarf.

	Uniform Distribution			80-20 Distribution		
G_{min}	Space (MB)	Construct (sec)	Queries (sec)	Space (MB)	Construct (sec)	Queries (sec)
0	490	202	154	482	218	199
100	400	74	110	376	81	262
1000	312	59	317	343	62	295
5000	166	29	408	288	53	1094
20,000	151	25	476	160	30	1434

Table 3.11: Performance measurements for increasing G_{min}

When we increase the value of G_{min}, the space that Dwarf occupies decreases, while at the same time query performance degrades. The only exception was for the Uniform distribution and G_{min} value of 100, where the reduction of space actually improved query performance, despite the fact that some aggregations needed to be done on-the-fly. The reason is that coarse-grained areas for this value fit in one -or at most two- pages and it is faster to fetch them and do the aggregation on the fly, rather than fetching two or more pages to get to the precomputed aggregate.

In Table 3.11 the pay-off in construct time is even higher than the space savings. A G_{min} value of 20000 results in 3 to 1 storage savings, but in more than 7 to 1 speedup of computation times. After various experiments we have concluded that a value of G_{min} between 100 and 1000 typically provides significant storage/time

savings with small degradation in query performance.

3.5.3 Updates

In this section we present experimental results to evaluate the update performance of Dwarfs when compared to the full and reduced Cubetrees.

Synthetic Dataset

In this experiment, we used the 8-dimension dataset B_{uni}. We originally constructed the Dwarf, the full Cubetrees and the reduced Cubetrees with 727,300 tuples and then proceeded to add 10 increments of 1% each (to reach the total of 800,000 tuples). The reduced Cubetrees were selected to have about the same size as the Dwarf cube when both are constructed using 727.300 tuples. Table 3.12 shows the update time for all 3 structures. We clearly see that the full Cubetrees require significantly more time, since their size is much larger than that of the Dwarf structure. Dwarf performs better at the beginning when compared to the incremental updates of the reduced Cubetrees. For example, for the first incremental update, the reduced Cubetrees took 34% more time than Dwarf. As we update the structures with more and more data, the difference in update times becomes smaller, and eventually Dwarf becomes more expensive to update incrementally. The main reason for this is the degradation of the Dwarf's clustering as nodes are expanded during updates, and overflow nodes are added to the structure. To demonstrate this, we ran on the final Dwarf structure (after the 10 increments had been applied) the same set of queries that we used in the previous experiment. Dwarf now required 211 seconds, 37% more time than the

154 seconds of the reorganized Dwarf.

We notice that Cubetrees (according to the specification of the update algorithm [KR98]) are always kept optimized by using new storage for writing the new aggregates. This results in having about twice the space requirements of Dwarf during updates, since the old structure is used as an input for the update process. The same technique can be implemented for Dwarf too. After a few increments we can reorganize the dwarf structure with a background process that writes a new Dwarf into new storage, restoring its clustering. For example, if we reorganize the Dwarf after the first 9 increments, the update time for the last increment is 82 seconds, which is faster than the corresponding update of the Cubetrees.

	Full Cubetrees		Dwarf		Reduced Cubetrees
Action	Time (sec)	Space (MB)	Time (sec)	Space (MB)	Time (sec)
Create	1089	3063	180	446	296
Update #1	611	3093	65	455	87
Update #2	605	3123	68	464	84
Update #3	624	3153	70	473	92
Update #4	618	3183	73	482	86
Update #5	631	3212	79	491	90
Update #6	626	3242	81	499	87
Update #7	636	3272	87	508	91
Update #8	633	3301	98	517	88
Update #9	651	3331	107	526	93
Update #10	644	3361	121	535	90

Table 3.12: Update performance on Synthetic Dataset

Using the APB-1 Benchmark Data

We tested the update performance of Dwarf on the APB-1 benchmark [Cou98], with the density parameter set to 8. The APB-1 benchmark contains a 4-d dataset

with cardinalities 9000, 900, 17 and 9 and two measure attributes. We mapped the string data of the fact table to integers, randomly permuted the fact table, and then selected about 90% of the tuples (22,386,000 tuples) to initially load the Cubetrees (full and reduced) and Dwarf, and then applied 10 successive increments of 1% each. Table 3.13 shows the results for the reduced Cubetrees and Dwarf. The full Cubetrees are always more expensive to update than the reduced Cubetrees (since they have more views to update) and, thus, are not included in the results. Dwarf surpassed the reduced Cubetrees in all the incremental updates. Moreover, it is interesting to notice that the update time of Dwarf decreased as more tuples were inserted. This is mainly because this dataset corresponded to a dense cube and, therefore, the number of coalesced tuples was small. Updating coalesced tuples is the most time consuming part of the incremental update operation for Dwarf. As more tuples were inserted, fewer coalesced links existed, and the update performance improved.

3.6 Summary

In this chapter we presented Dwarf, a highly compressed structure for computing, storing, and querying data cubes. Dwarf identifies prefix and suffix structural redundancies and factors them out by coalescing their storage. The Dwarf structure shows that suffix redundancy is the dominant factor in sparse cubes and its elimination has the highest return both in storage and computation time.

	Dwarf		Reduced Cubetrees
# Action	Time (sec)	Space (MB)	Time (sec)
Create	1124	346	1381
Update #1	42	350	76
Update #2	36	353	78
Update #3	39	359	77
Update #4	34	365	79
Update #5	24	369	80
Update #6	34	374	82
Update #7	24	378	79
Update #8	30	384	83
Update #9	22	390	82
Update #10	20	393	84

Table 3.13: Update performance on the APB-1 benchmark

Dwarf is practical because it is generated over a single pass over the data and requires no deep knowledge the underlying value distributions. It is scalable because the higher the dimensions the more the redundancy to harvest. Dwarf can be used to store the full cube (made possible because of its compact size) or, alternatively, precompute only aggregates whose computation will be too costly to be done on the fly, using the minimum granularity metric.

The great reduction in terms of storage space that the dwarf structure exhibits has positive effects in terms of query and update performance. The dwarf structure plays a double role as a storage and indexing mechanism for high dimension data. Roll-up and drill-down queries seem to benefit from the dwarf structure due to common paths that are exploited while caching. In terms of update speed, dwarf by far outperforms the closest competitor for storing the full data cube, while their performance is comparable when the competitor is reduced to storing only a partial cube of the same size as Dwarf.

Chapter 4

The Dwarf Complexity

4.1 Introduction

The data cube operator is an analytical tool which provides the formulation for aggregate queries over categories, rollup/drilldown operations and cross-tabulation. Conceptually the data cube operator encapsulates all possible multidimensional groupings and its an invaluable tool to applications that need analysis on huge amounts of data like decision support systems, business intelligence and data mining. Such applications need very fast query response on mostly ad-hoc queries that try to discover trends or patterns in the data set.

However the number of views of the data cube increases *exponentially* with the number of dimensions and most approaches are unable to compute and store but small low-dimensional data cubes. After the introduction of the data cube in [GBLP96] an abundance of research followed for dealing with its exponential complexity. The main ideas can be classified as either a cube sub-setting (partial materialization) [GHRU97b, HRU96b, TS97] or storing the full cube but with less precision (approximation or lossy models) [AGP00, VWI98]. However, all these techniques do not directly address the problem of space complexity. Furthermore, all problems associated with the data cube itself appeared to be quite difficult,

from computing it [AAD+96, DANR96, SAG96, ZDN97, BR99, RS97], storing it [JS97, FH00], querying and updating it[RKR97]. Even a problem that appears simpler, that of obtaining estimates on the cube size, is actually quite hard and needs exponential memory with respect to the dimensionality [SDNR96] in order to obtain accurate results.

Currently the most promising approaches for handling large high-dimensional cubes lie in the context of *coalesced* data cubes[SDRK02, LPZ03, WLFY02]. In [SDRK02] we demonstrate that the size of the dwarf data cube, even when every possible aggregate is computed, stored and indexed, is orders of magnitudes smaller than what expected. The coalescing discovery [SDRK02], completely changed the perception of a data cube from a collection of distinct groupings into a complex network of interleaved groupings that eliminates both *prefix* and *suffix redundancies*. It is these redundancies and their elimination that fuse the exponential growth of the size of high dimensional full cubes and dramatically condense their store without loss in precision.

To help clarify the basic concepts, let us consider a cube with three dimensions. In Table 4.1 we present such a toy dataset for the dimensions `Store`, `Customer`, and `Product` with one measure `Price`.

Store	Customer	Product	Price
S1	C2	P2	$70
S1	C3	P1	$40
S2	C1	P1	$90
S2	C1	P2	$50

Table 4.1: Fact Table for Cube Sales

The size of the cube is defined as the number of the tuples it contains, which

essentially corresponds to the sum of the tuples of all its 2^3 views. The size of the coalesced cube is defined as the total number of tuples it contains, *after* coalescing. For example, for the fact table in Table 4.1 and the aggregate function *sum* we have a cube size of 23 tuples, while the coalesced cube size is just 9 tuples as depicted in Table 4.2. The redundancy of the cube is eliminated by storing the coalesced areas just once. For example, the aggregate \$70 appears in total of five tuples, (S1|ALL,C2,P2|ALL) and (S1,ALL,P2), in the cube and it is coalesced in just one tuple. In [LPZ03] a similar notion of "quotient class" is used.

no	Coalesced	Price
1	(S1|ALL,C2,P2|ALL) (S1,ALL,P2)	\$70
2	(S1|ALL,C3,P1|ALL) (S1,ALL,P1)	\$40
3	(S1,ALL,ALL)	\$110
4	(S2|ALL,C1,P1) (S2,ALL,P1)	\$90
5	(S2|ALL,C1,P2) (S2,ALL,P2)	\$50
6	(S2|ALL,C1,ALL)	\$140
7	(ALL,ALL,P1)	\$130
8	(ALL,ALL,P2)	\$120
9	(ALL,ALL,ALL)	\$250

Table 4.2: Coalesced Cube Tuples

In this chapter we provide a framework for estimating the size of a coalesced cube and show that for a uniform cube the expected complexity is:

$$O\left(d^{\log_C T}\frac{T}{(\log_C T)!}\right) = O\left(T^{1+1/\log_d C}\right)$$

where d is the number of dimensions, C is the cardinality of the dimensions and T is the number of tuples. This result shows that, unlike the case of non-coalesced cubes which grow exponentially fast with the dimensionality, the 100%

accurate and complete (in the sense that it contains all possible aggregates) coalesced representation only grows *polynomially* fast. In other words, if we keep the number of tuples in the fact table constant and increase the dimensionality of the fact table (by horizontally expanding each tuple with new attributes) then the size of the coalesced cube scales only polynomially. The first form of the complexity shows that the dimensionality d is raised to $\log_C T$ which does not depend on d and is actually quite small for most realistic datasets[1].

The second form of the complexity shows that the coalesced size is polynomial w.r.t to the number of tuples of the data set T, which is raised to $1 + 1/\log_d C$ (and is very close to 1 for most realistic datasets[2]). In other words, if we keep the dimensionality of the fact table constant and start appending new tuples, then the size of the coalesced cube scales polynomially (and almost linearly). These results change the current state of the art in data-warehousing because it allows to scale up and be applicable to a much wider area of applications.

In addition we extend our analysis to cubes with varying cardinalities per dimension and we provide an efficient polynomial -w.r.t to the dimensionality- algorithm which can be used to provide close upper bounds for a coalesced cube based only on these cardinalities without actually computing the cube. Such estimates are invaluable for data-warehouse/OLAP administrators who need to preallocate the storage for the cube before initiating its computation. Current approaches [SDNR96] cannot be applied to high-dimensional data cubes, not only because they

[1]For example for a data set of 25 million tuples and a cardinality of 5,000, $\log_C T = 2$

[2]I.e., for a dimensionality of 30 and a cardinality of 5,000, $1 + 1/\log_d C \approx 1.4$

require an exponential amount of work per tuple and exponential amount of memory but mostly because they cannot be extended to handle coalesced cubes.

Although our algorithm is based on a uniform assumption is provides very accurate results for both zipfian and real datasets requiring as input only basic metadata about the cube –it's dimension cardinalities–.

In particular in this chapter we make the following contributions:

1. We formalize and categorize the redundancies found in the structure of the data cube into sparsity and implication redundancies

2. For the sparsity redundancies, we provide an analytical framework for estimating the size of the coalesced cube, and show that for uniform data sets it scales only polynomially w.r.t to the number of dimensions and number of tuples

3. We complement our analytical contributions with an efficient algorithm and an experimental evaluation using both synthetic and real data sets and we show that our framework not only provides accurate results for zipfian distribution but most importantly that real coalesced cubes scale *even better* than polynomially due to implication redundancies.

Our work provides the *first* analytical results showing that a full (i.e. contains all possible groupings and aggregates) and 100% accurate (no approximation) data cube is not *inherently exponential* in size and that an effective coalescing data cube model can reduce its size to realistic values. Therefore, we believe it has not only theoretical but also very practical value for data warehousing applications.

4.2 Redundancies

In this section we formalize the redundancies found in the structure of the cube and explain their extend and significance.

4.2.1 Prefix Redundancy

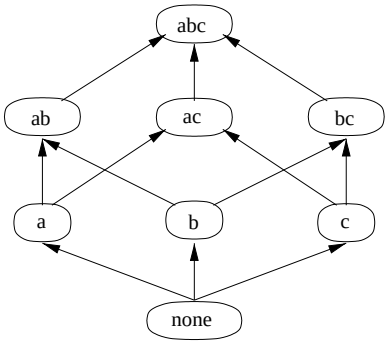


Figure 4.1: Lattice for the ordering a, b, c

This redundancy is the first that has been identified and can be used to build indexes over the structure of the cube. The idea is easily visualized in the lattice representation of the cube. For example, in Figure 4.1, one can observe that half the group-by's share the prefix a. We can exploit this by just storing the corresponding values just once and avoid replicating the same values over all views(prefix-reduction). By generalizing this to other prefixes (like for example to prefix b, which appears to one fourth of the views) we can reduce the amount of storage required to store the tuples of the cube.

Lemma 1 *The total number of tuples of the cube is not affected by prefix redundancy, only the storage required to store each tuple is reduced.*

This lemma essentially says that the prefix-reduced cube still suffers from the dimensionality curse, since we have to deal with every single tuple of the cube. The benefits of the prefix-reduction are therefore quickly rendered impractical even for medium dimensional cubes.

4.2.2 Suffix Redundancy

In this section we formally define the suffix redundancy and we give examples of different suffix redundancies.

DEFINITION 1 ***Suffix Redundancy*** *occurs when a set of tuples of the fact table contributes the exact same aggregates to different groupings. The operation that eliminates suffix redundancies is called* ***coalescing****. The resulting cube is called* ***coalesced cube****.*

EXAMPLE 1 *Suffix redundancy can occur for just a single tuple: In the fact table of Table 4.1, we observe that the tuple:*

$$\langle\ S1\ C2\ P2\ \$70\ \rangle$$

contributes the same aggregate \$70 *to two group-bys: (Store,Customer) and (Customer). The corresponding tuples are:*

(Store,Customer)	(Customer)
$\langle\ S1\ C1\ \$70\ \rangle$	$\langle\ C2\ \$70\ \rangle$

EXAMPLE 2 *We must point out that suffix redundancy does not work only on a per-tuple basis, but most importantly it extends to* whole *sub-cubes, for example the sub-cube that corresponds to the tuples:*

$$\langle\ S2\ C1\ P1\ \$90\ \rangle, \langle\ S2\ C1\ P2\ \$50\ \rangle$$

contributes the same aggregates to sub-cubes of (Store,Product), (Customer,Product), (Store), (Customer) :

(Store,Product)	(Customer,Product)
$\langle\ S2\ P1\ \$90\ \rangle$	$\langle\ C1\ P1\ \$90\ \rangle$
$\langle\ S2\ P2\ \$50\ \rangle$	$\langle\ C1\ P2\ \$50\ \rangle$

(Store)	(Customer)
$\langle\ S2\ \$140\ \rangle$	$\langle\ C1\ \$140\ \rangle$

The reason that whole sub-cubes can be coalesced is the *implication* between values of the dimensions. In our example, $C1$ *implies* $S2$, in the sense that customer $C1$ only buys products from store $S2$. Dwarf is the only technique that manages to identify such whole sub-cubes as redundant and coalesce the redundancy from *both* storage and computation time, *without* calculating any redundant sub-cubes. For comparison, the condensed cube[WLFY02] can only identify redundant areas only tuple-by-tuple, and QC-Trees[LPZ03] have to compute first all possible sub-cubes and then check if coalescing can occur.

Such suffix redundancies demonstrate that there is significant overlap over the aggregates of different groupings. The number of tuples of the coalesced cube, where

coalesced areas are only store once is much smaller than the size of the cube, which replicates such areas over different groupings.

DEFINITION 2 *The size of a cube is the sum of the tuples of all its views. The size of a coalesced cube is the total number of tuples after the coalescing operation.*

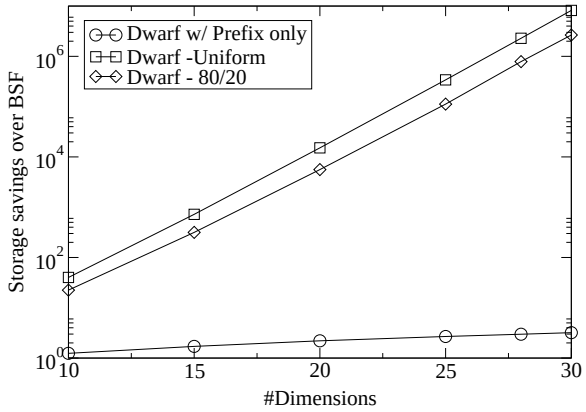


Figure 4.2: Compression vs. Dimensionality

Prefix redundancy works in harmony with suffix redundancy by eliminating common prefixes of coalesced areas. A comparison between these redundancies is demonstrated in Figure 4.2, where we depict the compression ratio achieved by storing all the tuples of a cube exploiting in the first case just the prefix redundancies and in the second both prefix and suffix redundancies w.r.t to the dimensionality of the dataset. We used a dataset with a varying number of dimensions, a cardinality of 10,000 for each dimension and a uniform fact table of 200,000 tuples. It is obvious that in high-dimensional datasets the amount of suffix redundancies is many orders of magnitudes more important the prefix redundancies.

4.3 Coalescing Categories

In this section we categorize suffix redundancies in *sparsity* and *implication* redundancies. We use the Dwarf model[SDRK02] -a summary is in the appendix- in order to ease the definition and visualization of the redundancies. In the rest of the chapter we will use this visualization, but our approach can be applied to other coalesced cube approaches[LPZ03, WLFY02].

4.3.1 Sparsity Coalescing

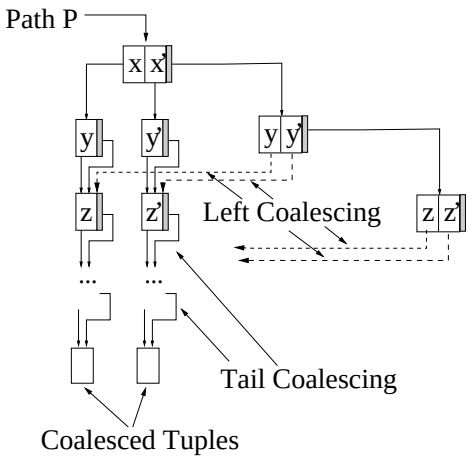


Figure 4.3: Sparsity Coalescings

In Figure 4.3 we depict two types of suffix redundancies due to the sparsity of the dataset. Lets assume that a path $\langle P \rangle$ leads to a sparse area and that for the paths $\langle P\ x \rangle$ and $\langle P\ x' \rangle$ there is only one tuple due to the sparsity of the cube. We differentiate to two different types of coalescing based on the nature of the path P.

DEFINITION 3 ***Tail coalescing*** *happens on all groupings that have* $\langle P\ x \rangle$ *as*

a prefix, where path $\langle P\ x \rangle$ *leads to a sub-cube with only one fact tuple and path* P ***does not follow any ALL pointers****.*

EXAMPLE 3 *In Figure 4.3, since there is only one tuple in the area* $\langle P\ x \ldots \rangle$ *then all the group-bys that have* $\langle P\ x \rangle$ *as a prefix (i.e.* $\langle P\ x\ ALL\ z \ldots \rangle$, $\langle P\ x\ y\ ALL \ldots \rangle$ *etc.) share the same aggregate.*

DEFINITION 4 ***Left coalescing*** *occurs on all groupings with prefix* $\langle P\ ALL\ y \rangle$*, where path* $\langle P\ ALL\ y \rangle$ *leads to a sub-cube with only one tuple. In this case,* P *follows* ***at least one*** *ALL pointer.*

EXAMPLE 4 *Left coalescing complements tail coalescing and in Figure 4.3 we depict the case where* $\langle P\ ALL\ y \ldots \rangle$ *is redundant and corresponds to* $\langle P\ x\ y \ldots \rangle$*. Tha same is observed for* $\langle P\ ALL\ ALL\ z \rangle$ *and* $\langle P\ ALL\ ALL\ z' \rangle$*.*

Areas with just one tuple (like $\langle P\ xy \rangle$ and $\langle P\ x'y' \rangle$) therefore produce a large number of redundancies in the structure of the cube. The difference between tail and left coalescing is two-fold:

- Paths the lead to tail coalescing *do not follow* any *ALL* pointers while in left coalescing the paths follow at least one *ALL* pointer -the one immediately above the point where coalescing happens-.
- Tail coalescing introduces one coalesced tuple in the coalesced cube, while left introduces no coalesced tuples.

In our analysis we consider these two types of coalescing (tail and left) and we show that their effect is so overwhelming that the exponential nature of the cube reduces into polynomial.

4.3.2 Implication Coalescing

The sparsity-coalescing types defined in Section 4.3.1 work only in sparse areas of the cube where a single tuple exists. The *implication-coalescing* complements these redundancies by coalescing *whole sub-cubes*. For example, for the fact table in Table 4.1 we observe that $C1$ implies $S2$ -in the sense that customer $C1$ only buys products from $S2$. This fact means that *every* grouping that involves $C1$ and $S2$ is essentially exactly the same with the groupings that involve $C1$. This redundancy can be depicted in Figure 4.4

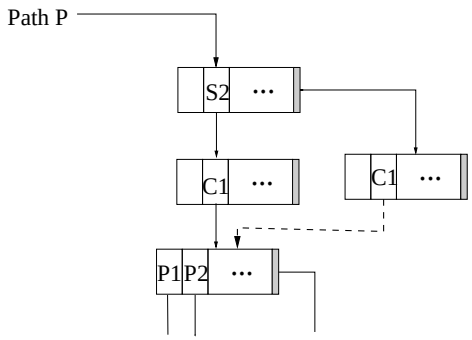


Figure 4.4: Implication Coalescing, where $C1 \rightarrow S2$

The implication coalescing is the generalization of left-coalescing when implications between the values of dimension occur. Such implications are very apparent in real datasets and -since we do not consider those in our analysis- they are the reason that in the experiments section we *overestimate* the size of the coalesced cube for real data sets.

4.4 Basic Partitioned Node Framework

In this section we formulate the coalesced cube structure by first introducing the *basic partitioned node* and then by building the rest of the coalesced cube around it -by taking into account both tail and left coalescing-. Although in this chapter we focus on uniform datasets our framework is applicable to more general distributions by properly adjusting the probability that is used in lemma 2.

Assume a uniform fact table with d dimensions, where each dimension has a cardinality of $C = L!$ and that there are $T = C$ tuples. The root node of the corresponding coalesced cube is depicted in Figure 4.5, where the node has been partitioned[3] into L groups. We refer to such a node as the *basic partitioned node*. Group G_0 contains cells that get no tuples at all, group G_1 contains cells that get exactly one tuple, group G_2 contains cells that each one gets exactly two tuples, etc.

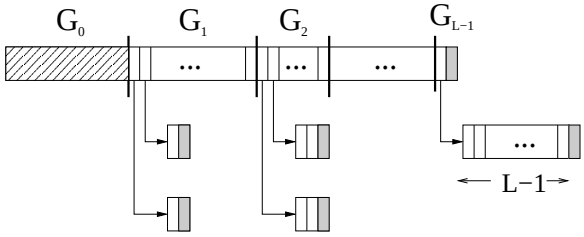


Figure 4.5: Basic Partitioned Node; Group G_z gets exactly z tuples

Lemma 2 *From a collection of C items, if we uniformly pick an item and repeat T times, then the probability that we pick one item exactly z times is:*

$$P_z(C,T) = \frac{\binom{T}{z}}{(C-1)^z} e^{-T/C}$$

[3]for this analysis we relax the property of the dwarf, where the cells inside a node are lexicographically sorted

[Proof: The probability that we will pick one item exactly z times is:

$$P_z(C,T) = \binom{T}{z} 1/C^z (1-1/C)^{T-z} =$$
$$= \binom{T}{z} 1/C^z (C-1)^{-z}/C^{-z}(1-1/C)^T$$

where the quantity $(1-1/C)^T$ can be approximated by $e^{-T/C}$ and the binomial $\binom{T}{z}$ corresponds to the number of different ways the product $1/C^z(1-1/C)^{T-z}$ can be written.]

By applying lemma 2 to the basic partitioned node we get by substituting $T = C$:

Lemma 3 *A group G_z of a basic partitioned node, where $z = 0 \ldots L-1$, contains $\approx \frac{C}{z!}e^{-1}$ cells that get exactly z tuples each*

[Proof: The expected number of cells inside a group G_z is:

$$CP_z(C,C) = C\frac{\binom{C}{z}}{(C-1)^z}e^{-1} \approx \frac{C}{z!}e^{-1}$$

because $z \ll C$ (z is at most $L-1$, where $C = L!$).]

In Figure 4.5 we depict that the dominated nodes of a group G_z have exactly z cells. From lemma 3 we know that exactly z tuples are associated with each cell of group G_z and from the independence assumption we have that the probability that a key is duplicated for these tuples is $1/C^2$ with an expected number of duplicated keys z/C^2. Even for $z = L$, we expect $L/(L!)^2 \ll 1$ duplicate cells.

4.4.1 Left Coalesced Areas

In this section we deal with areas of the coalesced cube that are reachable through paths that follow ALL pointers. These areas have the possibility of left coalescing and as we'll show they are dominated by such redundancies.

In Figure 4.6 we show a basic partitioned node which corresponds to a path P that follows at least one ALL pointer and that it corresponds to a subset of the fact table with $T = C$ tuples. We refer to the corresponding sub-cube as *left-coalesced sub-cube* and we show that it introduces a "small" number of new coalesced tuples. Obviously cells in group G_0 that get no tuples offer no tuples at all. Cells in group G_1 that get only a single tuple, left-coalesce to other tuples in the structure and offer no aggregation. This is the reason we differentiate between paths that follow at least one ALL pointer and those which do not. Cells in groups $G_2, G_3, \ldots, G_{L-1}$ introduce only a single aggregate per cell.

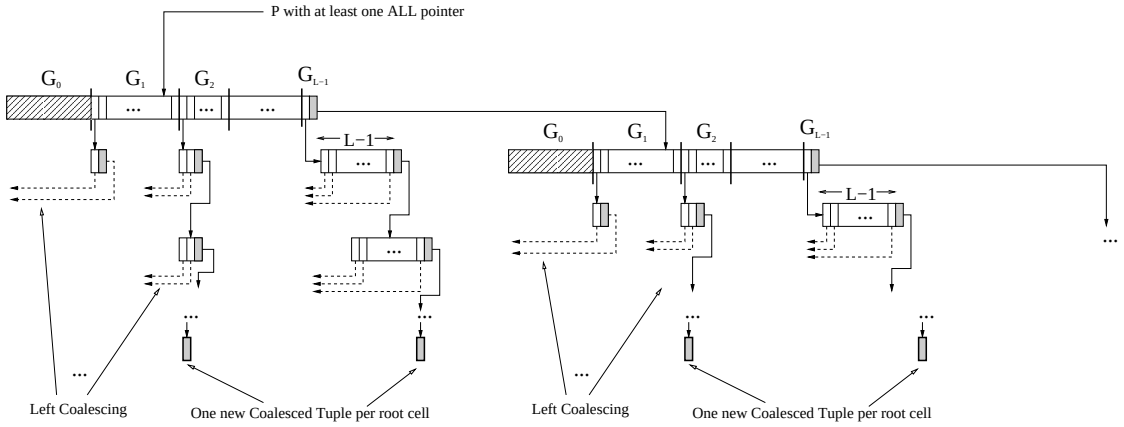


Figure 4.6: Left-Coalesced partitioned node with $T = C$

To help clarify this, consider a cell in group G_2. Since there are two fact tuples associated with this cell (by definition) there are two paths $\langle P\ x\ \rangle$ and $\langle P\ x'\ \rangle$ that

correspond to these two tuples. Since the path P follows at least one ALL pointer, the *exact same tuples* appear with another path Q that does not follow any ALL pointer, and therefore paths $\langle P\ x\ \rangle$ and $\langle P\ x'\ \rangle$ coalesce to $\langle Q\ x\ \rangle$ and $\langle Q\ x'\ \rangle$. The only aggregate that this sub-cube introduces is the aggregate of these two tuples (located at the leaf nodes). The same holds for all groups $G_2, G_3, \ldots, G_{L-1}$ and therefore the number of new coalesced tuples that a left-coalesced sub-cube with d dimensions and $T = C$ fact tuples introduces is (by using lemma 3):

$$NLeft(T = C, d, C) = a_0 \cdot C \cdot d + 1$$

where $a_0 = (e - 2)/e$.

[Proof: As depicted in Figure 4.6 a left-coalesced partitioned node introduces:

$$d(C/2!e^{-1} + C/3!e^{-1} + \ldots) + 1 =$$

$$= Cd/e(1/2! + 1/3! + \ldots) + 1 =$$

$$= a_0 \cdot C \cdot d + 1$$

]

We can extend our analysis to the general case where $T = C^k$, $k = \log_C T$ in the way that is depicted in Figure 4.7. By induction we prove that:

Lemma 4 *The number of new coalesced tuples that a left-coalesced area introduces*

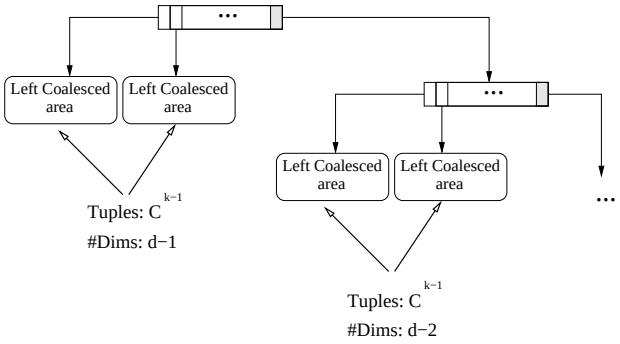


Figure 4.7: Left-Coalesced partitioned node with $T = C^k$

is:

$$NLeft(T = C^k, d, C) =$$

$$= C \cdot \sum_{i=1}^{d-1} NLeft(T = C^{k-1}, d - i, C) + 1 =$$

$$= a_0 C^k \binom{d}{k} + \sum_{i=1}^{k-1} C^{k-i} \binom{d}{k-i} + 1$$

4.4.2 Tail Coalesced Areas

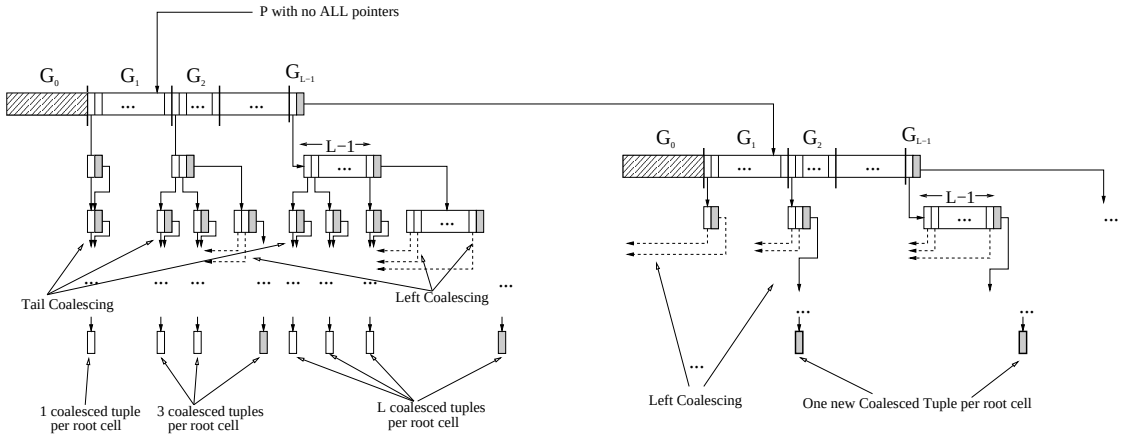


Figure 4.8: Tail-Coalesced partitioned node with $T = C$

In this section we deal with areas that are reachable through paths that do not follow any ALL pointers. These areas have less chances for left-coalescing but as will show the amount of coalescing is still very significant.

In Figure 4.8 we show a basic partitioned node which corresponds to a path P that *does not* follow any ALL pointers and that it corresponds to a subset of the fact table with $T = C$ tuples. We refer to the corresponding sub-cube as *tail-coalesced sub-cube* and we count the number of coalesced tuples it introduces. As in the left-coalesced case, cells in group G_0 that get no tuples offer no tuples at all. Cells in group G_1 that get only a single tuple, offer just a single aggregate, due to tail coalescing. Cells in groups G_z, where $z = 2, \ldots, L-1$ introduce $z+1$ coalesced tuples,the z tuples of the fact table plus their aggregation. The number of coalesced tuples a tail-coalesced sub-cube with d dimensions and $T = C$ fact tables introduces is:

$$NTail(T = C, d, C) = b_0 C + a_0 C(d-1) + 1$$

where $a_0 = (e-2)/e$ and $b_0 = (2e-2)/e$.

[Proof: The new tuples under the root tail-coalesced node (ignoring the all cell) are:

$$C/1!/e + C/3!/e + C/4!/e + \ldots = b_0 C$$

while the all cell points to a left-coalesced node with: $a_0 C(d-1)+1$ new tuples (as explained in Section 4.4.1)]

We can extend our analysis to the general case where $T = C^k$, $k = \log_C T$ in the way that is depicted in Figure 4.9. Using induction we prove that:

Lemma 5 *The number of new coalesced tuples that a left-coalesced area introduces is:*

$$NTail(T = C^k, d, C) =$$

$$= C \cdot NTail(C^{k-1}, d-1, C) + \sum_{i=2}^{d-1} NLeft(C^{k-1}, d-i, C) =$$

$$= a_0 C^k \left[\binom{d}{k} - 1 \right] + \sum_{i=1}^{k} c^{k-i} \left[\binom{d}{k-i} - 1 \right] + b_0 C^k$$

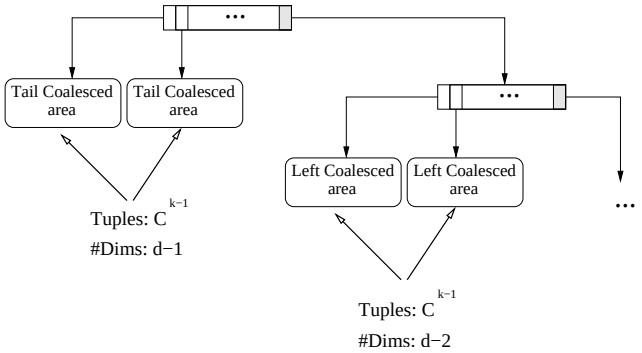


Figure 4.9: Tail-Coalesced partitioned node with $T = C^k$

4.4.3 Total Coalesced Size

The analysis for the tail coalesced areas gives the total number of coalesced tuples for the full coalesced cube with d dimensions, cardinality C per dimension and T

fact table tuples[4]. Lemma 5 gives that:

$$\#\text{CoalescedTuples} = O\left(T\frac{d^{\log_C T}}{\log_C T!}\right) = O\left(T^{1+1/\log_d C}\right)$$

with the surprising result that even if we consider only two out the three coalescings, the size of the coalesced cube is only polynomial w.r.t to the dimensionality of the fact table and polynomial (and very close to linear) w.r.t to the number of tuples in the fact table.

Additionally, if we consider the number of nodes or cells, that are introduced in the coalesced structure, the expected complexity is multiplied by d (i.e. the polynomial power increases by one), since we need *at most* d nodes and cells (ignoring any prefix reduction) in order to represent each tuple. Therefore the expected complexity for the number of cells (or the full size of the structure) is:

$$\#\text{TotalCells} = O\left(T\frac{d^{\log_C T+1}}{\log_C T!}\right)$$

Finally, the suffix coalesce algorithm described in [SDRK02] visits its coalesced tuples *at most* d times and therefore the time complexity for constructing coalesced cubes is:

$$\text{ComputationTime} = O\left(T\frac{d^{\log_C T+1}}{\log_C T!}\right)$$

[4]When we start creating the root node of the coalesced cube there is no chance of left-coalescing, since nothing has been created

4.5 Algorithm for Coalesced Cube Size Estimation

In this section we extend our analytical contribution to general case of varying cardinalities per dimensionality. Algorithm 3 can be used to estimate the number of coalesced tuples for sparse uniform data sets given the cardinalities of each dimension.

Algorithm 3 SparsityTraverse Algorithm

Input: d: Number of Dimensions
Card: array of dimension cardinalities
FactT: current no of fact tuples
nc: tail coalesce flag(0 or 1)

1: **if** FactT=0 **then**
2: return 0
3: **else if** FactT=1 **then**
4: return nc {here tail or left-coalescing happens}
5: **else if** d=0 **then**
6: return 1
7: **end if**
8: coalescedT $\leftarrow 0$
9: mC $\leftarrow$ Card[d]
10: zeroT $\leftarrow$ mC $\cdot\, e^{-\text{FactT/mC}}$
11: oneT $\leftarrow$ FactT/(mC $-$ 1) $\cdot$ zeroT
12: **if** oneT ≥ 1 **then**
13: x $\leftarrow 1$
14: **while** there are still fact tuples **do**
15: xT $\leftarrow \binom{\text{FactT}}{\text{x}}/(\text{mC}-1)^{\text{x}} \cdot \text{zeroT}$
16: coalescedT += SparsityTraverse(d-1,Card,xTuples,nc) {tail or left-coalescing may happen here}
17: FactT -= xT
18: x++
19: **end while**
20: **else**
21: coalescedT += SparsityTraverse(d-1,Card,FactT/mC,nc) {drill-down traversal}
22: **end if**
23: coalescedT += SparsityTraverse(d-1,Card,FactT,0) {roll-up traversal with left-coalescing}
24: return coalescedT

Initially the algorithm is called with the tail coalescing flag set to 1, since there is no chance for left-coalescing (there are no tuples to coalesce to). In line 4

we check if there is just one tuple in the subcube where tail or suffix coalescing happens depending on the tail coalescing flag. In lines 12- 19 we traverse the basic partitioned node by checking iteratively how many cells get one, two, three, ... tuples until all the available tuples for the subcube are exhausted. The quantity:

$$\frac{\binom{\mathrm{FactT}}{\mathrm{x}}}{(\mathrm{mC}-1)^{\mathrm{x}}} \cdot \mathrm{mC} \cdot e^{-\mathrm{FactT/mC}}$$

where FactT is the number of fact tuples for the current subdwarf and mC is the cardinality of the current dimension, returns the number of cells that get exactly x tuples

The algorithm works in a depth-first manner over the lattice and estimates recursively the number of coalesced tuples that its sub-dwarf generates. For example, for a three-dimensional cube *abc*, the algorithm in line 21 starts the *drill-down* to all subcubes with prefix *a* and recursively it proceeds to those with prefix *ab* and finally reaches prefixes *abc*, by estimating appropriately the number of tuples that each subdwarf gets. When (lines 1-7) there are no more dimensions to drill-down (or a tail or left coalescing can be identified), the drill-down over the subdwarfs with prefixes in *abc* stops and the algorithm *rolls-up* to the subdwarfs with prefixes *ab* in line 23 by setting the nC flag to 0 -since now there is possibility of left-coalescing with the subcubes in *abc*-. The process continues recursively to all the views of the lattice.

The running complexity of the algorithm is derived from the basic partitioned node framework and is polynomial on the number of dimensions. It also requires

minimal memory enough to accommodate the stack for performing a DFS to d dimensions deep.

4.6 Experiments

In this section we provide an extensive experimental evaluation of our approach based on synthetic and real data sets. We compare the results of our analytical approach with actual results taken from our implementation of Dwarf. The experiments were executed on a Pentium 4, clocked at 1.8GHz with 1GB memory. The buffer manager of our implementation was set to 256MB.

4.6.1 Synthetic Datasets

In this section we use the following formalism. The graph entitled "Actual" in the legend corresponds to numbers taken from our implementation, while the graph entitled "Estim" corresponds to the estimates our analytical framework and algorithm provides. We use the symbol d to refer to the number of dimensions, C to the cardinality and a to the zipfian parameter (skewness).

Scalability vs dimensionality

Uniform Distributions In Figure 4.10 we demonstrate how the number of coalesced tuples scales w.r.t to the dimensionality, for a uniform dataset. The number of fact table tuples was set to 100,000. We used two different cardinalities of 1,000 and 10,000. We see that our analytical approach provides extremely accurate re-

sults for large cardinalities.The reason that the error decreases as the cardinality increases is the approximation in lemma 3, where we assume that $C - 1 \approx C$. The second observation has to do with the scalability w.r.t. to the dimensionality. The quantity $\log_C T$ which determines the exponent of d is much smaller in the case of $C = 10,000$ and therefore this data set scales better.

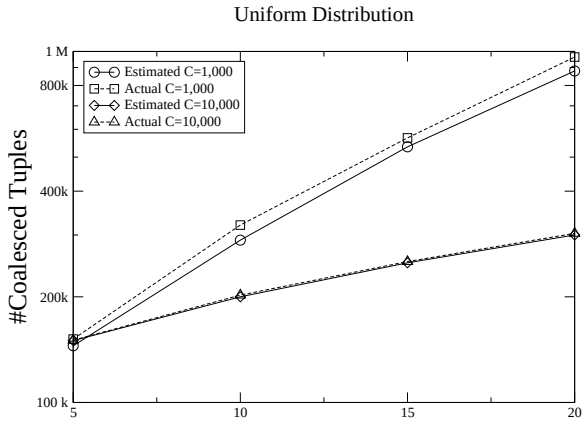


Figure 4.10: Size v.s. #dims, varying cardinalities (uniform)

In Figure 4.11 we depict the time scalability –w.r.t to the dimensionality– required to compute and store the coalesced cubes using the Dwarf approach for the uniform datasets. We must point that the y-axis are logarithmic and that the graphs –for both #coalesced tuples and computation time– correspond to a polynomial scaling.

Zipfian Distributions In Figure 4.12 we depict the size scalability w.r.t to the dimensionality for zipfian datasets for various values for the zipfian parameter a that controls the skew. The number of fact table tuples was set to 100,000. The cardinalities were again 1,000 and 10,000 respectively. We observe that our estimation algorithm approximates better the zipfian coalesced cube size for large

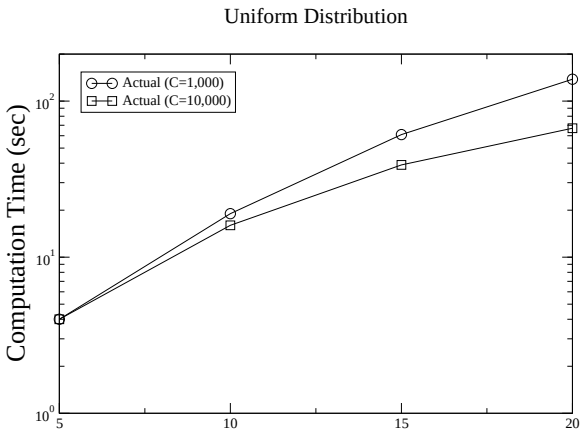


Figure 4.11: Time v.s #dims, varying cardinalities (uniform)

values of cardinalities than it does for smaller values of cardinalities[5]. On the other side, we observe that the skew parameter affects more the dataset with $C = 1,000$ than the dataset with $C = 10,000$. The reason for these two observations is that the zipfian parameter directly affects the sparsity of the cube. For lower values of cardinalities the percentage of sparsity coalescings is significantly less than the case of higher cardinality values. However it is evident that the zipfian distribution scales polynomially and that our estimation algorithm can be used to get good estimates about zipfian coalesced cubes. We must point out that from the graphs it can be derived that the zipfian distribution affects the scalability –w.r.t to the dimensionality– in a multiplicative way. In other words, it increases the complexity factor but not the polynomial power.

In Figure 4.13 we depict that the scalability of the required computation time for varying dimensionalities, cardinalities and skew parameters is again polynomial. We observe that the skew parameter affects proportionally the computation time as it affects the coalesced cube size.

[5]This behavior is observed (to a lesser degree) for uniform datasets as well

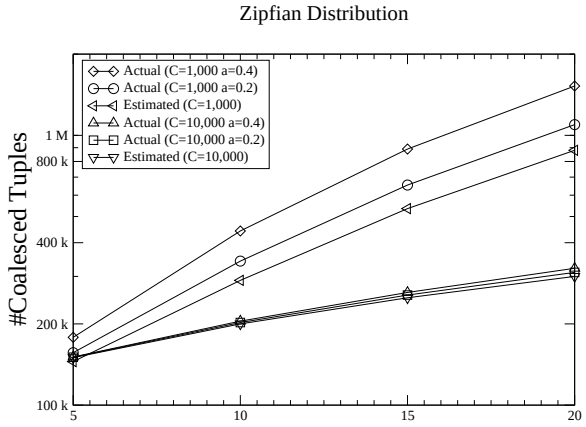


Figure 4.12: Size v.s. #dims, varying cardinalities & zipf parameters

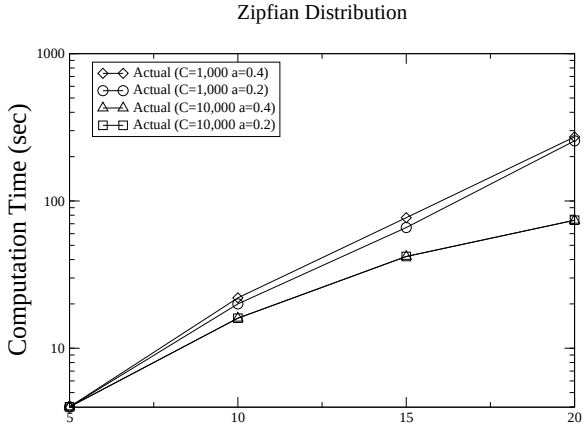


Figure 4.13: Time v.s. #dims, varying cardinalities & zipf parameters

Scalability vs #Tuples In Figures 4.14, 4.15 and 4.16 we depict the coalesced size scalability w.r.t to the number of tuples for uniform and Zipfian datasets for a variable number of dimensions, cardinalities and skew. We observe that in all cases both the number of coalesced tuples and the computation time scale almost linearly w.r.t to the number of tuples in the fact table. We must point that a value $C = 10,000$ for the cardinality offers more chances for sparsity coalescing and therefore the required storage and time is lower than the case of $C = 1,000$. The skewness of the zipfian distributions affects sparsity coalescing in a negative way and increases the corresponding coalesced cube size and computation time. For

completeness we also depict the required computation time for the same cubes in Figures 4.14 and 4.15.

In this series of experiments our estimation algorithm, although based on a uniform assumption, provides very accurate results over all the range of the parameters (cardinality, number of dimensions, skewness) that we experimented on.

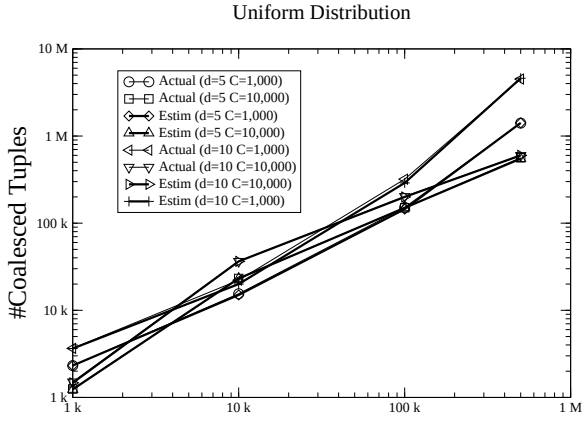


Figure 4.14: Size v.s. #Tuples, varying cardinalities (uniform)

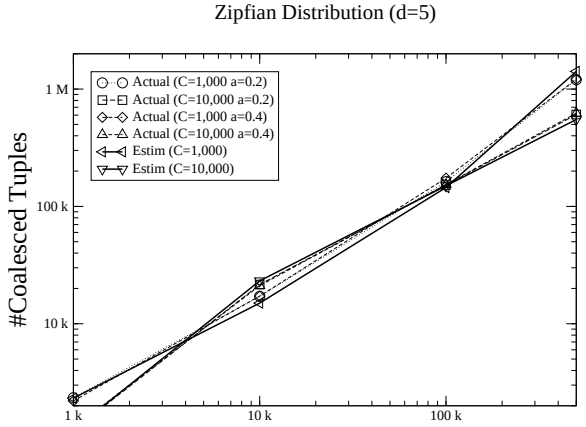


Figure 4.15: Size v.s. #Tuples, varying cardinalities & zipf parameters

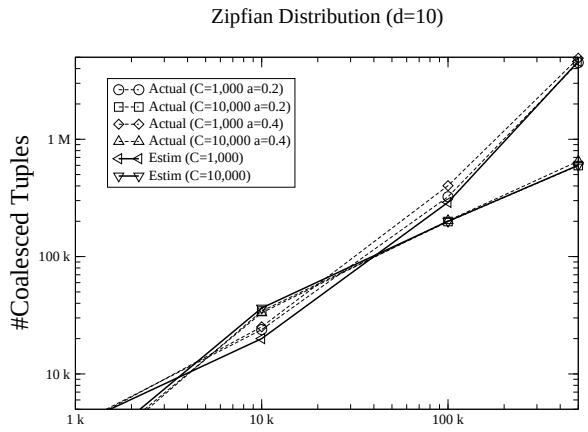


Figure 4.16: Size v.s. #Tuples, varying cardinalities & zipf parameters

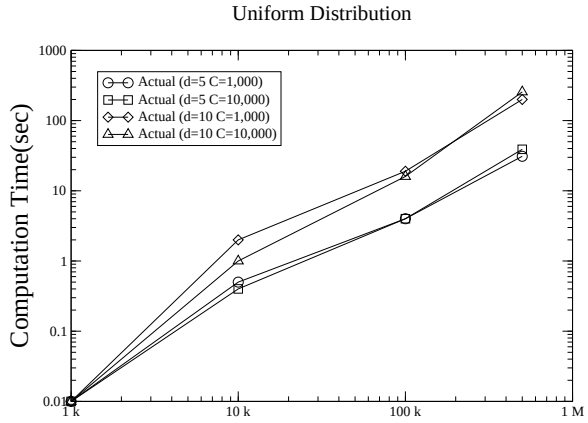


Figure 4.17: Time v.s. #Tuples, varying cardinalities (uniform)

4.6.2 Real Datasets

For this experiment we use a real eight-dimensional data set given to us by an OLAP company. The data set has varying cardinalities per dimension. We used various projections on the data set in order to decrease the dimensionality and study its

Projection	d	Cardinalities
A	5	1300,2307,2,2,3098
B	6	1300,2307,3098,130,561,693
C	7	1300,2307,2,3098,130,561,693
D	8	1300,2307,2,2,3098,130,561,693

Table 4.3: Real data set parameters

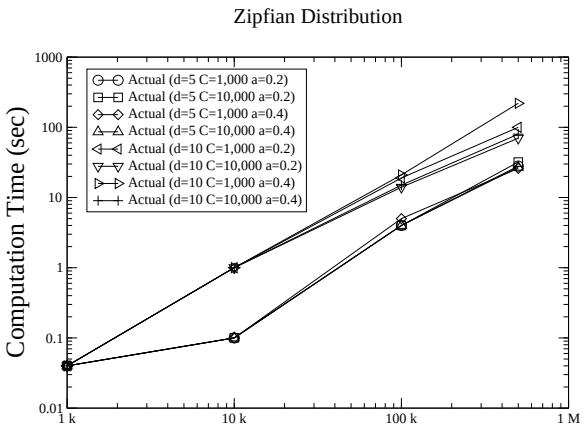


Figure 4.18: Time v.s. #Tuples, varying cardinalities & zipf parameters

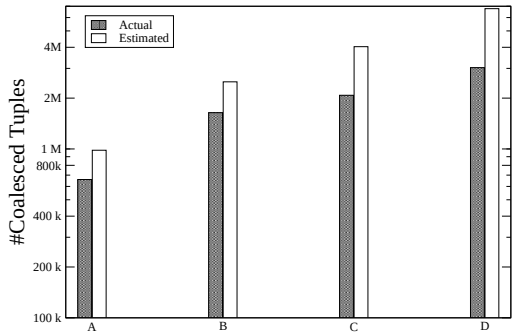


Figure 4.19: Size Scalability v.s. dimensionality for real data set

effect on the accuracy. For this experiment the fact table had 672,771 tuples and two measures. Table 4.3 summarizes the parameters of each projection. Column "Projection" denotes the name of the data set, column d the number of dimensions and column "Cardinalities" the cardinalities of each dimension. In Figure 4.19 we depict the estimates of our approach compared with the actual numbers taken, when the dwarf is computed and stored. In Figure 4.20 we depict –for completeness– the time scalability w.r.t the dimensionality of the real datasets.

Our approach *overestimates* increasingly more the coalesced size. The rea-

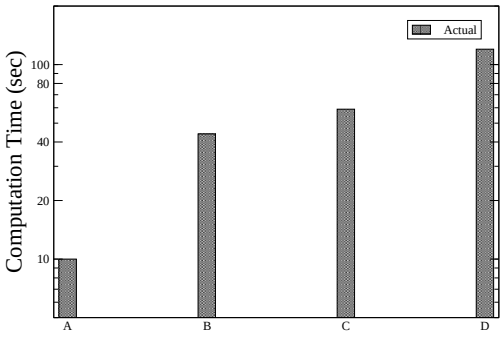


Figure 4.20: Time Scalability v.s. dimensionality for real data set

son is that our approach currently handles *only sparsity coalescing* and ignores the *implication coalescing* that is very apparent in high-dimensional data sets. As the dimensionality increases such implications increase and complement the sparsity implications reducing even further the coalesced size. This observation is in contrast to what happens with zipfian datasets, which affect the sparsity of the coalesced cube in a negative way without however creating any implications between the dimensions. However real datasets are not only skewed but present a large number of implications between values of their dimensions.

4.7 Summary

We have presented an analytical and algorithmic framework for estimating the size of coalesced cubes, where suffix redundancies diminish the number of aggregates that need to be stored and calculated. Our analytical framework although it uses only sparsity coalescing, derives the surprising result, that a uniform coalesced cube grows polynomially w.r.t to the dimensionality. This result changes the establish

state that the cube is inherently exponential on the number of dimensions and extend the applicability of data warehousing methods to a much wider area. We were also able to device an efficient algorithm for estimating the size of a coalesced cube based only its dimensions' cardinalities and demonstrated that it provides accurate results for a wide range of distributions. In addition we have demonstrated –using real data– that real coalesced cubes scale *even better* than our analysis derives. The reason is that the effects of implication coalescing complement the results of sparsity coalescing that we have presented here.

Chapter 5

Hierarchical Dwarfs (CRC)

5.1 Introduction

The introduction of the data cube operator has been both a blessing and a curse for the analysis of large datasets using On Line Analytical Processing (OLAP) applications. It provides the means for the succinct formulation of query primitives that are fundamental in OLAP, including histograms (aggregation over computed categories), roll-up and drill-down operations and cross-tabulation. The data cube operator has formalized the concepts of multidimensional aggregate views and the hierarchies within them.

Unfortunately, the expressive power unleashed by the data cube comes at a big cost. The number of views in the data cube increases exponentially with the number of dimensions. As a result, a naive computation and storage of all the views was deemed non-feasible but for toy-like data sets. Following the seminal paper of [GBLP96] there has been a flurry of literature for handling the alarming complexity of the data cube by pre-computing a subset of all possible views [GHRU97b, HRU96b, TS97], providing approximate answers using a lossy representation of the data cube [AGP00, VWI98], or by using some form of online aggregation [HHW97].

All these techniques, albeit their novelty, leave a taste of defeat; instead of

attacking the problem, they rather circumvent it. Two recent proposals, namely Dwarf [SDRK02] and QC-tree [LPZ03] have been shown to handle large data cubes by exploiting the inherit redundancy of their structure. A lot of the aggregates in the data cube, especially on sparse datasets, are computed by the same set of tuples. Dwarf, identifies these aggregates through a process called *suffix coalescing*, while QC-tree uses an analogous notion of *cover-equivalent classes*. The elimination of multiplicities of the aggregates in the data cube has a dramatic effect in its size; the authors of [SDRK02] have reported storing a petabyte data cube in just 2.3GB of disk space.

The techniques of [SDRK02, LPZ03] have paved the way for managing large data cubes by compressing their canonical structure but are both incomplete, as they are limited to aggregates computed directly on the raw data. Nevertheless, in most data warehouse applications, dimensional values are further annotated with hierarchies. Their importance is outlined in [JLS99] and their presence has a profound effect on the size of the data cube; in-fact it increases its computational and storage complexity exponentially.[1] One may handle hierarchies *externally* to the storage engine. First the data cube of the lowest hierarchy levels for each dimension is computed. Roll-up or drill-down hierarchical queries are then mapped into queries on the raw data cube and their results are further aggregated to the proper level of each hierarchy in a second step. This practice however is very costly, especially for queries at the higher levels of a hierarchy. Such queries are very common during

[1] A D-dimensional data cube has 2^D aggregate views. When each dimension has a hierarchy with L levels, the number of views increases to $(L+1)^D$ i.e. by an exponential factor of $(\frac{L+1}{2})^D$.

exploratory data analysis. For instance, in basket data analysis, one first aggregates sales on a yearly or monthly basis and then drills-down to particular weeks or days of interest. One, thus, would like to be able to support these queries directly without the post-processing cost.

In this chapter we present a new framework called "Coalesced Rollup Cubes" (CRC) for managing rollup data cubes; i.e. data cubes computed over all the dimension hierarchies. In CRC, not only are structural redundancies[SDRK02] eliminated across all the hierarchical levels, but, most importantly, the amount of materialization is controlled through a novel "knob" method that requires a very small time and storage overhead compared to flat (no hierarchies) cubes, while at the same time significantly improving query performance.

The knob materialization targets dense areas of hierarchical aggregates that are common in rollup cubes, for instance at the top levels of each hierarchy. Unlike view selection algorithms that work with a-priori estimates of the size of the aggregates, in CRC the selection of which rollup aggregates to materialize is performed in an online fashion using the data at-hand. The user controls the amount of materialization by specifying the maximum number of rollup aggregates that can be aggregated on the fly for a given query. What is important is that this selection process is interleaved with the construction algorithm. As a result, we retain the key benefit of the interleaved top-down/bottom-up computation, namely the ability to prune a lot of aggregates prior to their computation.

In our experiments we demonstrate that the online selection algorithm results in highly compressed rollup cubes that are very fast to compute. Furthermore, the

reduction of the size of CRC has a similar effect on query performance. A common misconception is that full materialization of the data cube results in optimal performance. In practice there is an optimal level of materialization that delivers best query performance, after which additional data materialization results in slower response times due to congestion in memory buffers and lack of locality. In CRC, the selection of the rollup aggregates to materialize is performed based on the I/O cost of consolidating finer-grained aggregates on the fly for a user query. This results in an extremely compact structure which exhibits better access locality during queries. In one of our experiments, compared to materializing the full rollup cube, we observed more than six times speedup in creation (768sec vs 4860sec) and eight times reduction in storage (806MB vs 6.6GB) while at the same time query performance increased by a factor of two(191sec vs 282sec) –for 1,000 queries–. Compared to the Dwarf techniques that materialize the raw data cube and handle hierarchical queries externally, the optimized CRC structure is about seven times faster during queries(191sec vs 1331sec) –for 1,000 queries–, while requiring less than 1% more time and just 5% more space to compute, although it handled $\approx$ 44 times more views (11,200 vs 256).

5.2 Framework

We first formally define the properties of our Coalesced Rollup Cube structure (CRC) and then present a small example of a CRC structure. Finally we present in detail the notion of the knob parameter in our CRC structure, and how it is

StoreId	Code	Name	Sales
S1	C2	N1	$10
S2	C3	N2	$30
S3	C1	N1	$60

Table 5.1: Fact Table w/ hierarchies

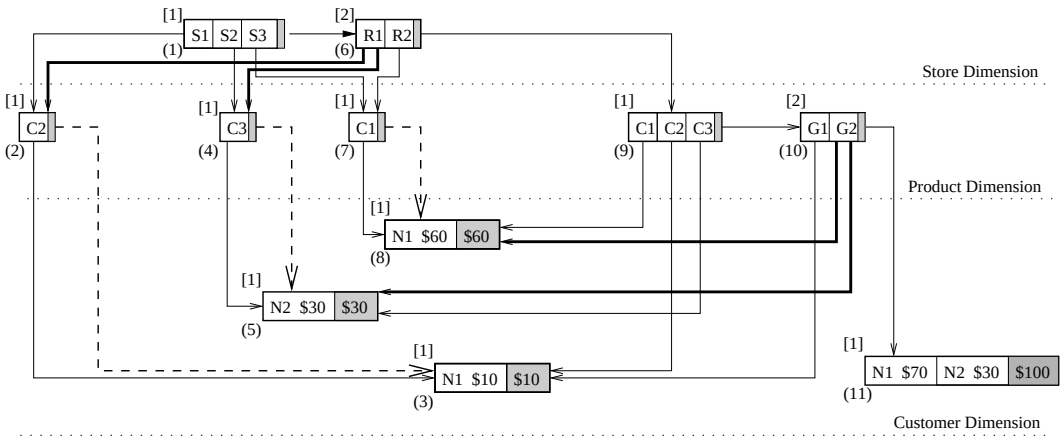


Figure 5.1: CRC example with knob=2 & Hierarchical Pruning

calculated. In our discussion throughout the chapter we will commonly point to Figure 5.1, which depicts the CRC for the fact table of Table 5.2 and the hierarchies/metadata of Table 5.2.

5.2.1 Properties of CRC

The CRC data structure for storing rollup cubes has the following properties. It is a directed acyclic graph (DAG) with just one root node and has exactly D levels, where D is the number of the cube's dimensions. Each level i is conceptually (only) partitioned into L_i fragments, where L_i is the number of hierarchy levels of dimension i. The fragments are considered ordered based on the hierarchy level they correspond to, starting from the most detailed level and moving towards the least detailed

level of the hierarchy. Nodes at the D-th level (*leaf nodes*) contain cells of the form: [*key*, *aggregateValues*], while nodes in levels other that the D-th level (*non-leaf nodes*) contain cells of the form: [*key*, *pointer+*]. Cells containing more than one pointers are termed as *split* cells, while cells containing a single pointer are termed as *normal* cells. Split cells may only appear in nodes that do not belong to the first (most detailed) fragment of a level. Thus, their presence is limited to dimensions which contain more than one hierarchy levels. Based on the above discussion, we define as the *content* of a cell C_i, belonging to a node N, to be either the aggregateValues of C_i or the set of pointers of C_i, depending on whether C_i stores a pointer, a set of pointers or aggregateValues.

Hierarchies			Declared Metadata	
Store	Product	Customer	Dimension	Metadata
ALL ↑ Retailer ↑ StoreId	ALL ↑ Group ↑ Code	ALL ↑ Name	Store Store Store Product Product Product Customer Customer	S1 → R1 S2 → R1 S3 → R2 C1 → G2 C2 → G1 C3 → G2 N1 N2

Table 5.2: Example of declared Hierarchies

A normal cell in a non-leaf node of level i points to a node, which it *dominates*, at the first fragment of level $i + 1$. A split cell in a non-leaf node of level i points to a set (of size between 2 and the value of the knob parameter) of nodes, which it *dominates*, at the first fragment of level $i + 1$. Each node also contains a *special cell*, which corresponds to the cell with the pseudo-value ALL as its key. For nodes that contain more than one cells, their special cell contains either a pointer to a

node in the next fragment of the node's level, if the node does not belong to the last fragment of its level, or, otherwise, either a pointer to a node at the first fragment of level $i + 1$, if this is a non-leaf node, or aggregateValues, if this is a leaf node.

Hierarhical Pruning If the node contains a single cell, then the *content* (see definition above) of the ALL cell is identical to the one of that single cell. In this case, some hierarchy levels are "shortcut" (see the dashed pointers in Figure 5.1) and not materialized, in a process called *hierarchical pruning*. The reasoning behind hierarchical pruning is that the materialization of the remaining, less-detailed levels does not perform any aggregation and would only introduce redundancy in the structure. Hierarchical pruning may occur at any level of the CRC structure.

Cells belonging to nodes at fragment j of the i-th level of the structure contain keys that are values of the cube's i-th dimension and which correspond to the j-th hierarchy level of this dimension. No two cells within the same node contain the same key value. Each cell C_i at the i-th level of the structure, corresponds to the sequence S_i of $i \leq |S_i| \leq \sum_{j=1}^{i} L_j$ keys found in a path from the root to the cell's key. This sequence corresponds to a group-by with the last $(D - i)$ dimensions unspecified. All group-bys having sequence S_i as their prefix, will correspond to cells that are descendants of C_i in the CRC structure. For all these group-bys, their common prefix will be stored exactly once in the structure (prefix reduction).

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. This happens, when the exact same tuples contribute the same aggregates to

different group-bys ([SDRK02]). In such a case, the coalesced node will be reachable through more than one paths from the root, all of which will share a common suffix. For example, in node 3 at the bottom of the Customer level of Figure 5.1, the first cell of the node corresponds to the sequences (among others) $\langle S1, C2, N1\rangle$ and $\langle ALL, ALL, C2, N1\rangle$, which share the common suffix $\langle C2, N1\rangle$. If a node N is a coalesced node, then any node X which is a descendant of N will also be a coalesced node, since it can be reached through multiple paths from the root.

A traversal in the CRC structure follows a path of length at least D and at most $\sum_i L_i$ (where L_i is the number of hierarchy levels of dimension i) starting from the root to a leaf node. For each dimension i, the path contains at most L_i ALL values, if the dimension is left unspecified (fewer than L_i ALL values may occur in this case due to the hierarchical pruning process). If a value V at the j-th fragment is specified, in the absence of hierarchical pruning in this path at this level, the path for the i-th dimension contains $j-1$ ALL values, followed by the V value. In the presence of hierarchical pruning at this path and in this level, this path for the i-th dimension contains at most $j-1$ ALL values, the last of which is the ALL cell of a node containing a single cell C, and where V is an ancestor of C in the hierarchy of the i-th dimension. The CRC structure itself constitutes an efficient inter-level indexing method and requires no additional external indexing.

We now define some terms which will help in the description of the algorithms. The *cube of a node* N is defined to be the node itself and all the cubes of the nodes dominated by the cells of N. The cube of a node X that is dominated by some cell of N is called a *sub-cube* of N. Since leaf node cells at the last fragment dominate

no other nodes, the cube of such a node is the node itself.

5.2.2 A Simple CRC Example

We now explain the properties of the CRC structure using the sample CRC of Figure 5.1.

Levels, Fragments and Cells. The CRC of Figure 5.1 is a rollup cube using the aggregate function *sum* and containing a total of $(2+1) \times (2+1) \times (1+1) = 18$ views. The nodes are numbered (inside parentheses, located at the lowest left part of each node) according to the order of their creation. Each node also contains another number inside brackets, which is related to the knob value of the node, and which will be explained in Section 5.2.3. The height of the CRC structure is equal to 3, one for each of the *Store, Product* and *Customer* dimensions, which contain 2, 2 and 1 hierarchy levels, respectively.[2] The root node (node 1) contains cells of the form $[\text{key}, \text{pointer}]$, one for each distinct value of the first dimension at its most detailed (bottom-most) hierarchy level. The pointer of each cell points to a node on the next level containing all the distinct values of the next dimension that are associated with the cell's key. In the data of Table 5.2 the product with Code $C2$ is the only product associated with $S1$ (first tuple). Since the cell $S1$ of the root contains a pointer to node 2, $S1$ dominates node 2. Each node that does not correspond to the least detailed level of the last dimension (lowest level in CRC) has a special ALL cell, shown as a small gray area to the right of the node, holding a pointer and

[2]The top-most ALL levels of Figure 5.1 are not counted in this discussion as they are directly computed in the structure; the rest of the levels pose new challenges.

corresponding to all the values of the node. For example, node 1 that corresponds to the *StoreId* level of the *Store* hierarchy has an ALL cell that points to node 6, which corresponds to the *Retailer* level. Moreover, since node 6 corresponds to the top-most level of its hierarchy (*Retailer* level), its ALL cell in Figure 5.1 points to node 9 (corresponding to the *Code* hierarchy level for the *Product* dimension). In this figure two split cells exist, namely cells with keys $R1$ and $G2$ in nodes 6 and 10, respectively. The pointers of these split nodes are marked as bold, to easier distinguish them from the other pointers in the figure.

Paths, Queries and Hierarchical Pruning. The path $\langle ALL, R2, C1, N1\rangle$ leads to the cell [$N1$ \$60] which contains the aggregate value of the sales of product $C1$ that Customer $N1$ has bought from stores supplied by retailer $R2$. $\langle ALL,\ ALL,\ ALL,\ ALL,\ ALL\rangle$ in our example leads to the total sales (group-by NONE) of \$100 (the ALL cell of node 11). Moreover, the path $\langle S1, C2, N1\rangle$ leads to the cell [$N1$ \$10] of node 3, and corresponds to the sum of the prices paid by customer $N1$ for product $C2$ at store $S1$. Note that the same path would have been followed if we wanted to find the sum of prices paid by customer $N1$ for products of the group $G1$ at store $S1$, due to the process of hierarchical pruning. Nodes 2, 4 and 7, which all correspond to the Product dimension and contain a single cell, have their ALL pointers "shortcut" to the dimension level *Customer* below by ignoring the remaining hierarchy level (the Group level) of the dimension.

The reader can observe that the four paths $\langle S1, C2, N1\rangle$, $\langle ALL, ALL, C2, N1\rangle$, $\langle ALL, R1, C2, N1\rangle$, and $\langle ALL, ALL, ALL, G1, N1\rangle$, the two hierarchically pruned

paths $\langle S1, ALL, G1, N1\rangle$, $\langle S1, ALL, ALL, N1\rangle$ (answered by the path $\langle S1, ALL, N1\rangle$) and the hierarchically pruned path $\langle ALL, R1, ALL, G1, N1\rangle$ (answered by the path $\langle ALL, R1, ALL, N1\rangle$), whose values are extracted from processing just the first tuple of the fact-table, all lead to the same cell [$N1 \$10] of node 3, which, if stored in different nodes, would introduce *suffix redundancies*. By *coalescing* these nodes, we avoid such redundancies.

5.2.3 Knob Materialization

In huge real data sets, we expect that the cost to fully materialize the rollup cube will be prohibitive. We therefore need to develop techniques that effectively reduce the amount of materialization performed, and therefore the size of CRC, without sacrificing, if possible, query performance.

The work in [SDRK02] proposed the use of a *granularity* parameter G_{min} as a solution to effectively reduce the size of its Dwarf structure. This technique is heavily utilized in the lower parts of its Dwarf structure, where the data is typically very sparse, due to their proposed dimension ordering (order dimensions in decreasing cardinalities). The Dwarf algorithms calculate and store the content (sub-cubes) of cells only when this calculation aggregates at least G_{min} tuples.

We propose a technique that controls the amount of performed materialization termed knob, which is orthogonal to the G_{min} technique, and is used in dense hierarchical areas of the rollup cube. Typically, higher hierarchical levels are less detailed and contain areas that are considerably more dense and compact than those

at lower hierarchical levels. Our new materialization method avoids materializing certain such dense areas, based on a well-defined cost model that bounds the amount of online aggregation needed to calculate these areas. As we will demonstrate in this chapter, this method not only results in dramatic storage and computation time savings, but also provides significantly faster response times in queries (due to improved buffering and locality).

Knob Cost Model and Materialization. In this section we describe the knob cost model, which associates a knob value to each node of the CRC structure, depending on the worst-case cost of queries that address the sub-cube that this node dominates. This value is used to control the amount of performed materialization by avoiding the computation and storage of some sub-cubes which can be calculated on-the-fly with a cost (in the number of generated subqueries) of at most equal to the *knob threshold*. Any query that targets a non-materialized sub-cube, is broken in multiple queries that need to be further aggregated in run-time.

To calculate the knob value at each node in the CRC structure, we first assign a knob value $V = 1$ to all leaf nodes. For non-leaf nodes, we then need to consider the maximum number of generated subqueries to which any point query through that node will be split into, and whose results will need to be aggregated in *run-time*.

Therefore, the knob value of a non-leaf node N should be set to:

$$V(N) = \max_{\text{cell } c_i \text{ in } N} \begin{cases} V(c_i.\text{sub-cube}) & c_i \text{ normal cell} \\ \sum_{S_{ij} \text{ of } c_i} [V(S_{ij})] & c_i \text{ split cell} \end{cases}$$

In the above equation, $V(c_i.\text{sub-cube})$ denotes the knob value of the only sub-cube dominated by a normal cell c_i, while $V(S_{ij})$ denotes the knob value of the j-th sub-cube S_{ij} dominated by a split cell c_i. This model assigns very small costs to lower areas of the cube that can be aggregated efficiently during run-time while, through the combination of the max operation over the sum of the costs of the split nodes, denser areas that are difficult to aggregate in run-time get monotonically higher values.

5.3 CRC Construction and Updates

5.3.1 CRC Construction

The CRC construction algorithm is motivated by the ideas of [SDRK02], but is considerably more complex due to the significantly larger number of views that exist in the rollup cube, the dependencies between these views because of the defined hierarchies, the introduction of our knob metric to control the amount of materialization, and the hierarchical pruning property. By far the most important advantage of the Dwarf structure in [SDRK02] was a property called *suffix coalesc-*

Algorithm 4 ExpandRollupCube Algorithm

Input: sorted fact table, D : number of dimensions

```
1: Create all nodes and cells for the first tuple
2: active_path = root-to-leaf path of created nodes
3: while more tuples exist unprocessed do
4:    current_tuple = extract next tuple from sorted fact table
      {current_tuple may reveal that some nodes will not get additional cells}
5:    Moving bottom-up in active_path, call KnobMaterialize(N) for nodes that will not get additional input
6:    Create necessary nodes and cells for current_tuple
7:    Assign knob value = 1 to new nodes
8:    active_path = root-to-leaf path of last inserted tuple
9: end while
10: Moving bottom-up in active_path, call KnobMaterialize(N) for nodes that will not get additional input
```

ing, which was based on the observation that, especially in sparse datasets, there are multiple group-bys (views) that are produced by the same set of the fact table's tuples. These group-bys would create identical copies in the data structure. The algorithms in [SDRK02] managed to identify such group-bys and eliminate this type of redundancy *before* their computation, therefore not only reducing the size of the compressed data cube, but also reducing the running time of their algorithms, since the redundant parts of the structure did not have to be calculated.

The construction of the CRC structure is based on three interleaved processes: the ExpandRollupCube, the KnobMaterialize and the SuffixCoalesce algorithms. These algorithms will be presented shortly in detail. Briefly, the ExpandRollupCube parses the tuples in the sorted fact table and creates all the nodes and cells in the CRC structure to store the dimension coordinates and values of the input tuples. It is important to note that the ExpandRollupCube algorithm creates nodes and cells that correspond to the most detailed hierarchy level of each dimension. To calculate and store all the other views in the rollup cube, the KnobMaterialize algorithm is

used. Finally, this algorithm requires merging/aggregating the information stored in different sub-cubes. The SuffixCoalesce algorithm performs this merging process.

The ExpandRollupCube Algorithm. At the beginning the fact table is sorted based on the dimension ordering of decreasing cardinalities. Then the ExpandRollupCube algorithm, presented in Algorithm 4, parses the tuples of the fact table one by one, and creates all the necessary nodes and cells in the CRC structure to store the dimension coordinates of each tuple. These dimension coordinates correspond to the lowest (most-detailed) hierarchy level of each dimension. Due to the sorting of the tuples, it is trivial to figure out at each step whether there will be nodes in the CRC structure that will receive no more input. This is simply done by viewing the coordinates of the current and the immediately previously inserted tuple. The aggregate values of each of these tuples can be reached from a root-to-leaf path of length exactly D. If we consider the node where these two paths will diverge, for all the nodes of the previously inserted tuple lying below the diverging node, it is certain, due to the sorting of the tuples, that they will receive no more input. For these nodes, moving bottom-up, we therefore need to calculate the content of their ALL cell. This is accomplished by a call to the KnobMaterialize algorithm. Any node created by the ExpandRollupCube algorithm is assigned a knob value of 1.

As an example, consider the hierarchy definitions and metadata of Table 5.2, the fact table of Table 5.2, and the resulting CRC structure in Figure 5.1. The nodes in this figure are numbered based on their order of creation. We first process the tuple $\langle S1, C2, N1, 10\rangle$, create the nodes 1,2 and 3 (one for each dimension) and

insert one cell in each of these node. We then process the tuple $\langle S2, C3, N2, 30\rangle$. Nodes 2 and 3 will now certainly not receive any additional input and, therefore, the KnobMaterialize algorithm is called first for node 3, and then for node 2. Afterwords, a cell will be added to node 1 to store the $S2$ coordinate, and nodes 4 and 5 will be created.

The KnobMaterialize Algorithm. The KnobMaterialize algorithm, presented in Algorithm 5, is used to calculate the sub-cube of the ALL cell of each node. If the node contains a single cell, then hierarchical pruning happens here, and the content of the ALL cell is the same as the content of the lone cell in the node (Lines 2-4). Otherwise, in the presence of hierarchies, when we create the sub-cube pointed by the ALL cell of a node which does not belong to the last fragment of its hierarchy level, we first move one level up in the current dimension's hierarchy, and map each key value of the cells to be merged to their parent values in the hierarchy (Lines 6-9). The metadata manager provides us with this parent mapping (Line 7). We then calculate the knob value of each cell in the new node (of the less detailed level) and decide whether to perform or avoid the merging process of the corresponding sub-cubes (Lines 10-19). Finally, if the cells belong to the upper hierarchy level of their dimension, then we cannot move to a higher hierarchy level, and the SuffixCoalesce algorithm is called to merge/aggregate the data of the cells' sub-cubes (Line 25).

As an example consider node 9 in Figure 5.1. This node contains coordinates of the second dimension (Product), at its most detailed hierarchy level. When we compute the content of the ALL cell of the node, we want to move one level higher

in the Product hierarchy, to the Group level. Based on the metadata of Table 5.2, the products C1 and C3 correspond to group G2, while the product C2 corresponds to group G1. Therefore, node 10 is created with 2 cells containing the G1 and G2 values. The sub-cube of G2 is not materialized, since the knob value of the cell is 2 (the sum of the knob values of nodes 5 and 8). The sub-cube of G1 is produced by merging the single sub-cube pointed by the cell C2 of node 9. In this case, as we will show, the SuffixCoalesce algorithm trivially recognizes that the resulting sub-cube will be identical to the input sub-cube, and therefore the G1 cell points to the root of the input sub-cube (node 3). The knob value of node 10 is therefore equal to 2, which is the maximum knob value among all cells of the node. Finally, for the ALL cell of the node, we need to call the SuffixCoalesce algorithm, since the new node belongs to the last fragment of its dimension.

The SuffixCoalesce Algorithm. The SuffixCoalesce algorithm is presented in Algorithm 6. It requires as input a set of cubes (*inputCubes*) and merges them to construct the resulting cube. The algorithm makes use of the helping function calcAggregate, which aggregates the values passed as its parameter.

SuffixCoalesce is a recursive algorithm that tries to detect at each stage whether some sub-cube of the resulting cube can be coalesced with some sub-cube of *inputCubes*. If there is just one cube in *inputCubes*, then coalescing happens immediately, since the result of merging one cube will obviously be the cube itself. The algorithm then repeatedly locates the cells *toMerge* in the top nodes of inputCubes with the smallest key Key_{min} which has not been processed yet. A cell in the result-

ing cube with the same key Key_{min} needs to be created, and its *content* (sub-cube or aggrValues) will be produced by merging the *contents* of all the cells in the *toMerge* set. There are two cases:

1. If we are at a leaf node and at the last fragment, we call the function calcAggregate to produce the aggregate values for the resulting cell.

2. Otherwise, coalescing cannot happen at this level. We call the SuffixCoalesce algorithm recursively to calculate the cube of the current cell, and check if parts of the structure can be coalesced at one level lower.

At the end, the ALL cell for the resulting node is created, either by aggregating the values of the node's cells (if this is a leaf node at the last fragment) or by calling the KnobMaterialize algorithm, with the sub-cubes of the node's cells as input.

Memory Requirements. The memory requirements of the overall algorithm are quite small since the only memory needed is that required to keep open nodes (i.e. nodes where we are still inserting cells). Since in the worst case we will descend all D dimension levels of the structure when creating the ALL cell of the root node, the memory requirements of the algorithms are: $MaxMemoryNeeded = c \cdot \sum_{i=1}^{D} Card_i$, where c is the size of the cell and $Card_i$ is the cardinality of dimension i at its most detailed hierarchy level. We must point that the hierarchy levels inside the dimensions do not change the memory requirements since nodes that correspond to lower hierarchical levels are closed (and the memory required to maintain them released) before proceeding to higher level nodes.

Algorithm 5 KnobMaterialize Algorithm

Input: CubeSet: Set of cells pointing to the sub-cubes to merge (each sub-cube corresponds to one cell)

```
1: MappedCubeSet ← ∅
2: if CubeSet contains just a single cell then
3:    return Content_0 {Hierarchical Pruning happens here}
4: end if
5: if cells in CubeSet are not in last fragment of their hierarchy then
6:    for each cell [Key_i, Content_i] of CubeSet do
7:       fatherMapping = getFatherMapping(Key_i)
8:       MappedCubeSet.insert(cell [fatherMapping,Content_i])
9:    end for
10:   for cells c with the same key_i in MappedCubeSet that point to nodes S_j do
11:      {calculate the knob value if we do not materialize}
         knob_1 ← ∑_{S_j}[knob(S_j)]
12:      if knob_1 > knob-threshold then
13:         {create the sub-cubes in lower levels of the CRC Structure}
            create a normal cell using SuffixCoalesce(c)
14:         temp-knob(new cell)=max_{S_j}[knob(S_j)]
15:      else
16:         {avoid materialization}
            Create a split cell that points to all its underlying sub-cubes
17:         temp-knob(new cell)=knob_1
18:      end if
19:   end for
20:   Create a node N for the current hierarchy fragment with all the new cells
21:   {determine the knob value of the node}
      knob(new node)=max [temp-knob(new cells)]
22:   {recursively proceed to higher hierarchical levels}
      Create the ALL cell of N with KnobMaterialize(MappedCubeSet)
23:   return N
24: else
25:   return SuffixCoalesce(CubeSet) {Finished merging for hierarchy}
26: end if
```

5.3.2 Updating of CRC

In this section we describe how the CRC structure can be incrementally updated, given a set of delta tuples from the data sources. The incremental update process is based on the three interleaved algorithms that are used to initially construct the CRC structure. In a nutshell, the incremental update procedure merges the sorted delta fact table with the old CRC and stores the resulting CRC in the same file as the old one. During this process, our algorithms are able to efficiently identify

Algorithm 6 SuffixCoalesce Algorithm

```
Input: inputCubes = set of cubes
 1: if only one cube in inputCubes then
 2:    return cube in inputCubes {coalescing happens here}
 3: end if
 4: while unprocessed cells exist in the top nodes of inputCubes do
 5:    find unprocessed key Key_min with minimum value in the top nodes of inputCubes
 6:    toMerge = set of Cells of top nodes of inputCubes having keys with values equal to Key_min
 7:    if in the last level and fragment of structure then
 8:       write cell [Key_min  calcAggregate(toMerge.aggrValues)]
 9:    else
10:       write cell [Key_min  SuffixCoalesce(toMerge.sub-cubes)]
11:    end if
12: end while
13: create the ALL cell for this node either by aggregation or by calling KnobMaterialize
14: return position in disk where resulting cube starts
```

those sub-cubes that are not affected by the update tuples and remove them from consideration, thus significantly reducing the update time. For the remaining sub-cubes, new nodes may need to be created to accommodate the insertion of new cells caused by the update tuples.

All three algorithms update an already existing node, and only if there is not enough space in the existing node to hold all the cells, they allocate a new node and release the old one, thus making it available for allocation in subsequent steps. Due to the complexity of the algorithms (all three presented algorithms need to be modified) and the space limitations, we only sketch the necessary modifications.

UpdateRollupCube. This algorithm is similar to the ExpandRollupCube algorithm. It continuously parses the tuples in the sorted delta fact table and traverses the CRC structure to update existing aggregate values and create nodes and cells in areas that are not materialized, in the same way ExpandRollupCube does. If a new cell needs to be inserted in an existing node, then a new node is allocated with

enough space to hold all the cells and the old node is deleted, making the storage it occupied available for allocation. Our algorithms try to populate these deleted areas in subsequent steps of the update algorithm, in order to reduce the size of the updated CRC structure. When there are no more delta tuples that correspond to a node, then the UpdateKnobMaterialize algorithm is called in order to create the hierarchical levels of that node, with input the old hierarchy level and the cells to be merged divided into two regions, the cells that were not updated by the UpdateRollupCube and the new or updated cells. We must point out that cells from A may participate in the merging phase only when these cells have the same father values with cells in region B.

UpdateKnobMaterialize. This algorithm is used to update the sub-cube of the ALL cell of each node. The difference with KnobMaterialize is that the input set of the cells to be merged is divided into two regions. The first region A contains cells that were not updated and the second region B contains updated or new cells. This is used to guide the traversal only to areas of the cube that need updating. The algorithm works like KnobMaterialize by mapping keys to their father values and either merges the corresponding sub-cubes using the UpdateSuffixCoalesce algorithm for normal cells or avoids materialization and creates split cells if the corresponding knob value is less than the knob threshold. The merging only happens for father keys with cells in region B, otherwise there is no need to update the corresponding sub-cubes. As in UpdateRollupCube, cells from A may participate in the merging phase only when these cells have the same father values with cells in region B. More-

over, the process may eliminate from some existing nodes the hierarchical pruning pointers, due to the insertion of new cells in the nodes.

UpdateSuffixCoalesce. This algorithm is used to perform the merging of sub-cubes and its main difference with the SuffixCoalesce algorithm is that its input set of cells to merge is divided in two regions. Like in the UpdateRollupHierarchy algorithm the region A contains cells that were not updated and region B contains new or updated cells. The algorithm merges only areas of the sub-cube with at least one cell in region B (no merging is performed in the case where all the cells are in region A). When the UpdateSuffixCoalesce algorithm is called to update a coalesced sub-cube, it first checks to see if the area is still coalesced (i.e. there is only one cell to merge) and if this is the case, it returns immediately. Otherwise, the UpdateSuffixCoalesce algorithm creates a new sub-cube. The last case corresponds to the situation where a coalesced area is no longer coalesced because of the deltas.

5.3.3 Handling Complex Hierarchies

Some hierarchies are more complex than the ones that we have presented so far. Consider for example the lattice of Figure 5.2, where we demonstrate two different aggregation paths for the time hierarchy.

Figure 5.3 depicts, conceptually, how we can create nodes of CRC in different fragments by following the paths in the graph hierarchy. At each step we can create, from any node N, nodes that correspond to all possible immediate parent hierarchy levels based on the graph hierarchy. However, in our work we prefer to serialize

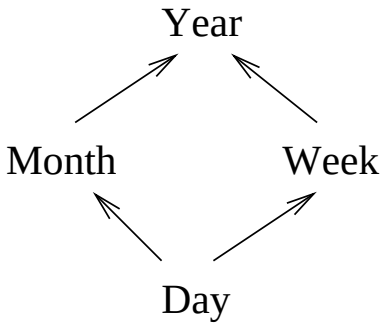


Figure 5.2: Non-Flat Hierarchy

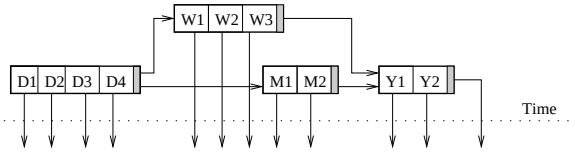


Figure 5.3: Conceptual representation of non-flat hierarchy

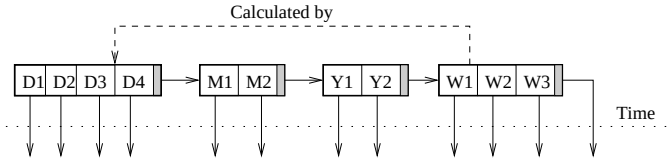


Figure 5.4: Real implementation of non-flat hierarchy

the hierarchy levels. Figure 5.4 demonstrates an example of such a serialization. By traversing the graph hierarchy in a Depth-First manner we construct the corresponding nodes and serialize them. The metadata manager can then be used to locate any level at query time.

5.4 Hierarchical View-Cover

In this section we present the *Hierarchical View Cover* (HVC) approach which stores all the views of the rollup cube in a non-redundant way. The alternative way of storing the full rollup cube by treating each hierarchical level as a different dimension is not scalable and requires significantly more space and time. The number of views, when treating each single level as a different dimension is: $2^{\sum L_i}$, while the number of views that hierarchical view cover stores is: $\prod (L_i + 1)$, where L_i is the number

of levels for dimension i. For example, even for the three-dimensional data set in Table 5.2, the number of views that hierarchical view cover handles and stores is 18 ($= 3 \cdot 3 \cdot 2$), while the number of views if we treat each hierarchical level as a different dimension is 32($= 2^{2+2+1}$).

The HVC approach can be applied to any method that handles flat (with no hierarchies) cubes in order to handle hierarchical dimensions and is used in this chapter as a reference point. In the following, without loss of generality, we describe HVC using the Dwarf structure([SDRK02]) to handle flat cubes.

Partial Dwarfs. To store all the views of the rollup cube, we create a forest of *Partial Dwarfs*, each of which will store a subset of the cube's views. There is a single Dwarf cube for each combination of hierarchy levels from different dimensions (except for the top-most *ALL* level). All but the first of these Dwarfs will be *partial* in the sense that they do not store every possible combination of views. This is done to avoid duplicating the storage of some views and will be made clear with an example.

Table 5.2 contains the declaration of the hierarchies imposed on the *Store*, *Product* and *Customer* dimensions of a sample dataset. There are 18 possible views (Table 5.3) defined in the rollup data cube in the presence of these hierarchies. For instance the view *Retailer.Code.Name* aggregates the measure(s) on three dimensions: *Store* (at the *Retailer* level), *Product* (at the finer *Code* level) and *Customer* (at the *Name* : level); i.e. corresponding to the group-by: `Store.Retailer, Product.Code, Customer.Name`.

Partial Dwarf	Calculated Views	Not Calculated Views
1	StoreId.Code.Name, StoreId.Code, StoreId.Name, StoreId, Code.Name, Code, Name, None	
2	StoreId.Group.Name, StoreId.Group, Group.Name, Group	StoreId.Name, StoreId, Name, None
3	Retailer.Code.Name, Retailer.Code, Retailer.Name, Retailer	Code.Name, Code, Name, None
4	Retailer.Group.Name, Retailer.Group	Retailer.Name, Group.Name,Retailer, Group,Name, None

Table 5.3: HVC for Dataset of Table 5.2

To store the 18 views of the rollup cube, we create 4 Dwarfs, as shown in Table 5.3, and store in each Dwarf a subset of the possible views (the original Dwarf [SDRK02] of a set of D attributes stores all 2^D views on every combination of the attributes). We here note that we cannot directly use the Dwarf structure as presented in [SDRK02] to store all 18 views, and that the definition of the 4 Dwarfs is not unique. Any non-redundant set of Dwarfs that *covers* all the views of the rollup data cube is acceptable as a solution and constitutes a *Hierarchical View Cover* (HVC). However, if the i-th dimension contains L_i hierarchy levels, then we can show that at least $\prod_{i=1}^{D} L_i$ Dwarfs need to be created to cover all the views. The proof is based on the fact that each Dwarf of D dimensions can only store views that contain a subset of these D dimensions. In this example, at least $2 \times 2 \times 1 = 4$ Dwarfs need to be created. However, these are constructed partially, to avoid duplicating the storage of some views. Table 5.3 presents the views that are stored at each *Partial* Dwarf, and the ones that are not calculated or stored, to

avoid their duplication. These Partial Dwarfs can be computed by modifying the algorithms of [SDRK02] to "block" some recursive calls (calls to the SuffixCoalesce algorithm) that create these views.

To understand why the HVC presented in Table 5.3 is a good candidate covering, one has to recall that the computation of each Dwarf requires an initial sort of the fact table (a single sort using the combined key composed by all dimension values). We can create the fact tables for Dwarfs 2,3 and 4 (in the specified order) from the fact tables of Dwarfs 1,1 and 3 (respectively) by using the declared metadata (ex: Table 5.2) to map the values of each hierarchy to the next level. Notice that the fact tables for Dwarfs 2 and 4 will then be partially sorted, because the initial dimension is the same as in the Dwarfs 1 and 3. Thus, the sorting operations for these Dwarfs are less costly.[3]

To construct the HVC of Table 2 we use a simple enumeration process. We first set the fact table of the first Dwarf to contain the most detailed levels of each dimension. We then enumerate all the possible combinations of hierarchy levels, by changing more quickly the hierarchy level of the last dimension, and slower the hierarchy levels of the first dimension, and assign each such combination one-by-one to the Partial Dwarfs. The advantage of this enumeration is that the duplication of views can be avoided by just looking at the definition of the latest Partial Dwarf.

CRC vs HVC. Any covering of the rollup cube's views using a set of Partial Dwarfs has certain disadvantages. First of all, there are cases when some prefix redundancies

[3]We assume that the mapping of the dimension values to higher levels of its hierarchy can be arbitrary and is not necessarily order preserving.

are not exploited. For example, views containing the hierarchy level `StoreId` exist in both the first and the second Dwarf. Moreover, by using multiple Dwarfs we cannot remove all the suffix redundancies of the rollup cube. Suppose for example that some retailer supplies a single store. Then the views containing this store in the Partial Dwarfs 1 and 2 (in the `StoreId` and `Retailer` levels correspondingly) will be identical, and their storage will be duplicated. On the other hand, the CRC structure does not suffer from these drawbacks, since it stores all the views in a single structure and manages, therefore, to eliminate all possible prefix and suffix redundancies.

5.5 Experiments

In this section we provide an evaluation of Dwarf, HVC and our proposed CRC structure for managing rollup data cubes. We used a real dataset provided by an OLAP company, whose name is not disclosed due to our agreement. The eight-dimensional dataset has hierarchies on four dimensions and its characteristics are summarized in Table 5.4. All the experiments were performed on a Pentium 4 PC clocked at 1.8GHz and with 1GB of memory. The memory available to the buffer manager of our implementation was limited to 256MB.

In our experimental evaluation we compare all techniques on all terms: query performance, computation time and required storage. We further include in the presentation the corresponding numbers for simply storing the base cube comprised just of the views corresponding to the most detailed hierarchy levels of each dimension

Dimension	Level Cardinalities
A	$7458 \rightarrow 2265 \rightarrow 737 \rightarrow 188 \rightarrow 32 \rightarrow 11$
B	$2765 \rightarrow 91 \rightarrow 31 \rightarrow 8$
C	$3857 \rightarrow 841 \rightarrow 111 \rightarrow 16$
D	$213 \rightarrow 68 \rightarrow 8$
E	3247
F	660
G	4
H	4

Table 5.4: Real Dataset Hierarchies

(without hierarchies) using the Dwarf algorithm presented in [SDRK02]. We denote this implementation as Base Dwarf. This provides better insight on the benefits of materializing the rollup cube (using our CRC structure or HVC). When the base cube is used, rollup or drilldown hierarchical queries induce a post-processing cost because aggregation to the proper level of each hierarchy has to be performed in an additional step. Finally, we perform an experimental evaluation of the incremental update algorithm of CRC.

Workload Description. In OLAP applications, the user typically performs a series of exploratory queries to identify areas of interest, and then drills down (or rolls up) to more (or less) detailed data. A very common operation in this case, is to use a children(x) function to specify interest to all the children for a given value x in a hierarchy level, and then drill down to those children. For example, in a sample $Time$ hierarchy, the value of $children(2003)$ could be the twelve months of year 2003.

We used two workloads that try to emulate this behavior by generating queries that reference parts of the cube. The difference in the two workloads lies in the

		Probabilities			
Workload	#Queries	All	DD/RU	Children	Width
A	1,000	60%	50%	30%	5%
B	1,000	60%	0%	30%	5%

Table 5.5: Workload parameters

amount of rollup/drilldown queries. Workload A contains rollup/drilldown queries with a probability 1/2, i.e every other query is either a rollup or a drilldown query. Workload B contains only random ad-hoc queries without any rollup or drilldown queries. We believe that a real query workload lies somewhere in the middle. Both workloads contained 1,000 queries and their parameters are presented in Table 5.5.

Each query can be described as a path $\langle D_A, D_B, ..., D_H \rangle$, where D_i is a subpath that corresponds to dimension i and has the pattern $\langle l_{i,1}, l_{i,2}, ...l_{i,k} \rangle$, where $l_{i,j}$ is the level j of dimension i. The query specifies at each $l_{i,j}$ either the pseudo-value ALL, or a set of "points". A point can either be a literal value of that level or all the children values of a father value in the immediately higher level. The column "ALL" represents the probability of D_i not participating (being specified) in the query. In the case where D_i participates, a level $l_{i,j}$ is uniformly chosen from all levels of D_i, and a set of values is generated for that level. The "Width" column corresponds to the number of values that are generated and in our case it is uniformly distributed over 5% of all possible values for the level. The "Children" column corresponds to the probability of asking for the children of a father value and in our case is 30%. The "DD/RU" column depicts the probability for a Drill-Down or a Roll-Up query. Such a query is created by rewriting appropriately the immediately previous query.

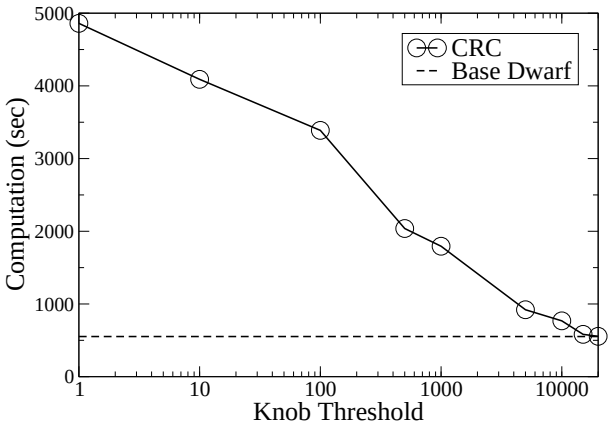


Figure 5.5: Computation vs Knob

Knob Evaluation. We first evaluate the effect of the knob materialization on the computation time, the storage requirements and the query performance of the CRC. For this experiment we used the full real dataset of 13,5 million tuples and set the parameter $G_{min} = 1,000$, as it was recommended in [SDRK02]. We also used the query workloads A and B with the parameters specified in Table 5.5. In Figures 5.5, 5.6 and 5.7 we show the results for varying values of the knob parameter. We observe that with a knob value of 10,000 CRC becomes about six times faster in computation time and requires eight times less space compared to CRC with knob=0, where all hierarchical aggregates are calculated and stored, while its query performance is almost doubled.

We observe that as the knob increases (note the logarithmic x-axis) the query performance gradually increases (the response time decreases) reaching a maximum point at a knob threshold $\approx 10,000$ and then the query performance starts to

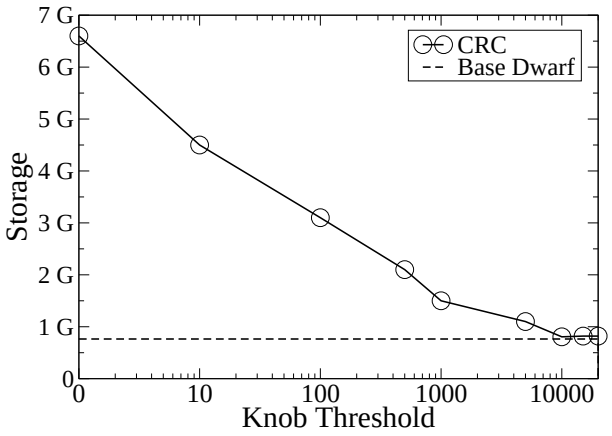


Figure 5.6: Storage vs Knob

decrease. As the value of the knob threshold increases, queries perform more work, as aggregation is required during query time. On the other hand, larger values require significantly less space to store the CRCand, therefore, improve buffering and locality. The tradeoff between processing and accessing secondary memory determines the optimum value for the knob threshold. Our experience with CRC reveals that a knob value between 1,000 and 10,000 often results in the optimal performance.

Compared to the base Dwarf with only the raw aggregates, we see that while CRC with knob=10,000 requires a negligible overhead in creation time (0.5%) and just 5% overhead in storage, it improves query performance by a factor of seven. The base Dwarf requires 1706 sec and 1331 sec for workloads B and A respectively.

The main reason for the superiority of the CRC structure is the use of the knob parameter which is a data-based metric that materializes only those subcubes that are more expensive to calculate during the queries. On the contrary, no such

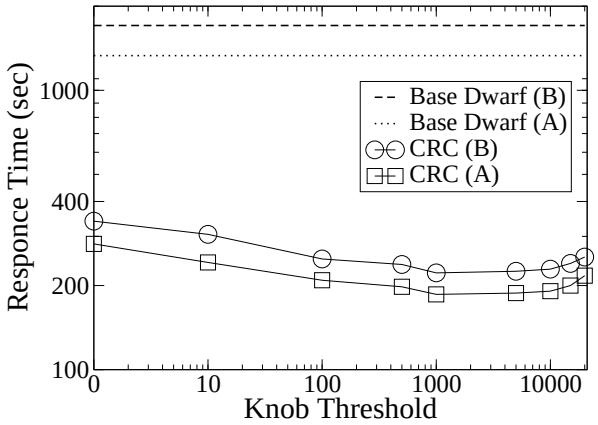


Figure 5.7: Workload vs Knob

optimization can take place in the base Dwarf, where only the lower levels of the dimensions exist and all other levels are handled externally.

Tuples	Base Dwarf	HVC	CRC
134,280	1s	286s	1s
1,344,591	10s	3160s	10s
2,690,181	34s	10221s	34s

Table 5.6: Sorting Evaluation

Fact Table (Tuples)	Base Dwarf	HVC	CRC
134,280	1:49	1:15	1:1244
1,344,591	1:29	1:11	1:523
2,690,181	1:25	1:12	1:531

Table 5.7: Compression Ratio over corresponding Data Cube

Computation/Storage Space Evaluation w.r.t #Tuples. In this experiment we use a varying uniform sample of the original dataset to demonstrate the scalability of the techniques with respect to the number of tuples. For the granularity parameter we used the proposed value $G_{min} = 1,000$ of [SDRK02], which avoids materialization

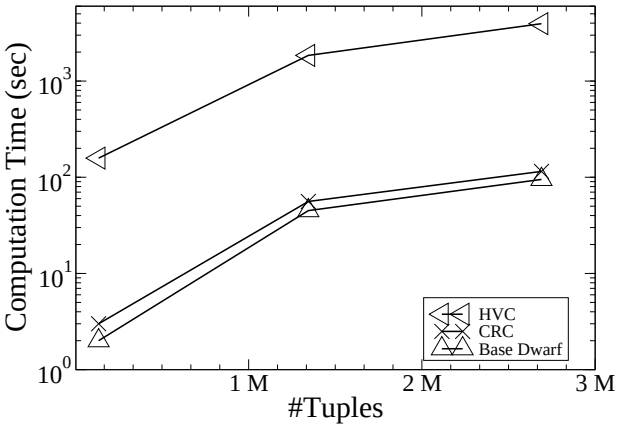


Figure 5.8: Computation vs. #Tuples

of areas of the cube with under $1,000$ tuples. The aggregation for these areas is performed during query time. For these experiments we also set the knob threshold to $10,000$ for CRC, because as it is demonstrated in section 5.5 this threshold optimizes both the query performance and the computation/storage requirements.

The computation times and the required storage for each structure are presented in Figures 5.8 and 5.9, correspondingly. Additionally, we present the sorting times for all methods in Table 5.6. The column names of this table refer to the corresponding structure being used. The Base Dwarf contains only the most detailed 256 views for this 8-dimensional dataset. In contrast, the rollup data cube has 11,200 views, partitioned among 288 Partial Dwarfs, as described in Section 5.4. Note that the first Partial Dwarf is identical to the Base Dwarf. CRC contains all 11,200 views in a single store.

We observe that in all cases CRC needs considerably less space and time to be computed than the HVC. It is important to note here that when the hierarchies are

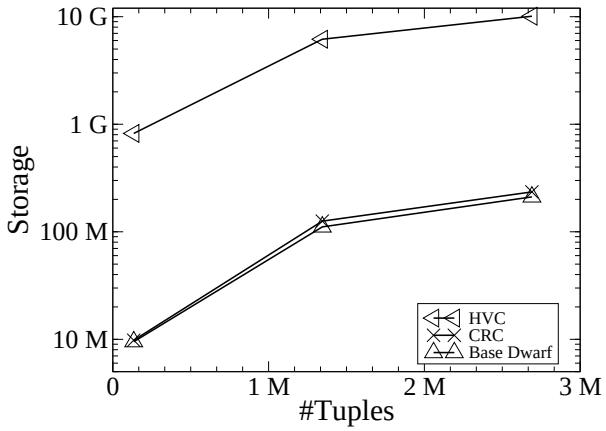


Figure 5.9: Storage vs. #Tuples

not order-preserving (i.e. sorted values in lower levels do not map to sorted values in higher levels) a sorting operation is required for each Partial Dwarf. The sorting time in that case dominates the computation time of the Partial Dwarfs, as shown in Table 5.6.[4]

Although, the Base Dwarf contains only a small fraction of the views stored in CRC (256 vs 11,200 views), we observe that the computation and storage requirements of the two structures are almost identical. However, in the query evaluation experiments we demonstrate that the query performance for the Base Dwarf suffers significantly due to the amount of online processing required. Since data warehouses are typically bulk-loaded at specific time intervals, the superior query performance of CRC would be more desirable in most applications where a significant amount of queries and post-processing is performed.

Table 5.8 shows the number of tuples and the binary storage footprint (BSF)

[4]For this experiment, we exploited partially overlapping sort orders, as explained in Section 5.4.

Fact Table	Full Rollup Cube		Full Base Cube	
Tuples	Tuples (billions)	BSF (GB)	Tuples (billions)	BSF (GB)
134,280	16.9	12.2	0.565	0.45
1,344,591	109.8	65.8	2.9	3.1
2,690,181	189.9	124.7	5.4	5.2

Table 5.8: Full Cube Statistics

of the base cube (without hierarchies) and of the rollup cube respectively. The BSF representation stores the cube in unindexed binary relations. The presence of hierarchies substantially increases the cube size. In Table 5.7 we further compute the compression ratios obtained by the three implementations[5] when G_{min}=1,000. We observe that the storage savings are even higher for CRC, since it eliminates redundancy over all the hierarchical views.

Query Performance Evaluation vs #Tuples. Figures 5.10 and 5.11 contains the results for workloads A and B over the Base Dwarf, HVC and CRC when G_{min}=1,000 and knob threshold = $10,000$ are being used. In the Base Dwarf a lot of external aggregation is required for most queries. In CRC the knob materialization is data-driven and avoids computing and storing hierarchical aggregates that are easy to compute during query-time. The result is that the Base Dwarf exhibits substantially slower query response times compared to both CRC and HVC. The latter is faster than the Base Dwarf –since hierarchical aggregates are computed and stored– but requires significantly more space and time to compute. For larger samples of the fact table, the HVC could not be stored within our disk space.

The new CRC structure significantly outperforms the Base Dwarf, while re-

[5]For the Base Dwarf the computation is over the size of the base cube.

quiring much less computation and storage than the HVC. Compared to the Base Dwarf it offers approximately x7 times better query performance. Compared to HVC, which stores all hierarchical aggregates, CRC exhibits better query performance due to better buffering and locality.

The effect of having Drill-Down and Roll-Up queries is the same overall for all storage techniques. Due to common paths being buffered between such queries, all structures benefit from their existence [SDRK02]. On the contrary, due to their lack of locality, random queries (workload B) that often access distant disk pages are proportionally worse.

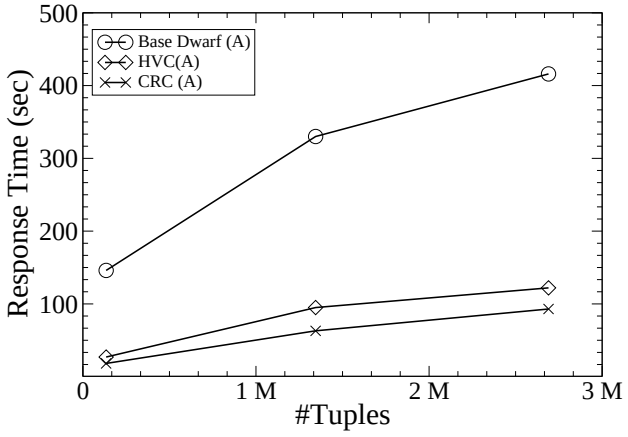


Figure 5.10: Workload A (1,000 queries) vs #Tuples

Update Performance. We evaluated the update performance of the CRC with $G_{min} = 1,000$ and knob= $10,000$ by using a sample of 1,344,591 tuples and incrementally updating the structure by adding ten batches of 134,459 tuples. In Figure 5.12 we depict the initial construction time along with the time required for each batch of updates. We observe that the time to incrementally update the struc-

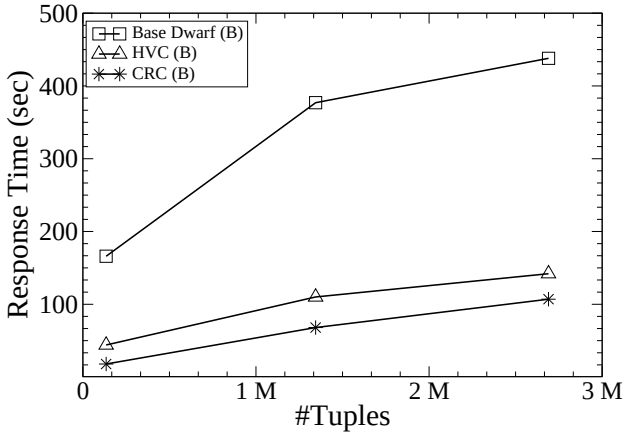


Figure 5.11: Workload B (1,000 queries) vs #Tuples

ture remains virtually unaffected by previous updates and that it is considerably less than fully recomputing the cube each time. For comparison the resulting CRC after being incrementally updated ten times required 145MB vs. 125MB if we fully reconstructed it. The difference in the size occurs because of the presence of nodes that had to be expanded during the update operations. A copying process could periodically run in the background and move the updated CRC structure into a new file, thus consolidating any unused areas inside the structure.

5.6 Related Work

The concept of aggregation over dimension hierarchies is fundamental in data warehouse modeling [BPT97b, CD97, JLS99] and its importance is exemplified in the Star and Snowflake schemata [Kim96]. A plethora of papers has focused on the conceptual modeling of data warehouses, see [VS99] for a survey. To a great extend, materialized aggregate views and their potential have been rediscovered in the context of OLAP and data warehousing due to the introduction of the data cube

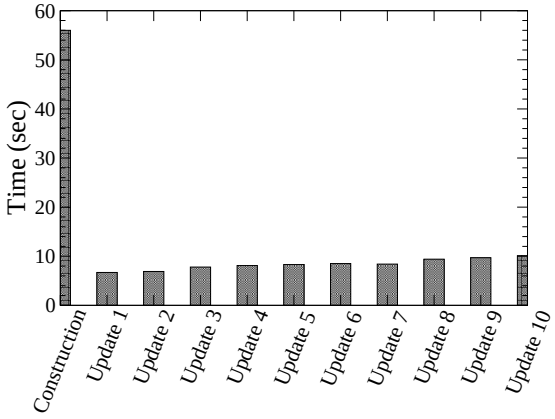


Figure 5.12: Incremental Update Performance

operator in [GBLP96]. Rather quickly, researchers saw the potential of the new operator as well as its drawbacks: its enormous storage and processing costs.

[Rou82] first explored the problem of selecting a set of materialized views (with no aggregations) for answering queries under the presence of updates and a global space constraint. View selection algorithms in the context of the data cube can be found in [GHRU97b, HRU96b, TS97]. The authors of [KM99] show that no polynomial time (in the number of views) approximation (with respect to query response time) algorithm exists for the view selection problem unless $P = NP$. We notice here that most of the aforementioned greedy algorithms have complexity that is polynomial in the number of views, which is in-fact exponential in the number of dimensions, making them impractical for multidimensional datasets with hierarchies. In [SDRK02] a different reduction technique, similar in spirit to *iceberg queries* [FSGM$^+$98], was presented. The key idea is to avoid materializing portions of the views whose aggregates can be computed by at most G_{min} other tuples, where

G_{min} is a tunable parameter of the data structure.

The time to compute the data cube is another overwhelming factor. Techniques that have been proposed take advantage of commonalities between different views by sharing partitions, sorts or partial sorts and intermediate results [AAD+96, DANR96, SAG96]. In [ZDN97] an array-based algorithm is proposed that uses memory-arrays to store partitions and to avoid sorting. The algorithms in [BR99, RS97] are designed to handle sparse data cubes, without however eliminating any cube redundancies. The Bottom-Up Cube (BUC) algorithm described in [BR99] stores only those partitions of a view whose values are produced by aggregating at least $MinSup$ tuples of the fact table. The parameter $MinSup$ is called the *minimum support* and is analogous to the G_{min} parameter of [SDRK02]. A key difference is that the interleaved top-down/bottom-up generation of the data cube in [SDRK02], in contrast to the bottom up computation in BUC, allows the elimination of redundancies on the aggregate views before actually computing the aggregates.

Several indexing techniques, in addition to multidimensional arrays mentioned above, have been devised for storing data cubes. Cubetrees [RKR97] and DC-trees [EKK00] extend the original R-tree structure, with the latter supporting hierarchical aggregates. However, they both store *uncompressed* data cubes and are thus limited to small datasets. Cube Forests [JS97] implement a forest of indexes that store the leading dimension coordinates of an aggregate only once. A similar optimization is obtained in the Statistics Tree [FH00]. A more fundamental view for exploiting the redundancy within the data cube is shared among more recent publi-

cations [WLFY02, LPZ03, SDRK02]. The main idea is to reduce the size of the data cube by identifying aggregates that are produced by the same set of tuples in the relation. Up until now, none of these techniques has been extended to support rollup cubes. The presence of hierarchies has a profound effect in the computation and storage complexity of the cube. As we demonstrate in this chapter, a straightforward extension of prior methods is inadequate for managing rollup cubes of realistic sizes. Our CRC construction algorithm employs an interleaved top-down/bottom-up computation strategy for the rollup data cube, which automatically discovers and eliminates all redundancies on a given dataset *in a single pass*. What is important is that this elimination happens prior to the computation of the redundant values. As a result, not only is the size of the CRC dramatically reduced but its computation is also drastically accelerated. In contrast, it is not obvious how the framework we present here can be incorporated in the QC-tree algorithm that employs a bottom-up computation [BR99] of the data cube and eliminates duplicate entries *after they have been discovered*. Clearly, this two-step approach can not be implemented for rollup data cubes of realistic sizes. Furthermore our knob optimization is orthogonal to the G_{min} parameter of [SDRK02]. While the latter works well for sparse areas of the cube, the materialization we present in this chapter focuses on dense areas resulting from the aggregation over dimension hierarchies. The immense size of the rollup data cube deems impracticable storage engines like Cubetrees [RKR97] and the DC-tree [EKK00] because they do not coalesce the aggregates.

5.7 Summary

Aggregate queries on dimension hierarchies are fundamental in exploratory data analysis. However, due to the enormous size of the resulting rollup cube there has been little work on directly supporting such queries. In this chapter we have demonstrated that previous techniques that materialize raw data cubes are very inefficient during drilldown and rollup hierarchical queries because aggregation of the proper level of each hierarchy has to be performed at run-time.

In this chapter we have presented a framework for managing compressed rollup cubes that aggregate data over all dimension hierarchies. Our construction algorithm detects and eliminates redundancy on hierarchical aggregates in a single pass requiring a single sort over the data. However, unlike previous techniques, support for hierarchies is embedded in the data structure, providing significant benefits when querying rollup data cubes. We have also presented a new cost-based optimization for controlling the amount of materialization of the cube. This knob materialization is interleaved with the creation algorithm and has been shown in our experiments to substantially reduce the size of the rollup cube. Using real high dimensional data, we have shown that CRC can store in a single data structure the rollup cube, using almost the exact time and 5% more space than what previous techniques require for the same cube without hierarchies. At the same time, query performance significantly improves (about seven times) because hierarchical aggregates are supported directly.

Chapter 6

Implicated Statistics

6.1 Introduction

Keeping track of "interesting" data trends by evaluating various relations between values of different attributes is the focus of a lot of research. For example, decision support systems are trying to evaluate efficient ways of accomplishing that in an offline fashion. The problem is computationally challenging even for such offline algorithms but is exacerbated for the case of data streams with high throughputs that are encountered in constrained, streaming environments.

However, environments like communication and sensor networks, security and monitoring applications need accurate and up-to-date statistics in real-time in order to trigger certain actions. The class of *Distinct Count*[Gib01] statistics are very useful for such applications since it provides at any moment the distinct number of values or species in a population[BF93]. For example a typical statistic, for a network router is to maintain the distinct number of sources and destinations or even (source,destination) pairs that the router handles. The distinct count problem has been extensively studied in the database literature(see [Gib01] for a survey) and has found applications in other areas like selecting a good query plan in query optimization[PHIS96].

In this chapter we focus on the problem of maintaining distinct *implication* statistics in constrained environments even under the presence of noise. Some items *imply* other items in the sense that they either always, or a given percentage of the time, appear together. We use the term *implication* to denote such properties between sets of attributes and the term *implication count* to refer to the number of items that exhibit such implication properties. The statistics we collect not only generalize the distinct value statistics we mentioned but also aggregate and complement information gathered from data-mining techniques such as association rules or frequent itemsets[MM02]. Such techniques return the set of frequently encountered associated itemsets, while the implication statistics we consider here, return aggregated information (counts, averages, ratios) without returning the involved itemsets. In constrained environments (like sensor networks, where aggregation is important for bandwidth conservation and energy consumption) frequent itemsets and association rules techniques cannot be extended in order to provide real-time aggregates and error guarantees for the implicated queries we are considering. The same stands for the class of "heavy hitters"[CKMS03] which identifies the set of objects whose frequency of appearance is above a given threshold. The cumulative effect of many objects, whose frequency of appearance is less than the given threshold, may overwhelm the implication statistics, although these objects are not identified.

To help clarify the meaning and the extend of the statistics that we address, consider a simplistic data stream called `Network Traffic`, a window of which is presented in Table 6.1. The stream is comprised of the attributes Source, Destina-

Source	Destination	Service	Time
S1	D2	WWW	Morning
S2	D1	FTP	Morning
S1	D3	WWW	Morning
S2	D1	P2P	Noon
S1	D3	P2P	Afternoon
S1	D3	WWW	Afternoon
S1	D3	P2P	Afternoon
S3	D3	P2P	Night

Table 6.1: Example network traffic data

tion, Service and Time and is obtained by the traffic a router observes. Table 6.2 contain examples of real-time statistics which are essential for monitoring purposes and that our framework supports.

A security administrator would like to maintain in real-time the statistic "*how many destinations are contacted by just a single source?*" in order to identify possible intrusion attempts. We consider the Destinations with the implication property: Destination $\rightarrow$ Source. In our example, we have that $\langle D2, S1\rangle$ and $\langle D1, S2\rangle$ have the implication property, $D2$ appears only with $S1$ and $D1$ only with $S2$, and therefore the returned implication count is two. Furthermore, one might want to consider destinations that 80% of the time are contacted by one single source. In that case D3 qualifies and the returned count is three.

Another similar query for this specific dataset is: "*how many services are being requested from only one source?*". The returned aggregate in our case is again two (because the corresponding implications are $\langle WWW, S1\rangle$, $\langle FTP, S2\rangle$).

A small set of all possible implication statistics that someone can keep track for our toy data set in Table 6.1 is classified in Table 6.2 based on the definition of the implication on each instance. The apparent wide range of implication statistics is the

Example	Class
How many sources have we seen so far?	Distinct Count
How many destinations are contacted by only one source?	Implication one-to-one
How many sources contact more than ten destinations?	Implication one-to-many
How many destinations are contacted by only one source 80% of the time?	Implication one-to-one with noise
How many sources do not use *only* the WEB service?	Complement Implication
How many sources contact only one destination during the morning?	Conditional Implication
How many sources contact only one target per service?	Compound Implication
Average number of destinations that 90% of the time are contacted from more than ten sources for the P2P service over a sliding window of 1h	Complex Implication

Table 6.2: Classification of Example Implication Queries

motivation behind this chapter. In our example, such statistics help to keep tracking, in real-time, various traffic parameters. More sophisticated statistics address either conditional or involving one-to-many implications. A security-expert may want to keep track of the answer to the following questions: "*How many sources connect to only one destination during the morning?*", and " *How many destinations are connected from more than ten sources for the P2P service 90% of the time?* ".

Similar aggregate queries are very important to users of decision support systems and data warehouses, where we assume that all the data can be stored and aggregated and that a lot of computation can be performed offline with bulk updates during down time. Even in this case however, the problem of maintaining distinct implication statistics is complicated and requires too many computational

and storage resources. Although we concentrate on data streams, our methods can be applied to offline query scenarios, since our algorithm does not require repeated rescans over the entire database. It can run with input the incremental updates to maintain the implication counts as it does for a data stream. The complication behind maintaining implication statistics is partially common with the problem of *Distinct Value* queries where there is a huge number of duplicates that cannot be accommodated by either available memory or processing power.

In this chapter we describe a framework that can be used to estimate a rich variety of distinct implication statistics in the context of streaming environments under constraints on storage and processing power, as for example in the case of communication routers or sensor networks. In addition our techniques can be applied directly to decision support systems and data warehouses, enriching the collection of aggregates that an analyst can use. For a data stream that is logically divided into two sets of attributes `A` and `B`, we calculate *implication statistics* of itemsets a_i of `A` that imply some itemsets of `B`. The problem is more difficult than identifying frequent itemsets in a data stream, because the contribution of a large number of *infrequent* implicated itemsets can be very significant, overwhelming the aggregate count. We actually show that current streaming algorithms for frequent itemsets[MM02] cannot be extended and provide error guarantees in the aggregate statistics.

The context of the environments we are considering, forces the following assumptions:

- There is not enough memory to accommodate the cardinalities of the attributes

participating in the query. For example, consider the case where one attribute is the network address of a client which in IPv6 has an address space of $O(2^{128})$.

- Although some itemsets may not appear frequently enough and therefore may pass "undetected" by some technique, they can seriously affect the total count. This is the case for first hop routers[WZS02] in distributed denial of service attacks where the counts are very small at the first hop but significantly contributing to the cumulative effect on the last hop routers.

- The exact meaning of the implication depends on the nature of the application. In most situations a analyst will need to allow some tolerance to avoid noise in the data.

The work in this chapter concentrates on how such implication counts can be accurately estimated, in the context described above, by using a small amount of memory that holds a "summary" data structure which makes possible the estimation of the answer. The key issue is that the data structure can be kept up to date with a small amount of effort. Our technique is based on using certain properties of hash functions. We also investigate the error bounds of the estimation and how one can improve those bounds.

The main contributions of our work are summarized as follows:

1. We describe a generalization of implication aggregate queries that frequently arise in the data stream model of data processing and also in other fields of database research.

2. We provide memory and processing efficient algorithms for estimating such aggregates, within small error bounds (typically less than 10% relative error).

3. We prove that the complement problem of estimating non-implication counts can be (ϵ, δ)-approximated under most conditions.

4. We extend online algorithms[MM02] that estimate frequent itemsets and prove that they cannot applied directly to the problem of estimating implication counts.

5. We demonstrate the accuracy of our methods, through an extensive set of experiments on both synthetic and real datasets.

6.2 Applications

The following applications can benefit from such implication statistics. We briefly describe them and how the statistics can be applied to solve important problems in them.

Network traffic monitoring and characterization: Accurately measuring aggregate network traffic from one router to another is essential not only for monitoring purposes but also for traffic control (rerouting), accounting (pricing based on usage), security[SLC+01] (detecting denial of service attacks). Certain characteristics of the traffic like router bottlenecks or patterns of resource consumption[ESV03] or even flash crowds[JKR02] and denial of service attacks can be modeled as implication queries. One can associate triggers when such implication counts exceed certain

thresholds and could for example reroute traffic.

Approximate Dependencies: Functional dependencies have a very "strict" meaning. They leave no room for exception. On the other hand, association rules are essentially probabilistic and allow room for exceptions which is typical in large databases. Approximate dependencies[KM95] attempt to bridge the gap by defining functional dependencies that "almost hold". Such approximate dependencies can be "validated" during updates or on a data-stream by conditions on the aggregate implication counts.

Multi-dimensional histograms or models: A fundamental problem in scenarios like query optimization or query approximation is creating an accurate and compact representation for a multi-dimensional dataset. Typical models used are histograms or probabilistic graph models[FGKP99, GTK01], or other original approaches[BF95]. In [DGR01] a methodology is proposed where the *independence assumption* between attributes is waived. The histogram synopsis is broken into one model that captures "significant" correlation and independence patterns in data and a collection of low-dimensional histograms. Estimations of implication counts can be used in a pre-processing step to provide information about significant dependent or independent areas among certain attributes. These counts can then be used to more efficiently and accurately construct the model part of the synopsis.

6.3 Problem Definition

In this section, we formally describe the problem and the notation. We assume that a node in a distributed environment receives a stream of data and wants to maintain a series of statistics about various implicated attributes. More specifically, we are interested in approximating the answer to the general query, written in SQL-like format[1] : "select count (distinct A) from R where A implies B", where R is a relation that models a data stream, A, B and C are *sets* of attributes(dimensions) of the relation R. We assume -without loss of generality- that A$\cap$B= $\emptyset$.

In order to fully define the meaning of the predicate *implies*, we first introduce the notion of an itemset and then proceed by defining certain implications between itemsets. Then we proceed by defining the aggregation of such implications over a distributed environment.

6.3.1 Itemsets and definitions

The projection of a single tuple τ of R on the attributes of A is defined as an *itemset* a, and we denote: $a = \tau[\mathtt{A}]$. For example, in the case of the data set in Table 6.1 if A={Source,Destination} the itemset of the first tuple is {S1,D2}.

At any given moment the number of tuples seen so far is denoted by T. The *compound cardinality* $|A|$ of the set of attributes A is the product of the cardinalities of the attributes of A. In our example the compound cardinality is: $|A| = 3 \cdot 3 = 9$, because there are three different sources and three different destinations.

[1]We are not trying to extend SQL but rather use it to describe the class of queries we are addressing

For an itemset a of $\mathtt{A}$ and b of $\mathtt{B}$, we denote as $\langle a, b\rangle$ their association and we define the following:

Implication Set $\kappa(a, \mathtt{B})$: An itemset a of $\mathtt{A}$ may appear with more than one itemsets of $\mathtt{B}$. We define as implication set of an itemset a of $\mathtt{A}$ w.r.t. $\mathtt{B}$, the set of different itemsets of $\mathtt{B}$ it appears with: $\kappa(a, \mathtt{B}) = \{b_i\}, (\tau[\mathtt{A}] = a) \wedge (\tau[\mathtt{B}] = b_i), \tau \in \mathtt{R}$. The cardinality $|\kappa(a, \mathtt{B})|$ of the implication set is called *multiplicity* of a w.r.t to $\mathtt{B}$. For example, the itemset $a = \{$S1,D3$\}$ of $\mathtt{A}=\{$Source,Destination$\}$ has a multiplicity with $\mathtt{B}=\{$Service$\}$, $|\kappa(a, \mathtt{B})| = 2$ since it appears with two different services (WWW and P2P).

Support $\sigma(a)$: An itemset a of $\mathtt{A}$ is said to have support σ when it appears at σ tuples out of T. For example, itemset $a = \{$S1,D3$\}$ of $\mathtt{A}=\{$Source,Destination$\}$ has a support of four, since it appears in four tuples. In the literature (for example association rules) the minimum support is expressed in terms of a ratio over all the tuples. In the case of streams, which are potentially unbounded in size, we chose to define it in terms of an absolute number of tuples. Additionally the relative minimum support has some interesting side-effects that are discussed in Section 6.5.1.

Confidence Level $\lambda(a, b)$: An itemset a of $\mathtt{A}$ and an itemset b of $\mathtt{B}$ have a confidence level $\lambda = \frac{\sigma(a \cup b)}{\sigma(a)}$, where $\sigma(a \cup b)$ is the number of tuples where itemsets a and b appear together. For example, the itemsets $a = \{$S1,D3$\}$ and $b = \{$WWW$\}$, of $\mathtt{A}=\{$Source,Destination$\}$ and $\mathtt{B}=\{$Service$\}$ respectively, have an confidence level $\lambda(a, b) = 2/4$ since itemsets a and b appear together in two tuples over the four tuples of itemset a.

Top-Confidence Level $\ell_c(a, \mathtt{B})$, $c \leq |\kappa(a, \mathtt{B})|$: Assume the sequence of all the

confidence levels of a, i.e the sequence of $\lambda_i = \lambda(a, b_i),\ b_i \in \kappa(a, \mathtt{B})$. We define as: $\ell_c(a) = \sum[\text{top}_c(\{\lambda_1, \lambda_2, ..., \lambda_{|k(a,\mathtt{B})|}\})]$, where the $\text{top}_c(X)$ operator returns the c-biggest items in sequence X. This metric can be used to keep track of approximate *one-to-c* implications, where one itemset a of $\mathtt{A}$ appears with at-most c different itemsets of $\mathtt{B}$ in $\ell_c(a, \mathtt{B})$ percent of the tuples where a appears. For example, for the itemset $a = \{\text{P2P}\}$ of $\mathtt{A}$=$\{$Service$\}$ and $\mathtt{B}$=$\{$Source$\}$ the confidence levels with sources $\{$S1,S2,S3$\}$ it appears with are: $\{2/4, 1/4, 1/4\}$. The top-confidence level for $c = 2$ is: $\ell_2(a, \mathtt{B}) = 2/4 + 1/4 = 75\%$. This means that P2P appears with at most $c = 2$ sources in 75% of all the tuples where P2P appears. The top-confidence level with $c = 3$ in this specific case is 100%, i.e. service P2P appears with at most three different sources in all the tuples of P2P. Similarly the top-confidence level with $c = 1$ is 50%, i.e in half the tuples, P2P appears with only one source.

Implications between itemsets and attributes

An *implication* of an itemset a of $\mathtt{A}$ to $\mathtt{B}$, denoted by $a \rightarrow \mathtt{B}$, holds for a given maximum multiplicity K, a given minimum support Σ and a given minimum top-confidence level Λ_c, when all of the following *implication conditions* are met:

1. *Maximum Multiplicity* K: $|\kappa(a, \mathtt{B})| \leq K$
2. *Minimum Support* Σ: $\sigma(a) \geq \Sigma$
3. *Minimum top-Confidence Level* Λ_c $\ell_c(a, \mathtt{B}) \geq \Lambda_c$

The cardinality $\mathcal{S}$ of the set of itemsets a_i of $\mathtt{A}$ such that $a_i \rightarrow \mathtt{B}$ is the *implication count* for the general query of Section 6.3. When an itemset a_i satisfies

implications conditions (1)-(3), then we say that the itemset *contributes* or *participates* in the implication count.

Note that by definition, we are interested in counting itemsets that satisfy the implication conditions *throughout* the life of the stream (we lift this restriction using sliding windows and incremental counts in Section 6.3.2). This has a direct effect on how the confidence level is interpreted: When an itemset satisfies the minimum support and maximum multiplicity but does not satisfy the minimum top-confidence level then we immediately discard that itemset from the implication count. It is possible that later in the stream, the same itemset may satisfy the top-confidence level condition. However, since the itemset *at least once* did not satisfy all the implication conditions then by definition we do not count its contribution to the implication count.

Example

The above parameters describe a very flexible framework for filtering out noise and defining one-to-many implications.

Consider the network traffic data set described in Table 6.1 and assume that an analyst is interested in identifying how many services are being used *at most two* different sources 80% of the time. The user may also want to consider all services even if they appear for just one tuple but does not want to consider services that are being used at more than five sources.

The corresponding implication conditions are:

Maximum multiplicity is set to five; a service that is being used by more than

five different sources does not contribute in the returned count.

Minimum Support is set to one; meaning that we take into account services even if they appear in just one tuple of the dataset.

Top-Confidence Level is set to 80% for $c = 2$; That means that a service P that contributes in the implication count, appears with at-most $c = 2$ different sources in at-least 80% of all the tuples where P appears.

Let's go over all services in Table 6.1 to see how the above parameters affect the returned count. Service WWW appears in two tuples with only S1 and therefore participates in the count. FTP appears in only one tuple (with S2) and also participates. P2P appears in four tuples with three different stores. The confidence level of $\langle P2P, S1\rangle$ is 2/4, of $\langle P2P, S2\rangle$ is 1/4 and of $\langle P2P, S3\rangle$ is 1/4. Therefore the top-confidence level for $c = 2$ is $2/4 + 1/4 = 75\%$ and service P2P doesn't satisfy the minimum top-confidence level condition. The returned count is two (for services WWW and FTP).

The minimum top-confidence level corresponds to the fact that the user needs to consider only the services that appear with *at most* $c = 2$ different stores. The value of 80% corresponds to the gravity of this constraint. If we change to minimum top-confidence level to 75% then P2P is valid and participates in the count.

The minimum support is used to filter out implications that hold for a very small fraction of the data set. For example, if the user increases the minimum support to two tuples, then the pair $\langle FTP, S2\rangle$ is not valid since it appears in only one tuple.

6.3.2 Incremental and Sliding Queries

Figure 6.1: Incremental maintenance

Our framework provides for implication counts given a reference point in the stream where the counting begins and the implication conditions must hold w.r.t to that reference point. We relax that constraint by using two techniques. The *incremental* technique can answer queries like: *How many new sources with some given implication conditions have appears in the last 1h.* The *sliding window* technique generalizes the incremental technique and provides the support for more general aggregates like moving averages.

In Figure 6.1 we demonstrate a count (ic) at two points t_1 and t_2. In many cases the user is interested in the incremental implication count, which is the distinct count of *new* itemsets that appeared and satisfy the implication conditions between t_1 and t_2. This can be derived by $ic(t_2) - ic(t_1)$.

On the other hand sliding queries where we want to retire old implication counts or compare implication counts with different origins can by supported by

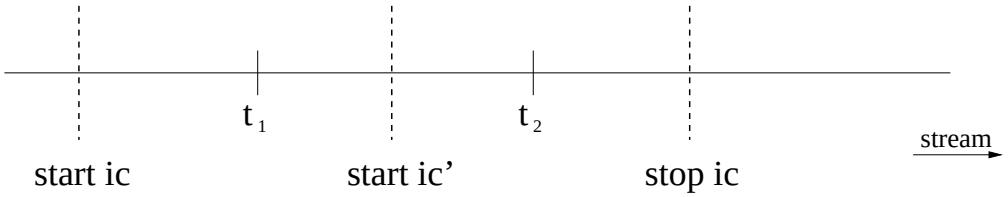


Figure 6.2: Sliding Windows

maintaining a vector of implications counts with different origins and appropriately retiring old ones as depicted in Figure 6.2.

6.4 Algorithm for Implication Counts

In this section we describe an algorithm that can be used to efficiently estimate implication counts. The algorithm uses selective sampling driven by hashing techniques and is based on ideas that are used to estimate the count of distinct elements on a stream or relation using limited memory and processing per data item.

We evaluate analytically the accuracy and describe techniques that can be used to increase the accuracy. Finally we describe the complexity requirements of the algorithm both for total space required and time per data item.

6.4.1 Counting Distinct Elements

The presence of duplicates in the data can traditionally be handled using sorting, sampling or using hash tables. Sorting and using hash tables do not scale well under both memory and time constraints, as those encountered in a streaming environment. Sampling appears as an attractive alternative mechanism. Taking a simple random sample and then extrapolating the answer, however may introduce an arbitrarily large error or require indexed access to the data[HNSS95]. The alternative is to approximate the answer using properties of hash functions [FM85, WVZT90, AMS99, BYJK$^+$02, Gib01].

Basic Probabilistic Counting

One can probabilistically estimate[FM85, AMS99] the number of distinct itemsets in a large collection of data, denoted as *zeroth-frequency moment* F_0, in a single pass using only a small additional storage of space complexity $O(\log |A|)$, where $|A|$ is the compound cardinality of `A`.

The basic counting procedure assumes that we have a hash function that maps itemsets into integers uniformly distributed over the set of binary strings of length L. The function $p(y)$ represents the position of the least significant 1-bit in the binary representation of y.

For each itemset a_i that appears in the stream we keep track of the *maximum* $p_i = p(\text{hash}(a_i))$. Let's consider an initially empty bitmap and define that an itemset a_i is *hashed in* position p_i of the bitmap. By assigning the value one to the corresponding bit in the bitmap, the maximum p_i can then be determined by the position of the most-significant one bit in the bitmap. If the number of distinct elements in M is F_0, then bm[0] (least significant bit) is accessed approximately $F_0/2$ times, bm[1] approximately $F_0/4$ times etc. This leads to Lemma 6.

Lemma 6 *The expected number of values that hash in the cell i of the bitmap is $\frac{F_0}{2^{i+1}}$, where $i = 0$ corresponds to the least significant bit of the bitmap.*

At any given moment, bm[i] will almost certainly be zero if $i \gg \log F_0$ and one if $i \ll \log F_0$ with a *fringe* of zeros and ones for $i \approx \log F_0$. The position R of the leftmost zero value in the bitmap is an estimator of $\log F_0$ with expected value: $\mathbf{E}(R) \approx \log(F_0)$ [FM85, AMS99].

6.4.2 Counting Implications

The basic probabilistic counting procedure can be extended in a straightforward (but inapplicable) manner in order to count implications. The idea is that the basic procedure can be thought of as *recording events*. When we are counting distinct elements, the recorded event is the existence of an itemset that hashes in a cell of the bitmap and it is recorded by assigning one to the value of that cell. Note that we only record events and never erase them.

When counting implications, the recording event is the existence of an itemset that satisfies the implication conditions. Whenever we discover such an itemset we must assign the value of one to the corresponding cell. The problem is that we don't know if an itemset will keep on satisfying the implication conditions in the future. However we can *postpone* the assignment of one to a cell for the time when the user requests the implication count.

We extend the cells of the bitmap so that we can store itemsets in them. When an itemset a_i hashes in a cell, we keep track of all the itemsets of `B` it appears with, postponing the assignment of one or zero to the corresponding cell. When the user asks for the count of itemsets a_i of `A` with the property $a_i \rightarrow$ `B`, we check each cell to see if there is *at least* one a_i such that $a_i \rightarrow$ `B` and we assign a value of one to the corresponding cells. Then -as described in Section 6.4.1- the position of the leftmost zero is an estimator for the implication count.

One obvious optimization is that whenever we can determine that some itemset $a_i \not\rightarrow$ `B` we can remove it from the cell. However the memory requirements of this

algorithm is still $O(K|A|)$, where K is the maximum multiplicity, since we must keep track of every single itemset a_i and the K different itemsets of B it appears with. Otherwise there is not enough information to determine if the implication holds for the itemsets. Such memory requirements are unacceptable, not only because we cannot assume that this memory is available, but also because with such memory requirements we can have an exact implication count.

6.4.3 Counting non-implications

Assume that instead of counting the itemsets $a_i : a_i \rightarrow \mathtt{B}$ we consider the complement problem of estimating the count of itemsets $\bar{a}_i : \bar{a}_i \not\rightarrow \mathtt{B}$. Let's call this problem *Non-implication Counting*. More specifically an itemset a_i has the property $\bar{a}_i \not\rightarrow \mathtt{B}$ with respect to the implication conditions in Section 6.3.1 when it satisfies the minimum support requirement but does not satisfy the maximum multiplicity or the minimum top-confidence level. In this section we describe how we can bound the required memory and still get an estimate of the non-implication count.

The recording event is the existence of an itemset $\bar{a}_i : \bar{a}_i \not\rightarrow \mathtt{B}$. Unlike the case when counting implications , we can now assign the value of one to a cell as soon as we discover such an itemset. Once an itemset does not satisfy the implication conditions we know that it will never satisfy them in the future. Below we define the *fringe zone* and we show that for all non-implication counts -except for very small counts- the size of the fringe zone is quite small.

We can observe in Figure 6.3 that the general format of the bitmap while

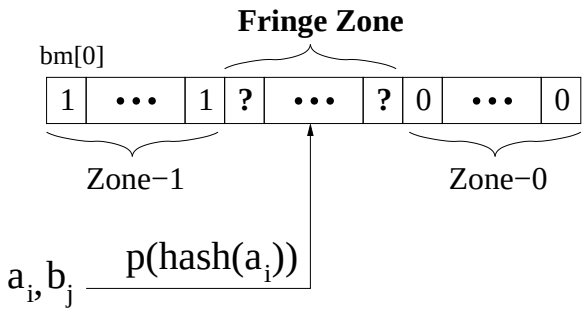


Figure 6.3: Fringe Zone

performing the probabilistic counting has three "zones".

The bitmap consists of cells bm[i], where the leftmost cell bm[0] is the least significant one. Zone-1 is always filled with ones -because we found an itemset $\bar{a}_i : \bar{a}_i \not\rightarrow \mathtt{B}$, while Zone-0 is always filled with zeros -because the corresponding cells are empty-. Note that these are the *only* two cases where we can assign a value of one or zero to a cell.

The *fringe zone* lies between Zone-1 and Zone-0. In its boundaries (at least) we cannot determine the existence of an itemset $\bar{a}_i$, i.e. all itemsets in the zone *so far* imply B and therefore we must keep track of all the itemsets a_i and the corresponding itemsets of B until we determine there is at least one $\bar{a}_i$.

Size of the Fringe Zone

In order to calculate how big the fringe zone is, lets assume that we are interested in estimating the non-implication count: $\bar{a}_i \not\rightarrow \mathtt{B}$ for some set of attributes A and B. Let $F_0(A)$ be the number of distinct elements of A and let $\bar{\mathcal{S}}$ be the non-implication count. The size in cells of the fringe zone is quantified with very high probability by the following lemma:

Lemma 7 *The size $\mathcal{F}$ of the fringe zone is: $\mathcal{F} = -\log q$, where $q = \frac{\bar{S}}{F_0(A)}$.*

Proof: This is a direct effect of the way the function $p(\text{hash}(a_i))$ distributes itemsets a_i of A to cells. We expect $2^{qF_0(A)-1}$ itemsets $\bar{a_i} : \bar{a_i} \not\rightarrow$ B in the first (less significant) cell, $2^{qF_0(A)-2}$ in the second etc. and therefore there are $\log(qF_0(A))$ such cells -that correspond to the Zone-1 of the bitmap-. The whole bitmap (ignoring Zone-0) holds $\log(F_0(A))$ cells and therefore the size of the fringe zone is $\log(F_0(A)) - \log(qF_0(A)) = -\log q$.

This observation demonstrates that the size of the fringe zone is quite small for almost all non-implication counts and that it logarithmically increases when the ratio $q \rightarrow 0$. For example, all non-implication counts greater than 1/16 of $F_0(A)$ correspond to a fringe zone of only four cells.

We must point out that the bounds given in Lemma 7 are very pessimistic and that the size of the fringe zone is actually smaller. For example in the case where all distinct elements satisfy the implication condition ($q = 0$) then the counting procedure degenerates to the basic probabilistic counting described in [FM85], where the fringe zone size is quantified with high probability by $O(\log\log F_0(A))$.

Bounding the size of the Fringe Zone

This gives rise to the idea of bounding the size of the fringe zone to a specific size. This limits the amount of itemsets we must keep in memory.

In section 6.4.1 we mention that the index of the cell in the bitmap is determined by function $p(\text{hash}(a_i))$, which represents the position of the least significant 1-bit in the binary representation of $hash(a_i)$. From Lemma 6 we know that as we

move to higher-order bits the number of a_i's that get hashed in the corresponding cells decreases exponentially. For example, if the number of distinct a_i's is 128, we know that about 64 will get hashed to the first -from left to right- cell, 32 to the second, ..., 2 to the 6th and only 1 to the 7th cell. Note that this distribution is the same regardless the original distribution of a_i values or the frequency that a_i's appear. In [AMS99], there is a discussion about using linear hash functions in order to accomplish that.

Assume that we arbitrarily choose to define that the fringe zone has a fixed size of four cells. We expect that in the rightmost cell of the fringe zone only one a_i will get hashed in, in the immediate left cell two, ..., and to the leftmost cell of the fringe we expect $2^3 = 8$ different a_i's. We keep track of *every* single itemset a_i that gets hashed in the cells of the fringe zone, as well as all the itemsets of `B` there itemsets appear with. This allows us to check if there is at least one $\bar{a}_i : \bar{a}_i \not\rightarrow$ `B` in a cell and therefore assign it a value of one. If this happens for the leftmost cell in the fringe zone, then we "float" the fringe zone to the right, by increasing the size of Zone-1 by one. By "limiting" the size of the fringe to four cells we bound the amount of memory required to make a decision. Note that for each different itemset that hashes in a cell we need to keep at most K different itemsets of `B` it appears with. Therefore, in our case where the fringe has a size of four cells, we need at most $(2^0 + 2^1 + ...2^3) \times K$ (i.e $O(K)$) itemsets to be stored in the corresponding cells. Note that the actual memory required is much less since we can free all the memory required by cells in the fringe that have been assigned a value of one.

We can also double the allocated memory (keeping the asymptotic require-

ments unaffected) to accommodate deviations from the expected distributions due to inefficiencies of the hash function.

Estimation Error due to Fringe Size Fixation

When there is not enough space in a cell to accommodate an itemset a_i we *arbitrarily* assign a value of one to the cell and shift the floating fringe to the right. This can happen under two different situations. In the first one, it just happens to hash an itemset to a cell in the fringe zone that already accommodates the expected number of itemsets. In the second situation, an itemset is hashed to Zone-0 and the fringe zone must float to the right to accommodate that itemset. Remember that -by definition- the rightmost cell of the fringe is always the rightmost cell where an itemset has been hashed. As the fringe floats to the right, the leftmost cells of the fringe now belong to Zone-1. This step is that "fixates" the length of the fringe zone. The only effect that it has, is the introduction of an error for small non-implication counts that cannot be "managed" by the chosen fringe zone size.

Note that *no error* is introduced when the fringe zone has a size of at least $\mathcal{F} = -\log(q)$. By limiting the size of the fringe, we essentially limit the *minimum* non-implication count we can estimate. If the fringe size is $\mathcal{F}$ then the minimum non-implication count we can estimate with the basic probabilistic counting algorithm is $2^{-\mathcal{F}} \cdot F_0(A)$. Smaller non-implication counts are "mapped" to that specific value.

For example, with a fringe size $\mathcal{F} = 4$ we can estimate an implication count accurately if that count is bigger than than $6.25\% \cdot F_0(A)$, ($2^{-4} \approx 6.25\%$). Without changing the asymptotic memory requirements one can increase the size of the

fringe to eight, in order to estimate accurately very small counts, $> 0.4\% \cdot F_0(A)$, ($2^{-8} \approx 0.4\%$). Smaller counts than that, are mapped to the same value: $2^{-8} \cdot F_0(A)$.

Tracking Non-Implication Conditions

In this section we describe how we can keep track if an itemset $\bar{a}_i$ $\bar{a}_i \not\rightarrow$ B given the implication conditions.

For each itemset a_i that hashes in the fringe zone we keep track of all itemsets of B it appears with. Therefore the implication count $\kappa(a_i)$ is known at any moment. The support $\sigma(a_i)$ of that itemset can be represented by a counter that is increased every time itemset a_i is hashed in the cell. For the confidence level $\lambda(a_i, b)$ with an itemset b of B we use a counter that represents the support $\sigma(a_i \cup b)$. Every time the itemset a_i appears with b in the stream we increase the corresponding counter.

At any moment the corresponding confidence level is: $\lambda(a_i, b) = \frac{\sigma(a_i \cup b)}{\sigma(a_i)}$, which can be determined just by dividing the corresponding counters. The top-confidence level of an itemset a_i can therefore be determined by summing the biggest $\lambda(a_i, b)$ at any given moment.

Whenever an itemset a_i satisfies the minimum support condition but does not satisfy the rest of the implication conditions we assign a value of one to the corresponding cell.

6.4.4 Deriving Implication Counts

So far we have described how to get an estimate of the non-implication count $\bar{\mathcal{S}}$ of A to B. The implication count can be derived by subtracting the non-implication

count from the distinct count $F_0^{sup}(A)$ of itemsets $a_i \in \mathtt{A}$ that *satisfy* the minimum support requirement, i.e: $\mathcal{S} = F_0^{sup}(A) - \bar{\mathcal{S}}$.

We can have an estimate of distinct count $F_0^{sup}(A)$ without using any additional memory from the bitmap used to estimate $\bar{\mathcal{S}}$, by virtually assigning a value of one to each cell in the fringe zone where at least one itemset of a_i, that meets the minimum support condition, is hashed in. The cells in Zone-1 by definition have at least one itemset that satisfies the minimum support condition.

6.4.5 Algorithm NIPS/CI

Algorithm 7 referred to as NIPS (Non-Implication Probabilistic Sampling) gives the complete algorithm using a floating fringe for performing the probabilistic sampling for non-implication counts with given implication conditions. This algorithm is designed to sample a small $O(K)$ number of pairs (a_i, b_j), based on the hash representation of a_i over a data stream. Any time a tuple arrives, the bitmap is updated accordingly.

In line 2 we project the tuple to the attributes of `A` and `B` respectively. Then we calculate the position i of the cell where itemset a is hashed. If the position i is in Zone-0 -right of the fringe zone, where no item has been hashed yet- then we "float" the fringe zone to the right by making position i its rightmost cell. In the process of floating, the leftmost cells of the fringe zone that become part of Zone-1 are cleared of all itemsets inside and are set to value 1. As explained in Section 6.4.3, this process introduces an error *only* when counting very small non-implications and

Algorithm 7 NIPS: Non-Implication Probabilistic Sampling

Input: M:stream of tuples
Input: `A,B`: set of attributes of M
Input: K, Σ, Λ_C: Implication Conditions
State: bm: bitmap of L cells

1: **for** each tuple t **do**
2: $a \leftarrow t$[`A`];$b \leftarrow t$[`B`];i $\leftarrow p(hash(a))$
3: **if** i in Zone-0 **then**
4: float fringe by making i its rightmost cell
5: **end if**
6: **if** i in fringe zone and bm[i].value=0 **then**
7: bm[i].supp$[(a,b)] \leftarrow$ bm[i].supp$[(a,b)]$+1
8: bm[i].supp$[a] \leftarrow$ bm[i].supp$[a]$+1
9: curConf $\leftarrow \Lambda_C$
10: **if** bm[i].supp$[a] > \Sigma$ **then**
11: curConf $\leftarrow \frac{\text{Sum}[\ \text{top}_c\ \text{bm[i].supp}[(a,b)]\ ]}{\text{bm[i].supp}[a]}$
12: **end if**
13: **if** (curConf $< \Lambda_C$)
or (bm[i].overflowed) **then**
14: bm[i].value=1
15: free all the memory allocated for the cell bm[i]
16: **if** bm[i] is the leftmost in the fringe **then**
17: float fringe one cell to the right
18: **end if**
19: **end if**
20: **end if**
21: **end for**

the size of the fringe zone is not appropriately set. In lines 7 and 8 the counters that represent the current support of the itemsets a and $a \cup b$ are increased. Line 11 calculates the top-confidence level of itemset a. In lines 14 to 17 a cell is assigned the value one, if we have found an itemset a that either does not imply `B` or if there is no room in the corresponding cell. Additionally the fringe zone is shifted to the right if necessary.

Algorithm 8 referred to as CI (Counting Implications) returns an estimate of the implication count $\mathcal{S}$ and is designed to work with the bitmap used in (and in parallel with) algorithm NIPS. Whenever the user wants an estimate of the current implication count she runs CI on the bitmap of NIPS.

Algorithm 8 CI:Counting Implications

Input: K, Σ, Λ_C: Same implication conditions used in NIPS
Input: bm: bitmap of L cells used in NIPS
1: $R_{F_0^{sup}(A)} \leftarrow 0$
2: **while** exists a_i in bm$[R_{F_0^{sup}(A)}]$ s.t. supp$[a_i] > K$ and $R_{F_0^{sup}(A)} < L$ **do**
3: $\quad R_{F_0^{sup}(A)} \leftarrow R_{F_0^{sup}(A)} + 1$
4: **end while**
5: $R_{\bar{\mathcal{S}}} \leftarrow 0$
6: **while** bm$[R_{\bar{\mathcal{S}}}]$.value=1 and $R_{\bar{\mathcal{S}}} < L$ **do**
7: $\quad R_{\bar{\mathcal{S}}} \leftarrow R_{\bar{\mathcal{S}}} + 1$
8: **end while**
9: return $2^{R_{F_0^{sup}(A)}} - 2^{R_{\bar{\mathcal{S}}}}$

Lines 1 to 3 find the position $R_{F_0^{sup}(A)}$ that corresponds to the number of distinct elements $F_0^{sup}(A)$ that satisfy the minimum support requirement. Lines 5 to 7 similarly locate the position $R_{\bar{\mathcal{S}}}$ that corresponds to the non-implication count. Line 9 returns the estimate of the implication count as described in Section 6.4.4.

6.4.6 Space and Time Complexities

The basic probabilistic counting algorithm has a space complexity (in bits) of $O(\log |A|)$ where $|A|$ is the compound cardinality of `A`.

In order to estimate the implication count $\mathcal{S}$, Algorithm NIPS , in addition to the memory $O(\log \log |A|)$ required for the counter for Zone-1 and the $O(\log |A|)$ memory required for the hash function, requires enough memory to accommodate all the counters for the pairs (a, b) that hash in the cells of the fringe zone. The distinct number of a's that hash in the fringe zone is bounded by the fringe zone (for example for a fringe size of eight we expect about $\sum_{i=0}^{7} 2^i = 255$ different a's). We can actually double or even triple the expected number of a's, without affecting the asymptotic complexities, in order to accommodate more a's due to inefficiencies

of the hash function.

The number of distinct b's that corresponds to each a is bounded by the maximum multiplicity K, i.e. there are at most K different such b's. The number of counters for each cell then is $O(K)$. In general we need $O(\log T)$ bits to represent each counter, where T is the number of tuples of the dataset or the data stream. Therefore the total space (in bits) complexity of algorithm 7 is: $O(K \cdot \log T + \log\log|A| + \log|A|)$,where K is the maximum multiplicity, T is the number of tuples and $|A|$ is the compound cardinality of $\mathtt{A}$. We do not include the $2^{\mathcal{F}}$ term since the size $\mathcal{F}$ of the fringe zone is fixed and usually a value of four is sufficient to estimate very large implication counts as described in Section 6.4.3.

By using hash tables to locate a counter in a cell given a pair (a, b) or an a, and a priority queue to handle the top_c operator, $c \leq K$ counters for a cell, the time complexity of the algorithm per data item is: $O(K \cdot \log K)$

The number of entries (a_i, b_j) that NIPS holds in memory is bounded by the fringe zone size and the maximum multiplicity condition. For example, for $\mathcal{F} = 4$, the number of entries in the bitmap is at most $15 \cdot K$. The above complexities demonstrate the scalability potential of algorithm NIPS. One can estimate accurately any implication counts for arbitrarily big attribute cardinalities or number of tuples.

6.4.7 Approximation

In this section we discuss how an algorithm approximates a value and we show how existing techniques can be used in order to get more accurate results based on algorithm NIPS/CI.

A probabilistic algorithm (ϵ, δ)-approximates a value A if it outputs a value $\tilde{A}$ such that: $P[\left|A - \tilde{A}\right| \leq \epsilon A] \geq 1 - \delta$ The parameters ϵ, δ are called *approximation* parameter and *confidence* parameter respectively. For example, if a user requests $\epsilon = 10\%$ and $\delta = 1\%$ then the algorithm should return an estimate $\tilde{A}$ that is at most 10% relatively off the actual value A with probability at least 99%.

Approximating Non-Implication Counts

The basic probabilistic algorithm is shown[AMS99] to approximate the number of distinct elements F_0 (zeroth frequency moment) in a different manner: $P[\frac{1}{c} \leq \frac{\tilde{F_0}}{F_0} \leq c] \geq \frac{c-2}{c}, \forall c > 2$ by using a linear hash function. In [BYJK$^+$02] techniques are presented that can be used to (ϵ, δ)-approximate F_0 based on the algorithm in Section 6.4.1.

NIPS approximates the non-implication count in exactly the same manner with the basic probabilistic algorithm under the condition that the non-implication count is large enough for the chosen size of the fringe zone. The same techniques used in [BYJK$^+$02] can be applied to get an (ϵ, δ)-approximation of the non-implication count.

The implication count is determined by subtracting two (ϵ, δ)-approximations, namely the number of distinct itemsets, that satisfy the minimum support condition, and the non-implication count. This operation however *does not* maintain (ϵ, δ)-approximation, since the relative error can grow arbitrarily large, when the non-implication count is very close to the number of distinct elements. This essentially means that the relative error for very small implication counts (close to zero) can be unbounded.

For a pragmatic approach however this is not an issue. We have already made the assumption that the user is not interested in very small non-implication counts in order to fixate the size of the fringe zone. We can make the assumption that the user is not interested in very small implication counts (very close zero) as well and in the experiments section we demonstrate that for a wide range of implication counts the estimates returned by algorithm 8 are very accurate.

6.5 Frequent Itemsets

In this section we extend the algorithms "Lossy Counting" and "Sticky Sampling" introduced in [MM02] so that they identify itemsets that satisfy given implication conditions. Then we discuss the advantages and disadvantages compared to NIPS/CI and point out why they cannot be applied successfully to the problem of estimating implication counts.

6.5.1 Implication Lossy Counting

The Implication Lossy Counting(ILC) algorithm is deterministic and requires at most: $\frac{K}{\epsilon}\log(\epsilon T)$(see [MM02]) sampling entries in order to compute an (ϵ, δ)-synopsis[2], that can be used to identify the implicated itemsets, for given implications conditions K, Σ_{rel}, Λ. It is important to point out that the minimum support condition is required to be specified *relatively* to the current number of tuples T in the stream, and that the approximation parameter ϵ must satisfy: $\epsilon \ll \Sigma_{rel}$. These requirements have some very interesting side-effects discussed in Section 6.5.1.

The stream is conceptually divided into buckets of width $w = \lceil \frac{1}{\epsilon} \rceil$. The current bucket is denoted by $b_{current}$. The algorithm samples entries of the form [a_i,support,Δ] and [(a_i, b_j),support,Δ], where Δ is the maximum possible error in the support. For each pair (a_i, b_j) that arrives in the stream we check if there is an entry for a_i and if it is we update the support of both a_i and (a_i, b_j). Otherwise we create two new entries [a_i,1,$b_{current}$-1] and [(a_i, b_j),1,$b_{current}$]. The supports of the itemset a_i and the pairs (a_i, b_j) allow us to check if the itemset a_i satisfies the implication conditions, as explained in Section 6.4.3. If we determine that an itemset a_i satisfies the minimum support requirement but not one of the remaining implication conditions then we mark the corresponding sample entry as dirty and delete all the pair entries for that itemset a_i. At bucket boundaries we prune all non-dirty entries of the form [a_i,support,Δ] where support + $\Delta \leq b_{current}$. For each non-dirty itemset a_i that is deleted we also remove the corresponding entries [(a_i, b_j),support,Δ]. When the user requests the implicated itemsets we output all

[2] $\delta = 1$ in this case

non-dirty a_i's with support $\geq (\Sigma_{rel} - \epsilon)T$.

Algorithm ILC has two differences with the original Lossy Counting algorithm, first we sample supports and errors for both itemsets and pairs of itemsets and second we mark an itemset as dirty -and remove all the corresponding pairs from the samples- as soon as we determine that the itemset does not satisfy the implication conditions.

It is possible to make the same modifications to the "Sticky Sampling" [MM02] algorithm in order to identify implicated itemsets, but the issue with the relative minimum support remains.

Relative Minimum Support Issues

The ILC algorithm returns the *actual itemsets* that satisfy any given implication conditions and not just their count. Although the complexity appear quite small (actually, in [MM02] it is shown that the required memory is much less than the worst case, which corresponds to a rather pathological situation), it does not tell the whole truth.

More specifically, one problem is that every single itemset that satisfies the minimum support Σ_{rel} has to stay in memory (marked dirty even if it doesn't satisfy the rest of the implication conditions). This property, although desirable for certain applications, limits the applicability of ILC due to the amount of memory required. In the worst case the number of entries that need to be sampled is in the order of the number of different itemsets in the stream. For conditional implications, the compound cardinality of the participating attributes can be quite high. The

only way to limit the memory used, is by increasing the Σ_{rel} minimum support condition, which implies that the contribution of many implications that hold for a smaller amount of tuples is totally lost (see [WZS02] for an example application).

The major problem however, is the relative nature of the minimum support itself. As the stream evolves, the number of tuples that correspond to the given implication conditions implicitly increases. The side-effect is that the contribution of "small" implications to the implication count is lost, although these specific implication may hold for a quite large (and continuously increasing) number of tuples.

The relative nature of the minimum support in the ILC algorithm cannot be removed since the approximation parameter ϵ must remain constant (the size of the buckets is a function of ϵ) *and* satisfy: $\epsilon \ll \Sigma_{rel}$, throughout the execution of the algorithm. The same holds for the extended sticky sampling algorithm.

On the other hand the NIPS/CI algorithm returns an accurate estimation of the implication count by keeping only $O(K)$ samples in memory, regardless of how small the minimum support requirement is, capturing the cumulative effect of small implications throughout the life of the stream.

6.6 Experiments

In this section, we present an extensive empirical study on the accuracy of the estimation for implication counts. The first section demonstrates the results using artificially generated datasets, while the second section describes the result obtained from real-world datasets. For the synthetic datasets we imposed implication patterns

—of known count— on the datasets and used both the bounded and the unbounded fringe estimator to estimate the strength of the correlations and report the relative error and the deviation. For the real datasets, we used an exact method (based on hash tables) for calculating the implication count and compared with the estimation of both the bounded and the unbounded version of our estimator.

6.6.1 Synthetic Dataset One

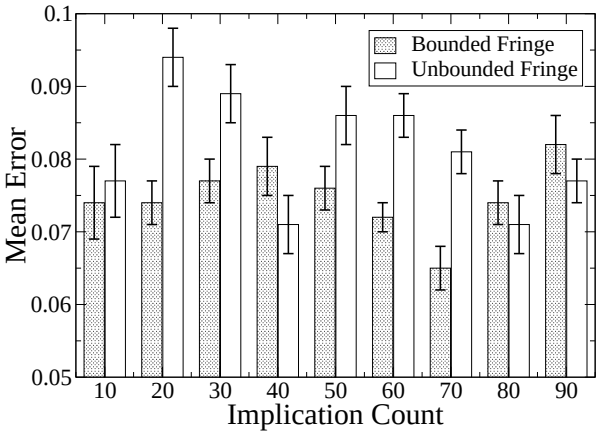


Figure 6.4: Dataset One with $c = 1$ and $|A| = 100$

We conducted a series of experiments to verify the error bounds of NIPS/CI with a fixed fringe size of four. We used a varying cardinality for attribute A and the imposed implications had a variable count between 10% and 90% of $|A|$, for various one-to-c implications, where $c = 1, 2, 4$. We increased the accuracy of our estimation to approximately 10% using stohastic averaging([FM85]). More specifically for the accuracy of 10%, we used 64 bitmaps with a fringe zone of size four (i.e. there was available space for 1920 itemsets in memory).

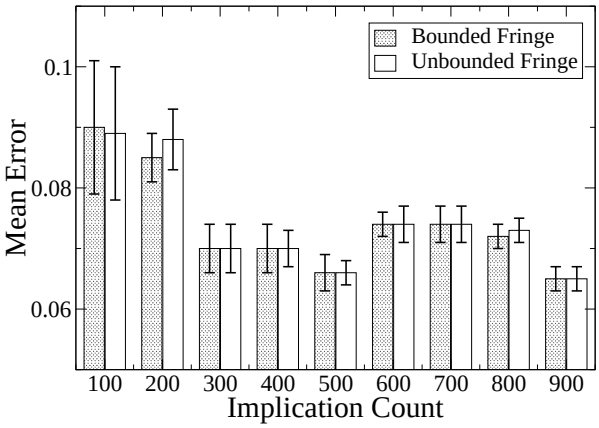


Figure 6.5: Dataset One with $c = 1$ and $|A| = 1000$

We chose a minimum top-confidence level Λ of 90% while the itemsets a's that should participate in the count, were imposed to have a top-confidence level of 92%. The chosen minimum support was 50 tuples. The maximum multiplicity was chosen to be equal to c.

To test the accuracy of the algorithms with respect to the implication conditions we also imposed a "noise". Some itemsets a_i where created in a way that breaks at least one implication condition and therefore they should not participate in the count.

For example, itemsets that did not participate because of the maximum multiplicity condition were imposed to appear with a number u of different itemsets of `B` that was uniformly distributed as: $c + 1 \leq u \leq c + 10$.

Each combination of these parameters was tested one hundred times. The experiments were performed by generating random numbers using a random number generator to simulate the itemsets. Specifically the experiments were organized as

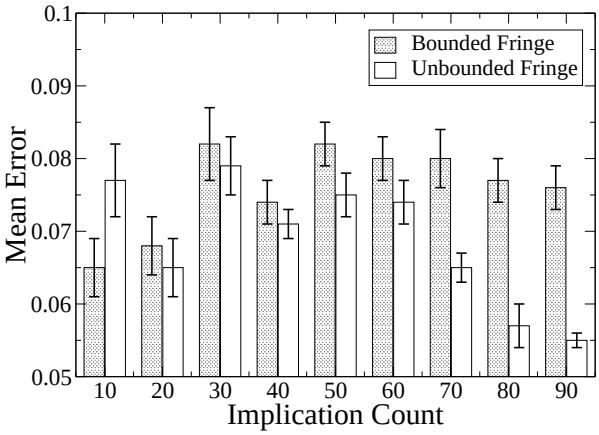


Figure 6.6: Dataset One with $c = 2$ and $|A| = 100$

follows: Pick a cardinality size $|A|$, an implication count $\mathcal{S}$ and a c. Generate $\mathcal{S}$ different itemsets a_i. For each a_i create at most c (uniformly distributed in the range $[1, c]$) different b_j. For each combination (a_i, b_j) write 50 tuples. Then for each a_i create four b'_j different than all b_j's created before. And write the four tuples (a_i, b'_j). This step creates $\mathcal{S}$ itemsets a_i such that $a_i \rightarrow \texttt{B}$ with a minimum support of 54 tuples and a top-confidence level of $50/54 = 92\%$ and therefore these itemsets participate in the implication count. The number of tuples created by this step is: $\mathcal{S} \cdot 50((c+1)/2 + 4)$.

The rest of the steps create itemsets that *should not* participate in the count. We create three different kind of tuples that break one implication condition. The relative weight of each kind is 1/3.

Generate $(|A| - \mathcal{S})/3$ pairs (a_i, b_j) where each a_i is different than all itemsets of `A` created before. As in the previous step, for each a_i create at most c different b_j. For each combination (a_i, b_j) write 50 tuples. Then for each a_i create *eight* b'_j

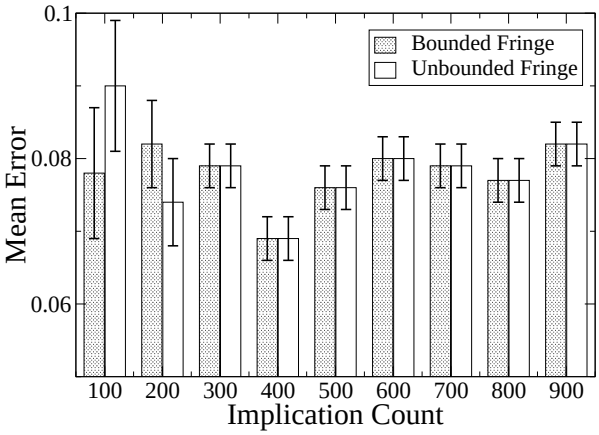


Figure 6.7: Dataset One with $c = 2$ and $|A| = 1000$

different than all b_j's created before. And write the eight tuples (a_i, b'_j). This step creates $(|A| - \mathcal{S})/3$ new itemsets of `A` that should not participate in the implication count because they do not satisfy the minimum top-confidence level, although they satisfy both the minimum support and the maximum multiplicity constraint. The number of tuples created by this step is: $(|A| - \mathcal{S})/3 \cdot 50((c+1)/2 + 8)$.

Generate $(|A| - \mathcal{S})/3$ pairs (a_i, b_j) where each a_i is different than all itemsets of `A` previously generated and each one appears with u different b_j, where: $c + 1 \leq u \leq c + 10$. Write 50 such tuples. This step creates $(|A| - \mathcal{S})/3$ new itemsets of `A` that should not participate in the implication count because they do not satisfy the maximum multiplicity condition. The number of tuples created by this step is: $(|A| - \mathcal{S})/3 \cdot 50(c + 5.5)$ Generate $(|A| - \mathcal{S})/3$ pairs (a_i, b_j) where each a_i is different than all itemsets of `A` previously generated. For each pair (a_i, b_j) write 40 tuples. This step creates $(|A| - \mathcal{S})/3$ new itemsets of `A` that should not participate in the implication count because they do not satisfy the minimum support requirement.

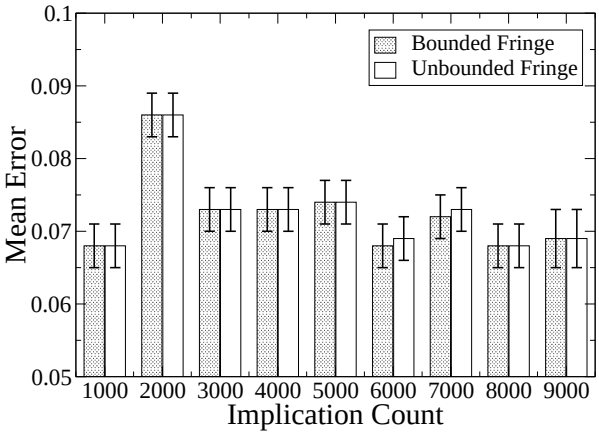


Figure 6.8: Dataset One with $c = 1$ and $|A| = 10,000$

The number of tuples created by this step is: $(|A| - \mathcal{S})/3 \cdot 40$.

Shuffle the output file. This step just demonstrates that the operation of the algorithm is independent to the ordering of the tuples. Estimate the implication count using algorithm NIPS/CI with a fringe size of four and also without a bounded fringe. Perform one hundred such experiments and calculate the mean and the standard deviation of both estimations.

The total number of tuples for each experiment can be derived by adding the partial number of tuples created in each step. For example, for $|A| = 10000$, $\mathcal{S} = 5000$ and $c = 4$ the average number of tuples for the corresponding experiment was $\approx 3,108,333$. A minimum support of 50 tuples for this case corresponds to only $\approx .001\%$ of the tuples, demonstrating that in the implication count contribute even implications that hold for a very small number of tuples.

Figures 6.4,6.6 and 6.12 show the results for $c = 1, 2, 4$, for varying cardinalities $|A|$. The x-axis corresponds to the *actual* implication count of the dataset as that

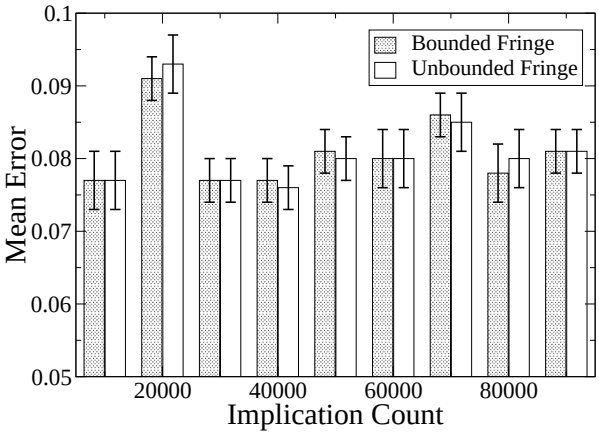


Figure 6.9: Dataset One with $c = 1$ and $|A| = 100,000$

was imposed by the creation process. The y-axis denotes the mean relative error as it is calculated by running one hundred experiments. We used the following formula to estimate the mean relative error: relative error $= \frac{|\text{Actual } s - \text{Measured } s|}{\text{Actual } s}$

Graphs "Bounded Fringe" express the experimental results for the case of a fringe with size $\mathcal{F} = 4$, while graphs "Unbounded Fringe" demonstrate the result for the case of an arbitrarily large fringe. The error bars correspond to the statistical deviation of the mean error as that was computed by one hundred such experiments.The deviation is generally negligible, which means that the error of the estimated $\mathcal{S}$ is always very close to the mean error. We also observe that the difference between the estimation using a bounded fringe of size four and a unbounded one, is negligible for a very wide range of implication counts and therefore a size of four for the fringe zone is sufficient to provide very accurate results for most applications.

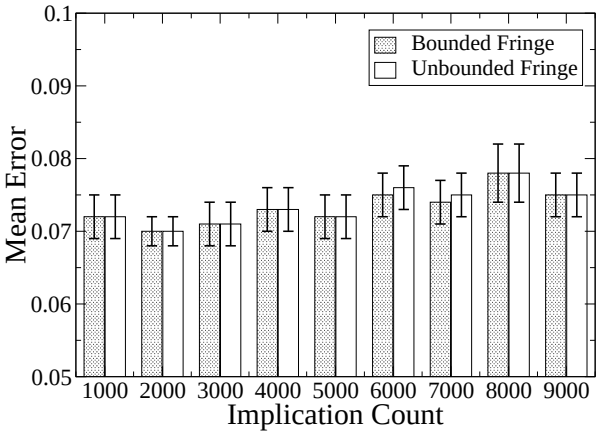


Figure 6.10: Dataset One with $c = 2$ and $|A| = 10,000$

6.6.2 Real-world datasets & Algorithmic comparison

We compare our estimates with the results taken using *Distinct Sampling* (DS)[Gib01] which has been shown provide highly-accurate estimates for distinct value queries and event reports. This algorithm outperforms other estimators that are based on uniform sampling[CCMN00, CMN98] even when using much less sample space. We also provide comparison with our Implication Lossy Counting (ILC) algorithm (described in Section 6.5.1) which is based in the Lossy Counting algorithm introduced in [MM02].

Dimension	Cardinality
A	1557
B	2669
C	2
D	2
E	3363
F	131
G	660
H	693

Table 6.3: Cardinalities

Tuples	Workload A $A * B * E \rightarrow G$	Workload B $E \rightarrow B$
134,576	608	50
672,771	12,787	125
1,344,591	34,816	152
2,690,181	84,190	165
4,035,475	132,161	182
5,381,203	187,584	188

Table 6.4: Impl. counts w.r.t tuples

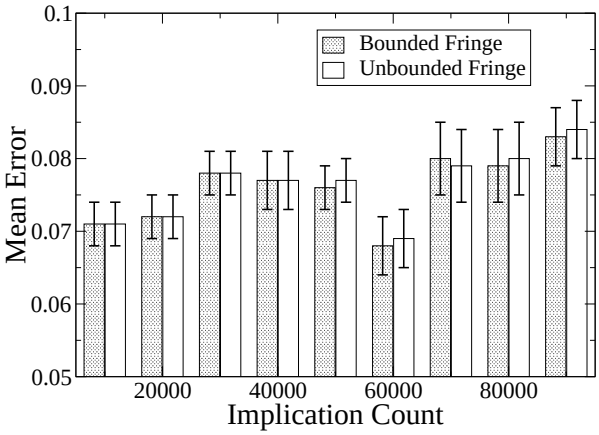


Figure 6.11: Dataset One with $c = 2$ and $|A| = 100,000$

NIPS/CI #bitmaps	64
NIPS/CI K	2
DS sample size	1920
DS bound t	39
ILC ϵ	0.01

Table 6.5: Algorithm Parameters

For this series of experiments we used a real dataset of eight dimensions which was given to us by an OLAP company, whose name we cannot disclose due to our agreement. The cardinalities of the dimensions are presented in Table 6.3. The parameters of the algorithms are presented in Table 6.5. For NIPS/CI, we used 64 concurrent bitmaps with a fringe size of four thus requiring memory enough to hold $(2^4 - 1) \cdot 64 \cdot K = 1920$ itemsets. We expect that the "averaging"([BYJK+02, FM85]) over these many bitmaps will result in an error less than 10%. We used the exact same sample space for DS. The bound parameter t for DS was set to $\lceil 1920/50 \rceil$ following the suggestion in [Gib01]. For ILC we used an approximation parameter $\epsilon = 0.01$ which increases the memory requirements of ILC relative to

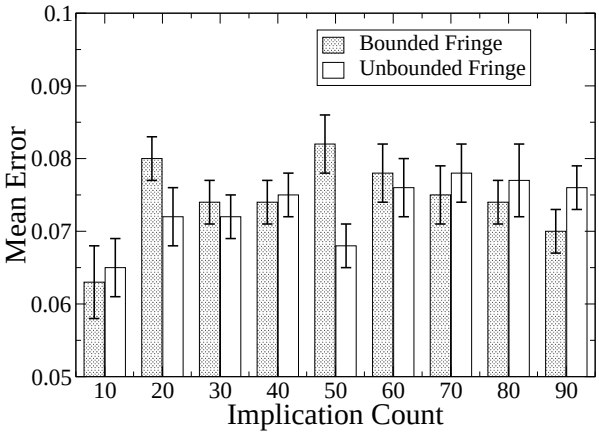
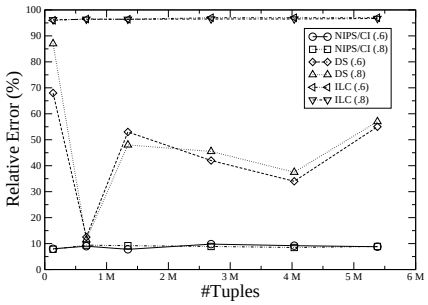


Figure 6.12: Dataset One with $c = 4$

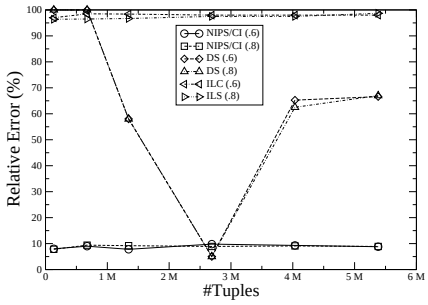
those of NIPS/CI or DS. On the average, ILC used more than twice the memory that NIPS/CI and DS used. For example for the experiment in Figure 6.14 it used more than 8,000 entries.

We evaluate the results of the algorithms with respect to the number of tuples, the cardinality of the participating dimensions and the implication conditions. To simulate a real data stream scenario we "tracked" the conditional implication counts of $A*B*E \rightarrow F$ and the unconditional $B \rightarrow E$ using the aforementioned algorithms. The first workload corresponds to quite large compound cardinality while the second to very moderate cardinalities. Table 6.4 presents the actual aggregates for various instances of the stream for $\Sigma = 5$ and $\Lambda_1 = 60\%$. We believe that most workloads fall somewhere in the middle with respect to the complexity of the wanted implications and the size of the returned counts.

Figure 6.13 depicts the relative error as the stream evolves for workload A, using the algorithms DS, NIPS/CI and ILC for different implication parameters. In



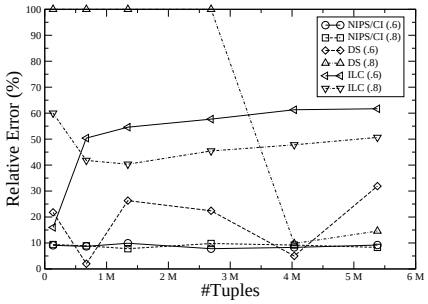
(a) $\Sigma = 5$



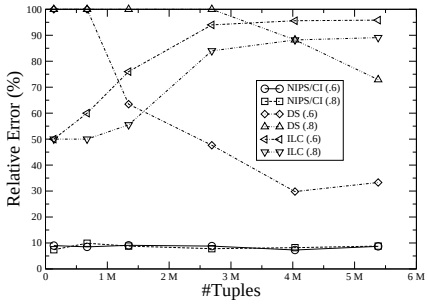
(b) $\Sigma = 50$

(A) Workload A

Figure 6.13: Relative Error vs stream size



(a) $\Sigma = 5$



(b) $\Sigma = 50$

(A) Workload B

Figure 6.14: Relative Error vs stream size

Figure 6.13(a) we show the results for minimum support $\Sigma = 5$ and $\Lambda_1 = 60\%$ or $\Lambda_1 = 80\%$. The different Λ_1 are encoded in the parentheses next to identification of the algorithm in the legend of the graph. In Figure 6.13(b) we increased the minimum support to $\Sigma = 50$. We observe that the behavior of DS varies widely while NIPS/CI remains always below the expected 10% error. DS actually keeps a sample of the distinct elements seen so far and tries to "scale" the implication count that holds for that sample to the whole set of distinct elements. In most cases the

data in the sample is not representative of the implication. The situation for DS is exacerbated when the minimum support increases, where quite a lot of samples do not participate in the count making the "scaling" even more error-prone. Algorithm ILC in all cases returned very erroneous results although it used much more space than NIPS/CI and DS, since it tries to store not the implication counts but the actual implicated itemsets. In these workload the implicated itemsets overwhelm its available memory (which is actually larger than the amount given to NIPS/CI and DS).

In figure 6.14 we present the results of the algorithms for workload B. The situation is still in favor of NIPS/CI whose relative error remains always close to the expected 10% unlike DS who returns highly skewed errors even though the domain cardinalities are much smaller and therefore keeps in the sample space much more data. As expected from the analysis the error guarantees of NIPS/CI are virtually unaffected by changes in the cardinalities or the number of tuples seen so far in the stream. ILC returns very erroneous results although now the cardinalities and the implicated items are much smaller compared to those of workload A. The reason is not only because it keeps too much information in memory (i.e. all the implicated itemsets) (while both NIPS/CI and DS only hold a "mantissa" for the count) but also because the constraint $\epsilon \ll \Sigma_{rel}$ is broken as the number of tuples increases.

6.7 Related Work

There are unique challenges in query processing for the data stream model. Most challenges are the result of the streams being potentially unbounded in size. Therefore the amount of the storage required in order to get an exact size may also grow out of bounds. Another equally important issue is the timely query response required although the volumes of data the need to be processed is continually augmented at a very high rate. Essentially the amount of computation per data item received should not add a lot of latency to each item. Otherwise any such algorithm wont be able to keep up with the data stream. In many cases accessing secondary storage —such as disks— is not even an option. In [ABB+02] there is a discussion of what queries can be answered exactly using bounded memory and queries that must be approximated unless disk access is allowed. Sketching techniques([FM85, AMS99]) have been introduced to build summaries of data in order to estimate the number F_0 of distinct elements in a dataset. In [BYJK+02] three algorithms that (ϵ, δ) approximate the F_0 are described with various space and time requirements. Distinct Sampling[Gib01] is driven by hashing functions similar to those studied in [FM85, AMS99] and provides highly accurate results for distinct value queries compared to those taken by uniform sampling by using only a fraction of their sample size. In [MM02] the algorithms "Sticky Sampling" and "Lossy Counting" are introduced that estimate frequency counts with application to association rules and iceberg cubes. In [GGR03] a framework for performing set expression on continuously updated streams based on sketching techniques is presented. In [ZGTS03] a

general framework over multiple granularities is presented for both range-temporal and spatio-temporal aggregations. In [CKMS03] a framework for identifying "hierarchical heavy hitters", i.e. hierarchical objects (like network addresses whose prefixes defines a hierarchy) with a frequency above a given threshold, is described.

6.8 Summary

We have presented a generalized and parameterized framework that can accurately and efficiently estimate implication counts and can be applied to many scenarios. To the best of our knowledge, this is the first practical and truly scalable approach to the problem of online estimation —within small errors— of complex implication (and non-implication) counts between attributes of a data stream under severe memory and processing constraints and even in the presence of noise. We prove that the complement problem of estimating non-implication counts can be (ϵ, δ) approximated, when the size of the fringe zone is fixed appropriately. We demonstrate that existing algorithms for estimating frequent itemsets or sampling cannot be applied to the problem since they lose the cumulative effect of small implications. In addition, through an extensive set of experiments on both synthetic and real data, we have shown that NIPS/CI always remains very close to the actual implication count, capturing even very small implications whose total contribution is significant.

BIBLIOGRAPHY

[AAD+96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J .F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the 22nd VLDB Conf.*, pages 506–521, 1996.

[ABB+02] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirement for queries over continuous data streams. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2002.

[AES03] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *ACM–SIGMOD*, 2003.

[AGP00] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional Samples for Approximate Answering of Group-By Queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 487–498, Dallas, Texas, 2000.

[AMS99] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58:137–147, 1999.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-*

first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2002.

[BF93] J. Bunge and M. Fitzpatrick. Estimating the number of species: A review. *Journal of American Statistical Association*, 88:364–373, 1993.

[BF95] A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 299–310, 1995.

[Bla98] Jock A. Blackard. The Forest CoverType Dataset. ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype, 1998.

[BPT97a] E. Baralis, S. Paraboschi, and E. Teniente. Materialized View Selection in a Multidimensional Database. In *Proc. of VLDB Conf.*, pages 156–165, Athens, Greece, August 1997.

[BPT97b] E. Baralis, S. Paraboschi, and E. Teniente. Materialized View Selection in a Multidimensional Database. In *Proc. of the 23th International Conference on VLDB*, pages 156–165, Athens, Greece, August 1997.

[BR99] K.S. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, Philadelphia, Pennsylvania, June 1999.

[BYJK+02] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting Distinct Elements in a Data Stream. In *RANDOM*, pages 1–10, 2002.

[CCMN00] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guaranties for distinct values. In *ACM–PODS*, pages 268–279, 2000.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1), September 1997.

[CKMS03] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastana. Finding hierarchical heavy hitters in data streams. In *VLDB*, 2003.

[CMN98] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *ACM–SIGMOD*, pages 436–447, 1998.

[Cou98] Olap Council. APB-1 Benchmark. http://www.olapcouncil.org/research/bmarkco.htm, 1998.

[DANR96] P.M. Deshpande, S. Agarwal, J.F. Naughton, and R. Ramakrishnan. Computation of multidimensional aggregates. Technical Report 1314, University of Wisconsin - Madison, 1996.

[DGR01] A. Deshpande, M. Garofalakis, and R. Rastogi. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In

Proceedings of the annual ACM SIGMOD International Conference in the Management of Data, 2001.

[EKK00] M. Ester, J. Kohlhammer, and H-P. Kriegel. The DC-Tree: A Fully Dynamic Index Structure for Data Warehouses. In *ICDE*, 2000.

[ESV03] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *ACM–SIGCOMM*, 2003.

[FGKP99] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *IJCAI*, pages 1300–1309, 1999.

[FH00] Lixin Fu and Joachim Hammer. CUBIST: A New Algorithm for Improving the Performance of Ad-hoc OLAP Queries. In *DOLAP*, 2000.

[FM85] P. Flajolet and N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, pages 182–209, 1985.

[FSGM+98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J.D. Ullman. Computing Iceberg Queries Efficiently. In *Proc. of the 24th VLDB Conf.*, pages 299–310, August 1998.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proc. of the 12th ICDE*, pages 152–159, New Orleans, February 1996. IEEE.

[GGR03] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing set expressions over continuous update streams. In *ACM SIGMOD*, pages 265–276, 2003.

[GHRU97a] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proc. of ICDE Conf.*, pages 208–219, Burmingham, UK, April 1997.

[GHRU97b] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proceedings of ICDE*, pages 208–219, Burmingham, UK, April 1997.

[Gib01] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.

[GM98] P. B. Gibbons and Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 331–342, Seattle, Washington, June 1998.

[GTK01] L. Getoor, B. Taskar, and D. Koller. Selectivity Estimation using Probabilistic Models. In *ACM–SIGMOD*, 2001.

[Gup97] H. Gupta. Selections of Views to Materialize in a Data Warehouse. In *Proc. of ICDT Conf.*, pages 98–112, Delphi, January 1997.

[HHW97] J.M. Hellerstein, P.J. Haas, and H. Wang. Online Aggregation. In *Proceedings of the ACM SIGMOD Conference*, pages 171–182, Tucson, Arizona, May 1997.

[HNSS95] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, 1995.

[HRU96a] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proc. of ACM SIGMOD*, pages 205–216, Montreal, Canada, June 1996.

[HRU96b] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proc. of ACM SIGMOD*, pages 205–216, Montreal, Canada, June 1996.

[HWL] 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.

[JKR02] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *ACM–WWW*, 2002.

[JLS99] H. V. Jagadish, L. Lakshmanan, and D. Srivastava. What can Hierarchies do for Data Warehouses? In *VLDB*, 1999.

[JS97] T. Johnson and D. Shasha. Some Approaches to Index Design for Cube Forests. *Data Engineering Bulletin*, 20(1):27–35, March 1997.

[Kim96] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, 1996.

[KM95] J. Kivinen and H. Mannila. Approximate dependency inference from relations. *Theoritical Computer Science*, 149:129–149, 1995.

[KM99] H. J. Karloff and M. Mihail. On the Complexity of the View-Selection Problem. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 167–173, Philadelphia, Pennsylvania, May 1999.

[KR98] Y. Kotidis and N. Roussopoulos. An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–258, Seattle, Washington, June 1998.

[LPZ03] L. Lakshmanan, J. Pei, and Yan Zhao. QC-Trees: An Efficient Summary Structure for Semantic OLAP. In *SIGMOD*, 2003.

[MM02] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, August 2002.

[PHIS96] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM–SIGMOD*, pages 294–305, 1996.

[RKR97] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 89–99, Tucson, Arizona, May 1997.

[Rou82] N. Roussopoulos. View Indexing in Relational Databases. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

[RS97] K. A. Ross and D. Srivastana. Fast Computation of Sparse Datacubes. In *Proc. of the 23rd VLDB Conf.*, pages 116–125, Athens, Greece, 1997.

[SAG96] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

[SDN98] A. Shukla, P.M. Deshpande, and J.F. Naughton. Materialized View Selection for Multidimensional Datasets. In *Proceedings of the 24th VLDB Conference*, pages 488–499, New York City, New York, August 1998.

[SDNR96] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presense of hierarchies. In *Proc. of VLDB*, pages 522–531, Bombay, India, August 1996.

[SDRK02] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the Petacube. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, June 2002.

[SFB99] J. Shanmugasundaram, U. Fayyad, and P.S. Bradley. Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions. In *Proc. of the Intl. Conf. on Knowledge Discovery and Data Mining (KDD99)*, 1999.

[SLC+01] S. Stolfo, W. Lee, P. Chan, Wei Fan, and E. Eskin. Data mining-based intrusion detectors: An overview of the columbia ids project. *ACM–SIGMOD Record*, 30(4):5–14, 2001.

[TS97] D. Theodoratos and T. Sellis. Data Warehouse Configuration. In *Proc. of the 23th International Conference on VLDB*, pages 126–135, Athens, Greece, August 1997.

[VS99] P. Vassiliadis and T. Sellis. A Survey of Logical Models for OLAP Databases. *SIGMOD Record*, 28(4), 1999.

[VWI98] J.S Vitter, M. Wang, and B. Iyer. Data Cube Approximation and Histograms via Wavelets. In *Proc. of the 7th Intl. Conf. Information and Knowledge Management (CIKM'98)*, 1998.

[WLFY02] Wei Wang, Hongjun Lu, Jianlin Feng, and Jeffrey Xu Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. In *ICDE*, 2002.

[WVZT90] K. Whang, B. Vander-Zander, and H. Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems*, pages 209–229, 1990.

[WZS02] Haining Wang, Danlu Zhang, and Kang G. Shin. Detecting SYN Flooding Attacks. In *INFOCOM 2002*, 2002.

[ZDN97] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. of the ACM SIGMOD Conf.*, pages 159–170, 1997.

[ZGTS03] D. Zhang, D. Gunopulos, V. Tsotras, and B. Seeger. Temporal and spatio-temporal aggregations over data streams using multiple time granularities. *Information Systems*, 28(1-2):61–84, 2003.