Computing and Querying Datacubes

Kazi Atif-Uz Zaman

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2001

Abstract

Computing and Querying Datacubes

Kazi Atif-Uz Zaman

Datacube queries compute aggregates over database relations at a variety of granularities, and they constitute an important class of decision support queries. In this thesis we study problems pertaining to the computation of datacubes and frameworks for querying them.

Often one wants only datacube output tuples whose aggregate value satisfies a certain condition, such as exceeding a given threshold. For example, one might ask for all combinations of model, color, and year of cars (including the special value "ALL" for each of the dimensions) for which the total sales exceeded a given amount of money.

Computing a selection over a datacube can naively be done by computing the entire datacube and checking if the selection condition holds for each tuple in the result. However, it is often the case that selections are relatively restrictive, meaning that a lot of work computing datacube tuples is "wasted" since those tuples don't satisfy the selection condition.

Our approach is to develop algorithms for processing a datacube query using the selection condition internally during the computation. By making use of the selection condition within the datacube computation, we can safely prune parts of

the computation and end up with a more efficient computation of the answer. Our first technique, called “specialization”, uses the fact that a tuple in the datacube does not meet the given threshold to infer that all finer level aggregates cannot meet the threshold. We propose a scheme of specialization transformations on the underlying data sets, using properties of the aggregates and threshold functions.

Our second technique is called “generalization”, and applies in the case where the actual value of the aggregate is not needed in the output, but used just to compare with the threshold. We refer to these as “projected datacube” queries. Generalization uses the fact that a tuple meets the given threshold to infer that all coarser level aggregates also meet the threshold. We also propose a scheme of generalization transformations. We demonstrate that computing the median is easier for projected datacubes.

We demonstrate the efficiency of these techniques by implementing them within the sparse datacube algorithm of Ross and Srivastava. We present a performance study using synthetic and real-world data sets. Our results indicate substantial performance improvements for queries with selective conditions.

In the second major piece of work we study a main memory based framework for querying datacubes. For large datasets with many dimensions, the complete datacube may be very large. In order to support on-line access to datacube results, one would like to perform some precomputation to enhance query performance.

Existing schemes materialize selected datacube tuples on disk, choosing the most beneficial cuboids (i.e., combinations of dimensions) to materialize given a space limit. However, in the context of a data-warehouse receiving frequent “append” updates to the database, the cost of keeping these disk-resident cuboids up-to-date can be high.

We propose a main memory based framework which provides rapid response to queries and requires considerably less maintenance cost than a disk based scheme in an append-only environment. We materialize in main memory (a) selected coarse-granularity tuples of the datacube, and (b) all tuples at the finest level of granularity of the cube. Our approach is limited to applications in which the finest granularity tuples of the datacube fit in main memory. We argue that there are important applications that meet this requirement. Further, as main memory sizes grow over the coming years, more and more applications will meet this requirement.

For a given datacube query, we first look among our coarse-granularity tuples to see if we have a direct answer for the query. If so, that answer is returned directly. If not, we use a hash based scheme reminiscent of partial match retrieval to rapidly compute the answer to the query from the finest-level data without having to scan all of the base tuples. Our in-memory data structures allow for rapid updates in response to the appearance of new base data.

We show how to choose which coarse-level tuples to precompute, and how to select partial-match keys to minimize the effects of skew. We present analytical and experimental results demonstrating the benefits of our approach.

Contents

List of Figures

Acknowledgments

I would like to begin by thanking my advisor, Ken Ross. I learned an enormous amount from Ken, ranging from how to attack a research problem to the ins and outs of a well written technical paper. Ken's razor sharp insights never ceased to amaze me and our weekly meetings always made me approach research with renewed vigor. I would like to thank Al Aho for inviting me to study at Columbia and Sriram Padmanabhan for giving me the opportunity to spend a summer at IBM research. Thanks also go to Gail Kaiser and Divesh Srivastava who along with the above mentioned served on my thesis committee.

My brother and father were always there to provide me with support even though they were thousands of miles away. My mother isn't here to see the end of this long journey and this is my one and only regret. She always encouraged me to pursue my dreams and steadfastly believed in me. This thesis is dedicated to her.

To my Mother

Chapter 1

Introduction

1.1 Introduction

In this thesis we develop algorithms and frameworks for computing and querying datacubes. In this chapter, we first describe query processing in database systems. We then introduce Decision Support Systems and some typical complex queries which arise in these systems, focusing specifically on datacube queries involving the **CUBE** operator (defined in Section 1.5 below). We then briefly describe the major contributions of this thesis. The first of these is developing algorithms for efficiently optimizing selections for queries involving the CUBE operator. The second is a framework for efficiently organizing data in main memory to facilitate efficient query processing of CUBE queries. We conclude with a description of the organization of this thesis.

1.2 Query Processing in Databases

Queries in a relational database system are typically expressed in a high level query language like SQL. The data is stored on disk in low level structures like files and indices. Query processing bridges the gap between the two. Query processing can be divided into two stages: **Query Optimization** and **Query Evaluation**. The goal of query optimization is to translate a query expressed in a high level query language into an efficient sequence of operations (*query plan*) which can be executed by the query execution engine of the database system. Efficiency is measured in terms of a relevant performance measure which typically could be the the users wait time for the first or last result item, CPU, I/O and network costs, or any combination of the above. Query evaluation operators are the basic operations on the low level structures. Examples of these operators are external sorts, sequential scans, index scans, nested-loop joins and sort-merge joins. There does not necessarily exist a one-to-one correspondence between these physical operators and the operators used in the high level language. An abstract representation of a query plan is a *physical operator tree*. There are many possible query plans corresponding to a high level query. Query optimization can be viewed as a search problem over the space of all possible plans. In order to solve this problem the following are required:

- A space of plans (*search space*)
- A *cost estimation* technique so that a cost may be assigned to each plan in the search space.
- An *enumeration algorithm* that can search through the search space.

A desirable optimizer is one where

1. The search space includes plans with low cost.

2. The costing technique is accurate.

3. The enumeration algorithm is efficient.

Query optimization is described in detail in [Cha98] and query evaluation techniques are described in [Gra93]. Datacube queries belong to the class of *aggregation* queries. We will briefly describe the traditional query evaluation techniques for aggregate queries in SQL which make use of the `GROUP BY` clause in the next chapter.

1.3 Analytical Queries and Data Warehouses

Many organizations have large amounts of data which can be harnessed to answer queries that provide useful insights. For example, a manager of a company might be interested in knowing the proportion of late deliveries for each business unit or the number of business units having a net loss despite having an advertising budget of more than a million dollars. A decision support system aims at enabling the user to efficiently answer queries of this nature. While decision support systems are typically referred to in the context of large business organizations, the complex queries characteristic of such systems are also useful for medical and scientific databases. These queries often involve aggregations, joins, selections, views or subqueries. Many of these queries are often automatically generated by front end tools.

While relational database systems have been available commercially for more than twenty years, the focus has been on developing systems with robust transaction-processing capabilities with high throughput. The needs of such a system are different from our scenario described earlier where we have mainly read-only operations with infrequent updates. Typically, operational systems are kept separate from the decision support system which has its data stored in a *Data Warehouse*. Data warehouses tend to be orders of magnitude larger than operational databases. Query throughput and response times are more important than transaction throughput.

Data in a warehouse is typically modeled multidimensionally. In a warehouse containing data about cars, some of the typical dimensions could be *manufacturer, model, color* and *sales district*. These are referred to the *dimensional* attributes. In addition to the dimensional attributes we will also have *measure* attributes. In our example a measure attribute could be the total sales for a particular manufacturer, model, day, color and district. Dimensional attributes are often *hierarchical*, the time of sale dimension may be organized as day-month-quarter-year hierarchy.

1.4 Warehousing Architectures: ROLAP vs MOLAP

The data stored in a data warehouse needs to be efficiently accessed by users. While most relational database systems vendors have moved to support warehousing requirements, specialized solutions have been developed which consist of an OLAP (Online Analytical Processing) server on top of the warehouse which is accessed by front end analysis tools. There are two types of OLAP servers, Relational OLAP servers (ROLAP) and Multidimensional servers (MOLAP). Relational OLAP servers extend traditional relational database servers with specialized middleware to efficiently support complex queries. This middleware layer optimizes for specific back-end relational servers by identifying views which are to be materialized, rephrasing queries in terms of these materialized views and generating the appropriate SQL. The main advantage of this approach is that ROLAP servers exploit the strengths of traditional relational databases, namely scalability and transactional support. The ROLAP approach has been championed by Microstrategy.[1]

In contrast, MOLAP servers have a special multidimensional storage engine with an array based data structure for storing the multidimensional data. This data structure stores only the measure attributes; the dimensional attributes are implied by the dimensions of the array. Multidimensional queries are answered by carrying out the appropriate mapping to the data structure and aggregating if necessary. While this approach leads to enhanced query performance, its space utilization is poor particularly for large sparse data sets. Another drawback is that while ROLAP queries can take advantage of the expressive power of SQL, MOLAP servers are limited to features provided by the vendor. The MOLAP approach was pioneered by Arborsoft with its Essbase product.

1.5 The CUBE Operator

The complex queries submitted to decision support systems necessitated the need for many new operators. One of the most popular of these is the CUBE operator, which was first introduced by Gray et. al in [GBLP96]. We illustrate the CUBE operator by means of an example.

[1] Now best known for its stock price collapse due to questionable accounting practices.

Example 1.5.1: Consider the following table containing data about the sales of cars by model, year and color.

Model	Year	Color	Sales
Chevy	1994	Red	90
Chevy	1994	Black	50
Ford	1994	Black	70
Chevy	1995	Red	60
Chevy	1995	Black	65
Ford	1995	Black	95

This table has 3 dimensions *Model* , *Year* and *Color* in addition to the attribute *Sales* that we are aggregating. We might be interested in the sales of cars by model and year for all colors, the sales by year and color for all models and so on. Since we have 3 dimensions we have $2^3 = 8$ different granularities at which we can compute the sum of sales. So that we can represent all these tuples consistently, we replace the dimensions being aggregated over with the special value ALL. The resultant tuples are shown below, the tuples from each distinct granularity are shown separately. We will subsequently refer to each of these granularities as a *cuboid*. In general, a d dimensional relation has 2^d cuboids.

Model	Year	Color	Sales
Chevy	1994	Red	90
Chevy	1994	Black	50
Ford	1994	Black	70
Chevy	1995	Red	60
Chevy	1995	Black	65
Ford	1995	Black	95

Chevy	1994	ALL	140
Ford	1994	ALL	70
Chevy	1995	ALL	125
Ford	1995	ALL	95

ALL	1994	Red	90
ALL	1994	Black	120
ALL	1995	Red	60
ALL	1995	Black	160

Chevy	ALL	Red	150
Chevy	ALL	Black	115
Ford	ALL	Black	165

ALL	1994	ALL	210
ALL	1995	ALL	220

ALL	ALL	Red	150
ALL	ALL	Black	280

Chevy	ALL	ALL	265
Ford	ALL	ALL	165

ALL	ALL	ALL	430

We notice that while each cuboid can be directly computed from the base data, some cuboids can be directly computed from other cuboids. [1] The "sum of sales of all black cars" can be computed from the "sum of sales of all black Chevys" and "the sum of sales of all black Fords" rather than having to go to the base data. These relationships are captured in Figure 1.1 which shows the different granularities at which aggregations can be expressed. An edge is present between two granularities when the first granularity can be used to compute the second and the second granularity is specified by one less attribute than the first. All edges are from finer granularities to coarser granularities. □

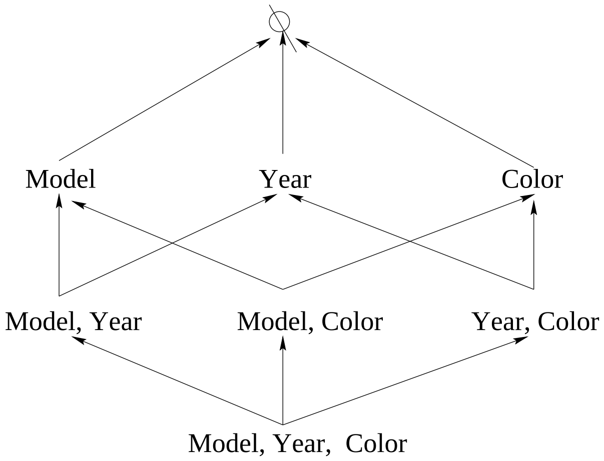


Figure 1.1: The Datacube Lattice

[1] Assuming certain properties of aggregate functions. We discuss different classes of aggregate functions in the next chapter.

There are a number of different algorithms for computing the datacube [AAD+96, GBLP96, RS97b, ZDN97, SLCJ98]. We notice that the size of a datacube is often much larger than that of the base data itself. Datacube computation is very expensive and if the cube is going to be frequently queried it is inefficient to compute the necessary datacube tuples from the base data for each query. In this scenario, performance can be greatly improved by precomputing parts of, or the whole of, the datacube. There are issues of which datacube tuples to precompute, which depend upon space constraints in memory and disk. In this thesis, we will be dealing with both the aspects of this problem.

1.5.1 Vendor Implementations

A number of OLAP specific language proposals have been incorporated into a SQL draft amendment. A non final version of this draft amendment can be found online [SQL]. The syntax of our example CUBE query given earlier is shown in Figure 1.2. The syntax of a CUBE query can also include `WHERE` and `HAVING` clauses.

Most major relational database servers now support the CUBE operator, this list includes Oracle's Oracle-8i, IBM's DB2 and Microsoft's SQL server. One point of difference is the treatment of ALL values in the result of a query. Some vendors like Oracle and IBM use a $NULL$ value instead of ALL. This $NULL$ is indistinguishable from stored NULL values caused by missing attribute values. The vendors allow users to distinguish between these $NULL$'s by adding specialized constructs which add extra columns to the output which indicate which cuboid of the datacube a particular row is from. Oracle has the `GROUPING` function while IBM introduces the `GROUPING SETS` function. Since these constructs are vendor specific we will not discuss them further. In our treatment of datacubes, we will use the *ALL* notation for ease of exposition.

```
SELECT    model, year, color, SUM(sales)
FROM      cars
GROUP BY  CUBE model, year, color
```

Figure 1.2: SQL syntax for example CUBE query

1.6 Optimizing Selections over Datacubes

Often one wants only datacube output tuples whose aggregate value satisfies a certain condition, such as exceeding a given threshold. For example, one might ask for all combinations of model, color, and year of cars (including the special value “ALL” for each of the dimensions) for which the total sales exceeded a given amount of money. Syntactically, these queries are CUBE queries which include a `HAVING` clause as illustrated in Figure 1.3.

```
SELECT    model, year, color, SUM(sales)
FROM      cars
GROUP BY  CUBE model, year, color
HAVING    SUM(sales) > 10000
```

Figure 1.3: SQL syntax for example selection over datacube

Computing a selection over a datacube can be done naively by computing the entire datacube and checking if the selection condition holds for each tuple in the result. However, it is often the case that the selection condition holds for relatively few tuples, meaning that a lot of work computing datacube tuples is “wasted” since those tuples don't satisfy the selection condition. Since datacubes tend to be large in size this effect is particularly pronounced for selective predicates, making efficient evaluation of such queries a practical and important problem.

Our approach is to develop algorithms for processing a datacube query using the selection condition internally during the computation. By making use of the selection condition within the datacube computation, we can safely prune parts of the computation and end up with a more efficient computation of the answer. Our first technique, called "specialization", uses the fact that a tuple in the datacube does not meet the given threshold to infer that all finer level aggregates cannot meet the threshold. We propose a scheme of specialization transformations on the underlying data sets, using properties of the aggregates and threshold functions.

Our second technique is called "generalization", and applies in the case where the actual value of the aggregate is not needed in the output, but used just to compare with the threshold. Generalization uses the fact that a tuple meets the given threshold to infer that all coarser level aggregates also meet the threshold. We also propose a scheme of generalization transformations.

We demonstrate the efficiency of these techniques by implementing them within the sparse datacube algorithm of Ross and Srivastava described in detail in the following chapter. We present a performance study using synthetic and real-world data sets. Our results indicate substantial performance improvements for queries with selective conditions.

These techniques are described in detail in Chapter 3 of this thesis. This work has been published as [RZ00a].

1.7 Serving Datacube Tuples from Main Memory

For large datasets with many dimensions, the complete datacube may be very large. In order to support on-line access to datacube results, one would like to perform some precomputation to enhance query performance.

Existing schemes materialize selected datacube tuples on disk, choosing the most beneficial cuboids (i.e., combinations of dimensions) to materialize given a space limit. However, in the context of a data-warehouse receiving frequent "append" updates to the database, the cost of keeping these disk-resident cuboids up-to-date can be high.

We propose a main memory based framework which provides rapid response to queries and requires considerably less maintenance cost than a disk based scheme in an append-only environment. We materialize in main memory (a) selected coarse-granularity tuples of the datacube, and (b) all tuples at the finest level of granularity of the cube. Our approach is limited to applications in which the finest granularity tuples of the datacube fit in main memory. We argue that there are important applications that meet this requirement. Further, as main memory sizes grow over the coming years, more and more applications will meet this requirement.

For a given datacube query, we first look among our coarse-granularity tuples to see if we have a direct answer for the query. If so, that answer is returned directly. If not, we use a hash based scheme reminiscent of partial match retrieval to rapidly compute the answer to the query from the finest-level data without having to scan all of the base tuples. Our in-memory data structures allow for rapid updates in response to the appearance of new base data.

We show how to choose which coarse-level tuples to precompute, and how to select partial-match keys to minimize the effects of skew. In Chapter 4 of this thesis we present analytical and experimental results demonstrating the benefits of our approach. This work has been published as [RZ00b].

1.8 Organization

This thesis is organized as follows. In Chapter 2 we introduce the terminology and notation typically associated with datacubes. We describe algorithms for computing the datacube and other research problems connected with datacube computation. Chapter 3 describes our algorithms for optimizing selections over datacubes along with experimental results. The details of our framework for serving datacube tuples from main memory are in Chapter 4. We then present the conclusions of this thesis.

Chapter 2

Datacubes: Definitions and Computations

In this chapter we first introduce notation and terminology used to describe datacubes. We then describe various algorithms developed for the computation of datacubes. We conclude with a discussion of various research problems which have arisen in the context of datacubes.

2.1 Notation and Terminology

The relationship between cuboids can be captured in terms of the *search lattice* of the datacube. This is illustrated in Figure 2.1.

Consider a relation with attributes $B_1, \ldots, B_k$ over which we are computing a datacube. We use the notation $Q(\vec{B}_i)$ to denote the cuboid at granularity $\vec{B}_i$. Each granularity $\vec{B}_i \subseteq \{B_1, \ldots, B_k\}$ is a node in the search lattice, and there is an edge from node $\vec{B}_i$ to $\vec{B}_j$ if $\vec{B}_j$ is a subset of and has one fewer element than $\vec{B}_i$; $\vec{B}_i$

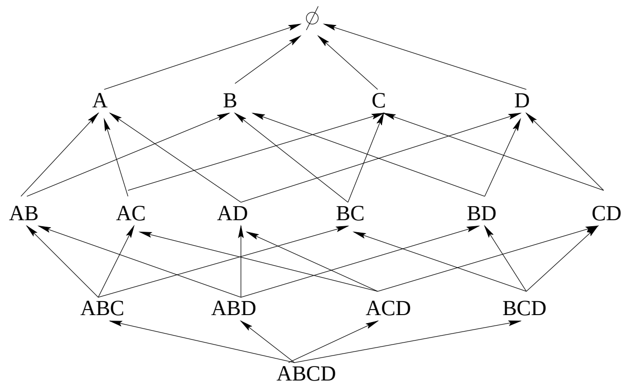


Figure 2.1: Lattice representation of a Datacube

is said to be a *parent* of $\vec{B_j}$ in the search lattice. If there is a path from $\vec{B_i}$ to $\vec{B_j}$ in the search lattice, $\vec{B_i}$ is said to be of a finer granularity than $\vec{B_j}$, and $\vec{B_j}$ is said to be of a coarser granularity than $\vec{B_i}$. Paths in the search lattice precisely determine which of the cuboids can be computed from which others. In particular, cuboid $Q(\vec{B_j})$ can be computed using $Q(\vec{B_i})$ if and only if $\vec{B_j}$ is of coarser granularity than $\vec{B_i}$. The computation of the various cuboids are not independent of each other, but are closely related in that some of them can be computed using others.

We define a cuboid consisting of tuples with exactly n non-ALL dimension attribute values to be a n-cuboid.

2.1.1 Aggregation Functions

The aggregation function greatly influences the possible optimizations while computing the aggregate. Optimizations used for functions like `SUM` cannot be used for functions like `MEDIAN`. It is convenient to divide aggregation functions into categories with similar properties. The following categorizations of aggregate functions was introduced in [GBLP96]. Let $\cup$ denote multiset union, and let S, S' be multisets.

Definition 2.1.1: An aggregate function g is *distributive* if there exists a binary function h such that for every nonempty S and S', $g(S \cup S') = h(g(S), g(S'))$. Examples of such aggregate functions include SUM, COUNT, MIN and MAX. For all the above functions except COUNT, $g = h$. For COUNT, $h = SUM()$.

Definition 2.1.2: An aggregate function $g()$ is said to be *algebraic* if there is a M-tuple valued function $h()$ and a binary function $i()$ taking M-tuples as input such that for every nonempty S and S', $g(S \cup S') = i(h(S), h(S'))$. AVG, standard deviation and center_of_mass() are all algebraic. For AVG(), the function $h()$ records the sum and count of the subset. The $i()$ function adds these two components and then divides to produce the global average. The key point is that a fixed size result (in this case a 2-tuple) can summarize the sub-aggregation.

Definition 2.1.3: An aggregate function $g()$ is said to be *holistic* if there is no constant bound on the size of storage needed to describe a sub-aggregate. There is no constant M such that a M-tuple produced by invoking function $h()$ over S and S' which can be combined in a function $i()$ to produce $g()$. Median, mode and rank are examples of holistic functions.

Datacubes of distributive and algebraic aggregation functions are easier to compute than that of holistic functions since aggregates computed for one granularity can be reused while computing aggregates of a coarser granularity. Most work in the research community has focussed on distributive aggregation functions.

Apart from the distributive aggregate functions in SQL (MIN, MAX, SUM, COUNT), we also consider $\bigcup$ and $\bigcap$ as aggregate functions. If we use a bitmap representation for a set, then $\bigcap$ corresponds to the AND operation on bits and $\bigcup$ to the OR operation. Both of these operations are clearly distributive since we can pick $h = \bigcap$ and $h = \bigcup$ respectively.

A property that is important when considering potential optimizations of aggregate computation is the idempotence.

Definition 2.1.4: Let S range over multisets over a domain D, and let the aggregate function g be a mapping from multisets over D to D. We say g is *idempotent* if for every nonempty multiset S, $g(\{g(S)\}) = g(S)$. $\square$

2.1.2 Sparse vs Dense Cubes

A relation is *sparse* with respect to a set of attributes if its cardinality is a small fraction of the size of the cross-product of the attribute domains. Correspondingly, a relation is dense if its cardinality is large compared to the size of the cross-product of the attribute domains. Sparseness exists for two distinct reasons: large domain sizes of some `CUBE BY` attributes and a large number of `CUBE BY` attributes in the datacube query. Real-World data in application domains is often very large and sparse. Hence, efficiently computing datacubes over large sparse relations is important.

2.2 Datacube Computation Algorithms

We first examine how a single cuboid (a `GROUP BY`) is evaluated in a traditional relational database system. We then describe the most popular cube computation algorithms.

2.2.1 Evaluating a GROUP BY

An example of a SQL query with a `GROUP BY` is given in Figure 2.2

```
SELECT    model, year, color, SUM(sales)
FROM      cars
GROUP BY  model, year, color
```

Figure 2.2: Example GROUP BY query

Algorithms for aggregate functions require grouping; the car records for each combination of model, year and color are grouped together and then one output item is computed per group. This grouping process is similar to duplicate removal where equal data items must also be brought together. Grouping is achieved by one of two major methods: sorting and hashing. Both techniques are not substantially different in performance. The dominant cost in computing a `GROUP BY` is the I/O cost. The performance cost of evaluating a `GROUP BY` is given in terms of the following parameters of the database system. The I/O unit C indicates the number of disk pages which are treated as a single unit. The maximal fan-in F is a useful variable in the analysis of execution algorithms requiring external sorting or partitioning.

Size of relation	R
Memory Size	M
I/O unit (number of pages)	C
Maximal Fan In F	$\lfloor M/C - 1 \rfloor$
Size of final output	O

Hash Based Grouping

The algorithm for computing a `GROUP BY` is to successively partition the base data until we have a dataset which fits in memory. If the base data is substantially larger than memory and $O > M$, the dominant cost is the I/O incurred partitioning the data. The number of partitioning levels before output of a partition fits in memory is given by $\lceil log_F(O/M) \rceil$. The total I/O costs incurred are $2 * R * \lceil log_F(O/M) \rceil$, the factor of two for reading and writing.

Sort Based

One can external sort the data according to a prespecified order of the `GROUP BY` attributes. Equal items will be together and the desired aggregate can be computed with a scan of the sorted data. The dominant cost is that of external sorting. The number of merge levels is $L = \lceil log_F(R/M) \rceil$. If the output size O is G times smaller than R, only the last $\lceil log_F(G) \rceil$ merge levels including the final merge are affected by early aggregation. The number of affected levels, $L_2 = \lceil log_F(G) \rceil - 1$ since the final merge creates an output stream. In the affected levels, the size of the output runs is constant while the number of runs in each unaffected level is given by $R/(M * F^i)$ for level i. Let the number of unaffected levels be $L_1 = L - L_2$. The total sorting cost is given by $2 * R * L_1 + 2 * O * \sum_{i=L_1}^{L-1} R/(M * F^i)$

The important point to note is that performance of both sort based and hash based techniques is logarithmic and improves with increased reduction factors. The sort order chosen for sort based aggregation has an analogue in the choice of partitioning attributes in hash based aggregation.

2.2.2 Array Based Datacube Algorithms

The array-based algorithm proposed by Gray et al. [GBLP96] is essentially a main memory algorithm, where all the space for the tuples of the finest level of the datacube are kept in memory as a k dimensional array, where k is the number of `CUBE BY` attributes. A single pass is made over the base data, each base tuple is examined and the appropriate result tuple is incremented appropriately. The remaining cuboids of the datacube are computed by aggregating over the finer granularity cuboids. When a cuboid can be computed over one or more possible parent cuboids, it is proposed to use the *smallest parent* optimization. This optimization proposes using the parent cuboid which is smallest in size. The data structure needed for this algorithm will often not fit into memory for sparse relations even when R does. In this case, the algorithm does not apply.

Zhou et al. [ZDN97] have proposed an array based algorithm that computes the datacube using array-chunking techniques. By managing the order in which chunks are processed, substantially less of the result array needs to be kept in memory at any one time than with the algorithm of Gray et al. Nevertheless, when the data is very sparse, this algorithm too may fail since it cannot allocate enough main memory to hold the needed parts of the result array. Array based algorithms do not scale well with an increase in the number of dimensions.

2.2.3 PIPESORT

The PIPESORT algorithm [AAD+96] tries to optimize the overall computation of a datacube by providing a set of paths which cover the search lattice and then executing each path in turn. This algorithm makes use of cost estimates of the various ways to compute each cuboid $Q(\vec{B}_j)$ to determine which cuboid will actually be used to compute the tuples of $Q(\vec{B}_j)$.

PIPESORT annotates each edge $(\vec{B}_i, \vec{B}_j)$ of the search lattice of the datacube Q with two costs: $S(\vec{B}_i, \vec{B}_j)$ is the cost of computing $Q(\vec{B}_j)$ from an unsorted $Q(\vec{B}_i)$ and $A(\vec{B}_i, \vec{B}_j)$ is the cost of computing $Q(\vec{B}_j)$ from a sorted $Q(\vec{B}_i)$.

Given the annotated search lattice, PIPESORT adopts a level-by-level approach to match cuboids with their parents. Consider the empty cuboid (the super aggregate with just a single tuple) to be level 0. PIPESORT seeks to match each cuboid in level k with a parent in level $k+1$ such that the overall sum of the edge costs between levels is minimized. This is equivalent to finding a minimum cost matching in a bipartite graph. A point to be noted is that not more than one edge of type A can be picked per node. This is because edges of type A correspond to a sort order and node can only have one sort order at a time. This level-by-level matching is carried out starting with the root (cuboid with k dimensions in a k dimensional datacube) and ending with level 0. PIPESORT adds a node corresponding to the relation R and adds an edge marked S from R to the root of the tree.

After this matching has been carried out for all the levels, the chosen edges form a tree. A graph will not be formed since there is a matching at each level and there is only one incoming edge into a node. This tree can be broken up into a set of paths such that every edge in the tree is present in one and only one path and all the edges in each path except the first edge are marked A.

At evaluation time, these paths are executed in sequence. Each path with root $\vec{B}_i$ (or R) will have the tuples of the root sorted according to an order determined by the attributes of the children. For example, if the cuboid at the lowest level has two attributes, these two attributes will be a prefix of the sort order of the path. This will hold for all non-root cuboids on the path allowing us to determine the sort order. Once the tuples are sorted, the pipelined evaluation of the path requires only a single tuple to be maintained per non-root node which is very space efficient.

The main drawback of PIPESORT is that it does not scale well with an increase in dimensionality of the datacube. For a datacube of dimensionality k, PIPESORT will carry out at least $\binom{k}{\lceil k/2 \rceil}$ sorts. Sorting these cuboids requires external sorts which requires a large amount of I/O.

2.2.4 OVERLAP

The OVERLAP algorithm proposed by Deshpande et al [AAD$^+$96] tries to minimize the number of disk accesses by overlapping the computation of the cuboids, by making use of the partially matching sort orders to reduce the number of sorting steps performed.

OVERLAP first computes the finest granularity cuboid $Q(\{B_1, \ldots, B_k\})$ from R and sorts the tuples of this cuboid in a chosen sort order. The attributes at each node of the search lattice are ordered such that they have a maximal length common prefix with the sort order of the finest granularity cuboid. The parent of each cuboid is chosen to be the smallest cuboid with the maximum shared length prefix. Thus, OVERLAP computes a cuboid tree which has the finest granularity cuboid as the root and covers the entire search lattice. Each cuboid is computed from the tuples of its parent cuboid in this tree.

Next a set of cuboids is chosen that can be computed concurrently within the available memory constraints. Nodes of the tree are labelled with estimates of the memory required to compute the corresponding cuboid from its parent, assuming the tuples of the parent cuboid are sorted in the order indicated by the attribute ordering of the parent node. The estimates are computed using the uniform distribution assumption. If memory can be allocated the cuboid is marked in the *Partition* state. At evaluation time, the cuboids in the *Partition* state are immediately available for pipelining purposes. Other cuboids are allocated a single page of memory and are said to be in the *SortRun* state. The allocated page can be used to write out sorted runs for the cuboid on disk. These sorted runs are merged in further passes to complete the computation. The allocation of memory, which determines whether a cuboid is in the *Partition* or *SortRun* state, is carried out by using the heuristic of traversing the tree in breadth first order and giving priority to larger cuboids. For sparse relations the I/O cost of OVERLAP is at least quadratic in the dimensionality of the cube, k. This is illustrated in [RS97b].

2.2.5 Memory-Cube and Partitioned-Cube

`Partitioned-Cube` and `Memory-Cube` [RS97b] are efficient algorithms for computing datacubes which work particularly well for sparse data. These algorithms are based on two fundamental ideas that have been successfully used for performing complex operations (such as sorting and joins) over large relations.

- Partition the large relations into fragments that fit into memory
- Perform the complex operation over each memory-sized fragment independently.

`Partitioned-Cube` (Figure 2.3) is an algorithm which uses a divide-and-conquer strategy to divide the problem of computing the datacube over a relation with T tuples and k `CUBE BY` attributes into $n+1$ sub-datacubes for a large number n. The first n of these sub-datacubes each has approximately T/n tuples and k `CUBE BY` attributes. The final sub-datacube has no more than T tuples and has $k-1$ `CUBE BY` attributes. Algorithm `Partitioned-Cube` assumes the existence of a subroutine `Memory-Cube` (Figure 2.4) which efficiently computes datacubes for relations that fit in memory.

The structure of `Partitioned-Cube` follows the recursive structure of datacubes themselves. A datacube is obtained by fixing each possible value of a `CUBE BY` attribute B_j in turn and computing the tuples in the corresponding sub-datacube, followed by computing the datacube tuples with the value ALL for B_j. Rather than rereading the input relation R for the ALL datacube we read the finest granularity cuboid F, which may be significantly smaller than R if there are many tuples in each group, and is never larger than R. These observations form the basis of an inductive proof of the correctness of algorithm `Partitioned-Cube`. The I/O cost is typically proportional to the number of `CUBE BY` attributes k, and not exponential in k as in PIPESORT or quadratic in k as OVERLAP.

Algorithm `Partitioned-Cube` is adaptive to skew in the sense that partitions that fit in memory will be handled by Algorithm `Memory Cube`, while larger partitions will be repartitioned.

`Memory Cube` computes the various cuboids of the datacube using the idea of pipelined paths where each path requires the relation to be sorted in a particular attribute ordering. The paths are generated by an algorithm which generates

Algorithm `Partitioned-Cube`$(R, \{B_1, \ldots, B_m\}, A, G)$

INPUTS:
A set of tuples R, possibly stored in horizontal fragments;
`CUBE BY` attributes $\{B_1, \ldots, B_m\}$;
Attribute A to be aggregated,
Aggregate function $G()$.

OUTPUTS:
The datacube result for R over $\{B_1, \ldots, B_m\}$ in two horizontal fragments F and D on disk.
F contains the finest granularity tuples and D contains the remaining tuples. (F and D may themselves be further horizontally partitioned.)

METHOD:

```
If (R fits in memory) return Memory-Cube(R, {B_1, ..., B_m}, A, G);
else {
        Choose an attribute B_j among {B_1, ..., B_m};
        Scan R and partition on B_j into sets of tuples R_1, ..., R_n;
        /* n ≤ card(B_j) and n ≤ number of buffers in memory */
        for i = 1 ... n
            let (F_i, D_i) = Partitioned-Cube(R_i, {B_1, ..., B_m}, A, G)
        let F = the union of the F_i's
        let (F', D') = Partitioned-Cube(F, {B_1, ..., B_{j-1}, B_{j+1} ..., B_m}, A, G)
        let D = the union of F', D' and the D_i's
        return (F, D) ;
}
```

Figure 2.3: Algorithm `Partitioned-Cube`

$\binom{k}{\lceil k/2 \rceil}$ paths for a datacube with k `CUBE BY` attributes. These paths are processed in an order which allows us to share as much computation as possible across paths. `Memory-Cube` does not incur any I/O beyond the input of the relation and the output of the datacube itself.

2.2.6 BUC

Bottom-Up Cube(BUC) [BR99] is a recent algorithm developed independently by researchers at the University of Wisconsin. It is similar to a version of `Partitioned-Cube` that never calls `Memory-Cube`. Instead the BUC algorithm, shown in Figure 2.5 builds the cuboid with a single attribute, followed by a cuboid with two attributes and so on. The intuition is that even though work is duplicated while computing these cuboids, this additional work is more than offset by the computation which can be pruned when a tuple does not meet minimum support. The `HAVING` clause considered is of the form `HAVING COUNT(*) >= X` which is equivalent to computing tuples having minimum support in an association rules context [AS94].

Since BUC seeks to trade partitioning costs against that of aggregation, it may not be a suitable algorithm if the aggregation operation is expensive.

2.3 Related Datacube Problems studied

We now examine several other research problems studied in the context of datacubes.

INPUTS:
A set of tuples R, that fits in memory; `CUBE BY` attributes $\{B_1, \ldots, B_m\}$; attribute A to be aggregated, aggregate function $G()$.

OUTPUTS:
The datacube result for R over $\{B_1, \ldots, B_m\}$ in two horizontal fragments F and D on disk. F contains the finest granularity tuples and D contains the remaining tuples.

METHOD:

```
Sort R and combine all tuples that share all the values of {B_1, ..., B_m};
/* Assume that tuples are sorted according to first sort order */
for each sort order {
        initialize accumulators for computing aggregates at each granularity
        combine first tuple into finest granularity accumulator
        for each subsequent tuple t {
            Compare t with previous tuple, to find position j of the first sort
            order attribute at which they differ
            if (j exceeds number of common attributes with next sort order) then {
                resort segment from t'(last tuple where condition is satisfied) to
                tuple prior to t according to next sort order.
            }
            if (grouping attributes of t differ from finest granularity accumulator) {
                output and combine each accumulator into coarser granularity
                accumulator until grouping attributes match those of t;
                /* Number of combinings depends on sort order length and j */
            }
            Combine current tuple with finest granularity accumulator;
        }
}
```

Figure 2.4: Algorithm `Memory-Cube`

INPUTS:
input: The relation to aggregate
dim: The starting dimension for this iteration

GLOBALS:
constant numDims: The total number of dimensions
constant cardinality[numDims]: Cardinality of each dimension
constant minsup: Minimum number of tuples in a partition for it to be output
outputRec: The current output record
dataCount[numDims]: Stores the size of each partition.
dataCount[i] is a list of integers of size cardinality[i]

OUTPUTS:
One record that is the aggregation of input
Recursively, outputs CUBE(dim,...,numDims) on input (with minimum support)

METHOD:
```
Aggregate(input); // Place result in outputRec
If ( input.count() == 1) then
        WriteAncestors(input[0],dim); return;
write outputRec;
for (d = dim; d < numDims; d++) do
        C = cardinality[d];
        Partition(input, d, C,dataCount[d]);
        k = 0;
        for (i = 0; i¡C; i++) do // For each partition
            c = dataCount[d][i];
            if ( c >= minsup) then
                outputRec.dim[d] = input[k].dim[d];
                BottomUpCube(input[k ... k+c], d+1);
            endif
            k += c;
        end for
        outputRec.dim[d] = ALL;
end for
```

Figure 2.5: Algorithm `BottomUpCube`

2.3.1 The Cuboid Selection Problem

Materializing the entire datacube requires considerable resources. One of the first problems studied in the context of datacubes was given the frequency distribution with which cuboids of the cube were queried and a space constraint, which cuboids should be materialized. In [HRU96] a polynomial time greedy algorithm was proposed. [SDN98] proposed an algorithm with a similar performance but with faster running times. The heuristic used was to materialize cuboids in increasing order of size. In [GHRU97] the algorithms proposed in [HRU96] were extended for the selection of cuboids *and* indexes given a space bound. [GM99] proposes algorithms for the case where there is a maintenance cost constraint rather than a space constraint. [SDN00] investigates the view selection problem for Multi-cube data models where there are number of datacube lattices which possibly have dependencies between them.

2.3.2 Range Queries

A number of specialized approaches have been developed for answering range queries over datacubes. An example range query would be :

Find the sum of daily sales to customers between the ages of 27 and 45 during the time period December 5th to December 25th.

A range query can be answered by aggregating all the necessary rows of the datacube. An alternative *prefix sum* approach was proposed in [HAMS97] where the idea is to precompute many prefix sums of the datacube offline. Any range sum query over the datacube can be answered in constant time by making use of these prefix sums. This approach can be extended to other distributive aggregates too. The disadvantage is that any updates would require recomputing the entire datacube in a worst case situation. [GAAS99] present a *relative* prefix sum approach which prevents unconstrained cascading updates and achieves reduced update complexity while maintaining constant time queries. [CI99] further improves upon the query update performance with a new class of cube representations called *Hierarchical Cubes*. [GAA00] introduces the Dynamic datacube which trades query performance for improved update performance and is targeted at online range querying of cubes.

2.3.3 Mining Datacubes

The field of data mining or knowledge discovery in databases has experienced rapid growth recently. Some approaches have been developed for data mining in the context of datacubes which we now discuss.

In [SAM98] the problem of detecting outliers in datacubes is investigated. An outlier is a datacube tuple whose measure attribute deviates greatly from its predicted value. A model is introduced to compute the predicted value that takes into account all the aggregates in which this value participates. The model used in this study is similar to table analysis methods used in the statistics community. The datacube computation algorithm, PIPESORT, is modified so that the outliers are computed simultaneously with the tuples of the cube. Though this procedure significantly increases the datacube computation cost it is not as expensive as a two pass approach.

Subsequent work [Sar99] proposes introducing a DIFF operator which explores the reasons for which a certain aggregated quantity is lower or higher in one cell compared to the other. The explanation for the difference is provided in terms other datacube tuples at a finer level of granularity. One challenging aspect of this problem is to express the explanation concisely. The author uses an information theoretic framework for expressing the differences concisely.

Iceberg Queries

Iceberg Queries [FSGM+98] are an important class of queries which perform an aggregate function over an attribute or set of attributes and then eliminates aggregate values that are below some specified threshold. A sample iceberg query over relation $R(target1, target2, target3, ..., targetk, sum)$ is illustrated below.

```
SELECT     target1, target2, target3, ..., targetk, sum(sales)
FROM       R
GROUP BY   target1, target2, target3, ..., targetk
HAVING     sum(sales) > T
```

Iceberg queries are not necessarily over a datacube. In the example query we have a GROUP BY clause rather than a CUBE clause. Nevertheless, they can be extended to be over cubes too. These queries are referred to as iceberg queries because relation R and the number of unique *target* values are typically huge (the iceberg) and the answer is very small (the tip of the iceberg). Many data mining queries are fundamentally iceberg queries. In terms of market basket analysis, the target sets would be item-pairs and T the minimum number of transactions require to *support* the item pair. Computing item sets with frequent support is the dominant cost of computing *association rules* [AS94].

2.3.4 Approximate Cubes

A number of separate approaches have been proposed for providing approximate answers to queries over datacubes. A histogram based approach is proposed in [PG99] which uses MHIST multidimensional histograms. MHIST histograms are multidimensional versions of *maxdiff* histograms which place the $B-1$ bucket boundaries between the pairs of data points with the $B-1$ maximum differences. MHIST histograms are calculated for a chosen set of cuboids of the datacube given a prespecified allowable error bound. Incoming queries are answered on the basis of these histograms. The set of histograms constructed is chosen based on a greedy heuristic.

Alternative approaches have been proposed which are based on wavelets. In [VW99] rather than using histograms to answer queries on the datacube, a *compact datacube* based on a multiresolution wavelet decomposition is used. The compact datacube is built by applying the wavelet transform on each dimension sequentially. Special algorithms are used for sparse data and for range queries. Earlier [SLCJ98] represented the datacube in terms of *aggregated*, *intermediate* and *residual* view elements where these different elements are obtained by making use of Haar wavelets.

2.3.5 Parallel Cube Computation

Techniques have been proposed for parallelizing the datacube computation algorithms described earlier. [DHRC01] develops separate strategies for parallelizing top-down algorithms like PIPESORT and bottom up strategies like BUC. These techniques reduce inter-processor communication overhead by partitioning the load in advance instead of computing each individual cuboid in parallel [GC98].

2.3.6 Caching

[DRSN98] proposed using a cache to enhance query performance over multidimensional queries. The idea was to store previous query results in the cache; incoming queries were broken up into fragments which could be answered using the cache or had to be recovered from the database. This idea was extended in the *Dynamat* system described in [KR99]. Recent work [DN00] has examined a variation of this problem, namely determining when it is possible to answer a query aggregating data in the cache and determining the fastest path for this aggregation since there are many possible ways this aggregation can be carried out.

2.3.7 Integrated Indexing frameworks

[KR98] argues that straight forward relational storage implementation of materialized cuboids is immensely wasteful on storage and inadequate on query performance and incremental update speed. They propose the use of Cubetrees, a collection of packed and compressed R-trees, as an alternative storage and index organization. Here the indexes and the data are stored in one integrated framework.

[JS97] introduce an index structure called a *hierarchically split cube forest* for answering datacube queries. The basic idea is to create an index structure where each level of the index corresponds to an attribute of the datacube. Thus, there are $d!$ possible cube forests for a d dimensional cube. Algorithms are introduced for designing and efficiently querying these structures.

Chapter 3

Optimizing Selections over Datacubes

3.1 Introduction

Often one wants only datacube output tuples whose aggregate value satisfies a certain condition, such as exceeding a given threshold. For example, one might ask for all combinations of model, color, and year of cars (including the special value "ALL" for each of the dimensions) for which the total sales exceeded a given amount of money. This query takes the form of the upper query of Figure 3.1 which we call a "projected datacube" because the aggregate is projected out of the result. In some cases we may need to know the exact value of sales too (the lower query of Figure 3.1).

We can naively execute these queries as follows. Compute the datacube using any of the existing datacube algorithms described in Chapter 2 and check if the predicate in the `HAVING` clause holds for each tuple in the datacube. This

```
SELECT     a,b,c,d
FROM       relation
GROUP BY   CUBE a,b,c,d
HAVING     aggregate(G) relop threshold
```

``Projected Datacube''

```
SELECT     a,b,c,d,aggregate(G)
FROM       relation
GROUP BY   CUBE a,b,c,d
HAVING     aggregate(G) relop threshold
```

``Datacube''

Figure 3.1: A selection over a data cube

strategy is reasonable if a large proportion of the datacube result tuples satisfy the condition. However, if only a small fraction satisfy the condition (i.e, the query is an example of an "iceberg query" described in Section 2.3.3) then it seems that we may be wasting a lot of time computing aggregates that do not qualify.

Depending upon the aggregate function and the relational operator in the predicate, there are certain optimizations we can make use of. In this chapter we propose two kinds of optimization that we call *generalization* and *specialization*. We defer the formal details until Section 3.1.1. For now, we motivate these techniques with examples.

Example 3.1.1: (Specialization) Suppose that we are computing our example datacube query

> Find all combinations of model, color, and year of cars (including the special value "ALL" for each of the dimensions) for which the total sales exceeded $100,000. Output the total sales also.

Suppose that during an intermediate step of the computation, we determine that the total sales for all green cars is below \$100,000. Then we can immediately infer that datacube output tuples grouped by (model,color), (year,color), and (model,year,color) will never meet the threshold when the color is green. If the computation corresponding to those aggregates has not yet been performed, then perhaps we can avoid that computation altogether. □

Example 3.1.2: (Generalization) Suppose again that we are computing our example projected datacube query "Find all combinations of model, color, and year of cars (including the special value "ALL" for each of the dimensions) for which the total sales exceeded \$100,000." Note that in this case we do not need the total sales in the output. Suppose that during an intermediate step of the computation, we determine that the total sales for white 1998 Taurus cars is above \$100,000. Then we can immediately infer that the class of white cars satisfies the condition, the class of 1998 Taurus cars satisfies the condition, etc. We can immediately output all of the additional seven "generalizations" of (white,1998,Taurus). If we have not yet performed the aggregation needed for some of these additional tuples, then we can potentially avoid such aggregation altogether. □

In this chapter we examine how we would make use of generalization and specialization to carry out selections over datacubes efficiently. We extend the `Memory-Cube` and `Partitioned-Cube` algorithms which can deal efficiently with sparse data [RS97b]. `Memory-Cube` computes a set of paths which cover the search lattice and then computes the cuboids on each path in turn. We exploit specialization by altering the set of base tuples with which each path is computed without affecting the correctness of the result. Depending on the selectivity of our condition, we can reduce the number of tuples which are processed in each path leading to substantial improvements in performance. Generalization is incorporated into the algorithms by introducing marker tuples which allow us to skip computing aggregates which are known to satisfy our selection criteria. We demonstrate the efficiency of these modifications by experiments carried out on synthetic and real-world data sets for a variety of selection conditions. These experiments support our overall conclusion that substantial work can be saved.

We further demonstrate that projected datacubes are, in general, easier to compute than datacubes that include the aggregate result. In particular, it is possible to compute projected datacubes with a `HAVING` clause on the median in a *distributive* fashion. In contrast, there is no known distributive algorithm for computing the median in the datacube output. Hence users can get efficient answers to queries such as "Find combinations of model, year and color for which the median sale is greater than $10,000."

3.1.1 Notation and Terminology

A datacube tuple t_1 is more *general* than tuple t_2 if it can be produced from t_2 by replacing one or more of t_2's non-ALL attributes with ALL values. We can restate this by saying that t_2 is more *specialized* than t_1. t_1 and t_2 come from cuboids at different levels of the search lattice with a path from the cuboid of the more specialized tuple (t_2) to the cuboid of the more general (t_1) one.

One property of all the aggregate function discussed is that they have the property of monotonicity. For operators such as MAX, SUM (for non negative numbers), COUNT and $\bigcup$, the aggregate value is monotonically increasing as we

Function	Idempotent	Concordant operators	Discordant operators	Inverse
Min	Yes	$\leq$	$\geq$	
Max	Yes	$\geq$	$\leq$	
Sum on R^+	No	$\geq$	$\leq$	-value
Count	No	$\geq$	$\leq$	-value
$\bigcup$	Yes	$\supseteq$	$\subseteq$	
$\bigcap$	Yes	$\subseteq$	$\supseteq$	

Figure 3.2: Distributive Aggregate Functions and their properties

move from a more specialized tuple to a more general one. The converse holds for operators like MIN and $\bigcap$. Let $G(t_i)$ denote the aggregate computed using aggregate function G which is associated with tuple t_i. A pair (G, op), where G is an aggregate function and op a relational operator, is said to be *concordant* if for all datacube tuples t_1 and t_2 (where t_1 is more general than t_2), $G(t_1)$ op $G(t_2)$ holds. Similarly (G, op) is *discordant* if for all datacube tuples t_1 and t_2 (where t_1 is more general than t_2), $G(t_2)$ op $G(t_1)$ holds. Hence for MAX, $\geq$ is a *concordant* operator while $\leq$ would be a *discordant* operator.

As we shall see each of generalization and specialization behaves in complementary ways for concordant and discordant operators respectively.

3.2 Datacube Algorithms

In this section we examine the applicability of our optimizations to the existing datacube algorithms introduced in Chapter 2.

3.2.1 Array Based Algorithms

The array-based algorithm proposed by Gray et al. [GBLP96] is essentially a main memory algorithm, where all the tuples of the finest level of the datacube are kept in memory as a k dimensional array, where k is the number of `CUBE BY` attributes. The data structure needed for this algorithm will often not fit into memory for sparse relations even when the base relation R does. In this case, the algorithm does not apply. When the algorithm does apply, it requires just a single pass over the data. If we are computing a selection over the data cube, we will not be able to make use of specialization since we gain by this only if we examine a tuple multiple times. We can make use of generalization by outputting a tuple as soon as we know that it will be meet the threshold. This saves us the cost of further aggregations made to that counter in memory, which might be significant if the aggregate function itself is expensive to compute.

3.2.2 PIPESORT

The PIPESORT algorithm [AAD+96] tries to optimize the overall computation of a datacube by providing a set of paths which cover the search lattice and then executing each path in turn. This algorithm makes use of cost estimates of the various ways to compute each cuboid $Q(\vec{B_j})$ to determine which cuboid will actually be used to compute the tuples of $Q(\vec{B_j})$.

Since this algorithm computes various paths in turn and each pass consists of examining all the data in the underlying relation, the techniques we describe with regard to the `Memory-Cube` algorithm of [RS97b] are applicable to PIPESORT too.

3.2.3 OVERLAP

The OVERLAP algorithm [AAD+96] tries to minimize the number of disk accesses by overlapping the computation of the cuboids, by making use of the partially matching sort orders to reduce the number of sorting steps performed.

This algorithm sorts the tuples once. Our scheme for exploiting specialization relies on rearranging and altering the base tuples, hence it cannot directly be applied here. Our technique for generalization improves performance because it allows us to skip aggregations we do not have to carry out. If we are computing aggregations for multiple cuboids concurrently, we may not be able to skip tuples. Hence, OVERLAP is not a good candidate algorithm for our optimizations.

3.2.4 Memory-Cube and Partitioned-Cube

`Partitioned-Cube` and `Memory-Cube` [RS97b], described in Chapter 2, are efficient algorithms for computing datacubes which work particularly well for sparse data. To recapitulate, `Partitioned-Cube` is an algorithm which uses a divide-and-conquer strategy to divide the problem of computing the datacube over a relation with T tuples and k `CUBE BY` attributes into $n+1$ sub-datacubes for a large number n. The first n of these sub-datacubes each has approximately T/n tuples and k `CUBE BY` attributes. The final sub-datacube has no more than T tuples and has $k-1$ `CUBE BY` attributes. Algorithm `Partitioned-Cube` assumes the existence of a subroutine `Memory-Cube` which efficiently computes datacubes for relations that fit in memory.

In the following section we examine how we would incorporate specialization and generalization into these algorithms. We first look at `Memory-Cube` and then at `Partitioned-Cube`. We then perform experiments with the modified version of these algorithms which show considerable speedup for queries with restrictive selections.

3.2.5 Bottom-Up Cube

Bottom-Up Cube(BUC) [BR99] is a recent algorithm developed independently by researchers at the University of Wisconsin. It is similar to a version of `Partitioned-Cube` that never calls `Memory-Cube`. Instead the BUC algorithm builds the cuboid with a single attribute, followed by a cuboid with two attributes and so on. The intuition is that even though work is duplicated while computing these cuboids, this additional work is more than offset by the computation which can be pruned when a tuple does not meet minimum support. The `HAVING` clause considered is of the form `HAVING COUNT(*) >= X` which is equivalent to computing tuples having minimum support in an association rules context [AS94].

BUC prunes computations which are specializations of a tuple but does not exploit generalizations. Since BUC seeks to trade partitioning costs against that of aggregation, it may not be a suitable algorithm if the aggregation operation is expensive.

3.2.6 Apriori

Apriori [AS94] is a popular algorithm for computing association rules. If we are dealing with a concordant relational operator and we know that f (the fraction of datacube tuples (f) which constitute the output) is very small, a variant of Apriori may apply. This algorithm is outlined in Figure 3.3 and proceeds as follows.

```
L_1 = {Large 1-Cuboids}
for (k = 2; L_{k-1} ≠ φ; k++) do begin
  C_k = Generate(L_{k-1})
  forall tuples t ∈ R do begin
    C_t = subset(C_k, t); (Generates all
    possible candidate cuboids from a tuple)
    forall candidates c ∈ C_t
      aggregate t into c
  end
  L_k = {c ∈ C_k | aggregate is large}
end
Answer = ⋃_k L_k.
```

Figure 3.3: Modified Apriori Algorithm

In the first step of the algorithm, we compute all 1-cuboids, namely all tuples which have only one non-ALL element. The amount of memory required for this step is proportionate to the sum of the cardinalities of each of the attributes of the relation. We can tighten this bound by initializing an in-memory counter only for those attribute values which are actually present in the base tuples. Once we have finished a pass over the data, we can determine which of them are large by checking against the threshold. We output all large 1-cuboids (L_1) and discard all small 1-cuboids. In the next step we generate all large 2-cuboids, however we do not use the naive technique of initializing in-memory counters for all possible 2-cuboids. We initialize in-memory counters as follows . We know that if our selection against the threshold does not hold for a particular tuple in a 1-cuboid, it will not hold for for any specialization of the tuple either. Hence, we do not need not initialize in memory counters for any of the 1-cuboid tuples discarded in the previous pass. We generate the set of possible counters (C_2) by choosing all possible pairs from L_1. C_2 should be small since we have discarded all small 1-cuboids. We now scan the data to determine which of the in-memory counters (L_2) satisfy the condition on the threshold.

In a similar fashion, we can generate L_k from L_{k-1}. The only addition in the generic case is that we have an additional pruning step to remove all candidate counters in C_k which have generalizations not present in C_{k-1}. We do not need this step for $k = 2$.

This algorithm is useful only if we know that the number of output tuples is going to be small. If this is not the case, we have the same deficiency as Gray's algorithm; namely, that the number of in-memory counters used is large. If we have n cube by attributes, on the $\lceil n/2 \rceil$'th iteration, we would have $\begin{pmatrix} n \\ \lceil n/2 \rceil \end{pmatrix}$ cuboids where each cuboid had $card^{\lceil n/2 \rceil}$, if each attribute had the same cardinality $card$. This memory requirement of $\begin{pmatrix} n \\ \lceil n/2 \rceil \end{pmatrix} * card^{\lceil n/2 \rceil}$ could easily be in excess of main memory. In fact if we had sufficient memory, in this situation it would be more efficient to directly apply Gray's algorithm rather than split the work across multiple passes.

3.3 Specialization

Let us define a datacube tuple to be *small* if its aggregate is below threshold and *large* if it is above. For concordant operators, we output *large* tuples; for discordant *small*. We can make use of Specialization by taking certain actions whenever we compute a *small* data cube tuple.

3.3.1 MIN and MAX

For MIN and MAX if the relational operator is concordant we can speed up Memory-Cube by simply dropping all base tuples for which the value of the attribute being aggregated is below the threshold. Note that if the relational operator is discordant we cannot make use of this optimization.

`Memory-Cube` can gain from this optimization in a number of ways:

1. The number of tuples for subsequent sort-orders is smaller. The time spent sorting will hence be less.

2. Since the number of tuples has decreased the time spent processing each sort-order is reduced.

3. We still maintain the exact attribute value for large tuples, so we can handle both datacubes and projected datacubes.

We can incorporate this optimization into Memory-Cube by incorporating a preprocessing filter for each tuple t being processed.

3.3.2 Other Aggregates

Let us investigate how we would apply Specialization for an aggregate function g, a concordant relational operator op and a threshold which is a constant T.

Specialization applies when we have computed a datacube tuple in a 1-cuboid. If we have 4 cube by attributes, $< a_1, ALL, ALL, ALL : v_1 >$ would be one such example tuple (v_1 indicates the aggregate computed using function G for this tuple). If we know that this tuple is small, (i.e., v op T is false), we can infer that there will be no datacube tuple with the value a_1 for attribute A in the result. Hence, we can replace all instances of a_1 in the base tuples with a special blank value $\sqcup$. This is distinct from a NULL value. The benefit of this optimization is that if $< a_2, ALL, ALL, ALL : v_2 >$) is also small, we may have altered base tuples of the form $< \sqcup, b_1, c_1, d_1 : v >$ produced by both specializations. These tuples can be combined into a single tuple where v_1 and v_2 are merged using h (See Definition 2.1.1). In a best case scenario, we reduce the number of tuples in the next sort order by a factor of the cardinality of attribute a. We do not increase the number of base tuples in this optimization.

Definition 3.3.1: We define 1-Specialization for a set of base tuples (R) and a datacube query with k `CUBE BY` attributes, an aggregate function g, a concordant relational operator op and a threshold (T). Consider a small 1-cuboid datacube tuple t with value x in attribute j. 1-Specialization consists of replacing x with a $\sqcup$ in the jth `CUBE BY` attribute for all tuples of R. $\square$

Example 3.3.1: We have a datacube with 4 `CUBE BY` attributes and the first sort order we are processing is $ABCD$. Hence the first lattice path which we compute is $ABCD \to ABC \to AB \to A \to \phi$. The next sort order is DAB and the corresponding lattice path is $DAB \to DA \to D$. Suppose both a_1 and a_2 are small.

Consider the following run of tuples:

$< a_1, b_1, c_1, d_1 : v_1 >$

$< a_1, b_1, c_1, d_4 : v_2 >$

$< a_1, b_1, c_2, d_3 : v_3 >$ $< \sqcup, b_1, c_1, d_1 : h(\{v_1, v_6\}) >$

$< a_1, b_1, c_2, d_4 : v_4 >$ $< \sqcup, b_2, c_1, d_1 : v_5 >$

$$
\begin{array}{ll}
< a_1, b_2, c_1, d_1 : v_5 > & < \sqcup, b_2, c_1, d_2 : v_{10} > \\
< a_2, b_1, c_1, d_1 : v_6 > \Rightarrow & < \sqcup, b_1, c_2, d_3 : h(\{v_3, v_8\}) > \\
< a_2, b_1, c_1, d_4 : v_7 > & < \sqcup, b_1, c_1, d_4 : h(\{v_2, v_7\}) > \\
< a_2, b_1, c_2, d_3 : v_8 > & < \sqcup, b_1, c_2, d_4 : h(\{v_4, v_9\}) > \\
< a_2, b_1, c_2, d_4 : v_9 > & \\
< a_2, b_2, c_1, d_2 : v_{10} > &
\end{array}
$$

After using the 1-Specialization optimization, compacting tuples and sorting we obtain the run of tuples on the right from the run of tuples on the left. □

At this point we have to explain how we treat a tuple t which has one or more $\sqcup$'s as attribute values in the context of the `Memory-Cube` algorithm. If the $\sqcup$ occurs in an attribute which is not a grouping attribute in any of the nodes on the lattice path, we can treat these tuples in an identical fashion to other tuples.

In general, if a $\sqcup$ occurs as the ith attribute in the first node of a lattice path with j attributes and the number of nodes of the lattice path is k, we have a formula for determining which accumulators will need to be affected. We use this formula in `Memory-Cube` to combine a tuple with a $\sqcup$ with the appropriate accumulator.

When sorting tuples we can consider a $\sqcup$ to be less than any other value. The other aspect which needs to be addressed is the timing of when tuples are marked with a $\sqcup$. In `Memory-Cube` we sort tuples according to the sort order of the next path immediately after processing a tuple. Now we sort tuples only when we have finished computing a tuple of a 1-cuboid. We then scan all tuples which have just been processed and contribute to this tuple, and replace all values of a_i

with $\sqcup$. After all the cuboids on a path have been computed, we can carry out the sorting and the subsequent compression step. If no 1-cuboids are being computed on a path, we can use the default of sorting tuples immediately.

In a similar fashion to 1-Specialization we can define n-Specialization which is described in detail in [RZ98]. The basic idea is to specialize on n-cuboids for larger values of n. We believe that this optimization is not as useful as 1-Specialization which is why we do not discuss it here.

3.3.3 Generating Paths for `Memory-Cube`

In `Memory-Cube` we generate a set of paths which cover the search lattice and execute them in turn. To make better use of specialization, it is better to execute paths containing a 1-cuboid earlier on, since 1-Specialization gives us the maximum benefit by decreasing the number of tuples. After executing these paths, we can use the previous heuristic of ordering paths lexicographically to maximize work shared across sort orders.

3.4 Generalization

The Generalization principle states that if a datacube tuple is large then all generalizations of this tuple are also large. If the data cube has n `CUBE BY` attributes and the tuple computed has m non-ALL attributes, the tuple has $2^m - 1$ generalizations. The Generalization optimization holds only for projected datacubes where we do not need to know the exact value of the aggregate. Let us see how this optimization would be incorporated into the `Memory-Cube` algorithm.

Whenever a large tuple is computed we know that all generalizations of this tuple are large. Some of these generalizations will lie in cuboids on the current path being executed. We can automatically flush the accumulators for these cuboids which are maintained for the pipelined execution of the path.

We can adjust for the remaining tuples in the following manner by adding a special extra tuple to the set of base tuples. This extra tuple would contain marked values of the non-ALL attributes. When sorting these marked tuples, we use the following rule: A marked value a_i^* is treated such that $a_{i-1} < a_i^* < a_i$. On a subsequent lattice path, we treat marked tuples as follows. If all the grouping attributes for the first node on the lattice path correspond to a *'d attribute, we can immediately reach the conclusion that tuple currently being computed is large and so are all of its generalizations along the path currently being computed. In a similar fashion we can flush the accumulators for these cuboids.

Example 3.4.1: If the first lattice path being examined is $ABCD \rightarrow ABC \rightarrow AB \rightarrow A \rightarrow \phi$, and we have just computed a large tuple $< a_1, b_1, c_1, ALL : v_1 >$, the tuples about which we can make generalizations are the following:

1. $< a_1, b_1, ALL, ALL : v_2 >$
2. $< a_1, ALL, c_1, ALL : v_3 >$
3. $< ALL, b_1, c_1, ALL : v_4 >$
4. $< a_1, ALL, ALL, ALL : v_5 >$
5. $< ALL, b_1, ALL, ALL : v_6 >$
6. $< ALL, ALL, c_1, ALL : v_7 >$
7. $< ALL, ALL, ALL, ALL : v_8 >$

Here, tuples 1,2,4 and 7 lie on the same lattice path and can be handled by flushing the accumulators.

We introduce a tuple $< a_1^*, b_1^*, c_1^*, ALL >$. Let the subsequent lattice path be $CAD \rightarrow CA \rightarrow C$. When we encounter tuple $< a_1^*, b_1^*, c_1^*, ALL >$ we have yet to encounter any tuples of the form $< a_1, b_1, c_1, d_1 >$ since $c_1^* < c_1$.

At this point, we know that we do not have to compute any aggregates for the CA and C cuboids for a_1 and c_1. We next have to alter values for the CA cuboid when we encounter a new value of a. At this point we can write out the values for $< a_1, ALL, c_1, ALL >$ and $< ALL, ALL, c_1, ALL >$ without further computation. We cannot skip computing CAD tuples. $\square$

Since we know what the lattice paths are going to be beforehand, we can encode the last lattice path for which the special tuple would be required and discard the marked tuple after we have computed that lattice path.

It is always a win to generalize for tuples which lie along the current lattice path since we are reducing the number of aggregations. It is not clear that our scheme of introducing marked tuples will always cause an improvement in performance. Since we are introducing extra tuples there will be an increase in the costs of sorting. Furthermore, the benefits of introducing this tuple depend upon how many computations it allows us to skip. This would be affected by which cuboids have already been computed as well as the number of non-ALL attributes in the tuple being generalized.

The gains of generalization are more pronounced when the the cost of computing an aggregate is expensive. If we were dealing with set valued aggregates such as $\bigcap$ over sets of large cardinalities, this would be the case whether we used a bit mapped representation or a list representation.

3.4.1 Discordant Operators

Our treatment of generalization has so far focussed on concordant operators. Let us see how our modifications to `Memory-Cube` are affected if the relational operator is discordant rather than concordant.

Consider a projected datacube with the aggregate function and the relational operator pair being discordant (e.g. MAX and $\leq$). We now want to output small datacube tuples rather than large ones. This enables us to handle datacubes in addition to projected datacubes.

From the perspective of generalization, we have to make just one change. For generalizations along the same path, we can flush the accumulators but in this case we do not output the corresponding tuples. We can still use our technique of introducing marked tuples. Marked tuples can be used to skip computing aggregates, once again the aggregations skipped are not output.

3.5 Partitioned Cube

We have dealt so far only with the `Memory-Cube` algorithm in [RS97b], however we need to address how the optimizations , specialization and generalization could be used in conjunction with `Partitioned-Cube`. The key idea behind this algorithm is divide and conquer. The problem of computing a relation R with T tuples and k `CUBE BY` attributes into $n+1$ sub-datacubes, for a large number n. The first n of these sub data cubes each have approximately T/n tuples and k `CUBE BY` attributes.

Algorithm `Modified Partitioned-Cube`$(R, \{B_1, \ldots, B_m\}, A, G)$

INPUTS:
A set of tuples R, possibly stored in horizontal fragments;
`CUBE BY` attributes $\{B_1, \ldots, B_m\}$,a concordant selection condition S;
attribute A to be aggregated, aggregate function $G()$.

OUTPUTS:
The datacube result for R over $\{B_1, \ldots, B_m\}$ as D on disk.

METHOD:
If (R fits in memory) return `Memory-Cube`$(R, \{B_1, \ldots, B_m\}, A, G, S)$;
else { Choose an attribute B_j among $\{B_1, \ldots, B_m\}$;
    Scan R and partition on B_j into sets of tuples $R_1, \ldots, R_n$;
    Determine large tuples of all 1-cuboids;
    `/*` $n \leq$ card(B_j) and $n \leq$ number of buffers in memory `*/`
    for $i = 1 \ldots n$ {
        Using 1-cuboid information transform R_i to F_i
        let (F_i, D_i) = `Partitioned-Cube`$(F_i, \{B_1, \ldots, B_m\}, A, G, S)$
    }
    let F = the union of the F_i's
    let (F', D') = `Partitioned-Cube`$(F, \{B_1, \ldots, B_{j-1}, B_{j+1} \ldots, B_m\}, A, G, S)$
    let D = the union of D' and the D_i's
    return (D) ;
}

Figure 3.4: Algorithm `Modified Partitioned-Cube`

INPUTS:
A set of tuples R, that fits in memory; `CUBE BY` attributes $\{B_1, \ldots, B_m\}$,a concordant selection condition S; attribute A to be aggregated, aggregate function $G()$.

OUTPUTS:
The datacube result for R over $\{B_1, \ldots, B_m\}$ in two horizontal fragments F and D on disk. F contains the finest granularity tuples and D contains the Output tuples.

METHOD:
```
Sort R and combine all tuples that share all the values of {B_1, ..., B_m};
/* Assume that tuples are sorted according to first sort order */
for each sort order {
    initialize accumulators for computing aggregates at each granularity
    combine first tuple into finest granularity accumulator
    for each subsequent tuple t {
        if ((G is MIN or MAX)&&(tuple is small)) { Drop t from set of tuples }
        Determine position l of first ⊔ in grouping attributes of tuple
        Compare t with previous tuple, to find position j of the first sort
        order attribute at which they differ
        if (grouping attributes of t differ from finest granularity accumulator) {
                combine each accumulator into coarser granularity
                accumulator until grouping attributes match those of t;
                Output those accumulator values which are large.
                Flush all accumulators for coarser granularities .
                /* Number of combinings depends on sort order length and j */
                if (we have just computed a small tuple in a 1-cuboid)
                    introduce blanks into contributing base tuples;
        }
        Combine current tuple with appropriate accumulator;
    }
    Sort according to next sort order making use of shared work if possible
    Merge duplicate tuples produced by specialization.
}
```

Figure 3.5: Modified Memory-Cube for Specialization and Generalization

the final sub-datacube has no more than T tuples and $k-1$ `CUBE BY` attributes. We modify the Partitioned Cube algorithm to optimize for selections. The outline of the process illustrated in Figure 3.6 follows.

We first split the relation on the basis of a partitioning attribute a, where each sub-datacube corresponds to a distinct value of a. While carrying out the partitioning scan we can simultaneously generate all 1-cuboids. This is done by maintaining in memory counters for each of the distinct attribute values. Since we are performing this step for 1-cuboids the memory requirements of this is bounded by the sum of the cardinalities of each attribute. We use this information while executing each of the individual sub-datacubes.

1. If a particular partitioning attribute a_i is not present in any tuple in a 1-cuboid for a, it means that we do not have to compute this sub-datacube at all.

2. Before we start processing each sub-datacube we can carry out a 1-Specialization step over the tuples. In this step, each value of each attribute in a tuple is replaced by a $\sqcup$ if that value is not present in the tuples of the corresponding 1-cuboid. We can compact these tuples before we process the sub data cube. This allows us to gain by reducing the number of tuples before processing any of the sort orders. We carry out this step even if we do not have to compute the sub data cube since we will require these tuples later for computing the sub-datacube with $k-1$ attributes.

The tuples passed to each of the n subcubes (marked with R's in Figure 3.6) each have a distinct value of the partitioning attribute. This immediately means that any generalization of a tuple in datacube i will either be in the same datacube or in the datacube with $k-1$ attributes. Within each datacube we can carry out generalization and specialization.

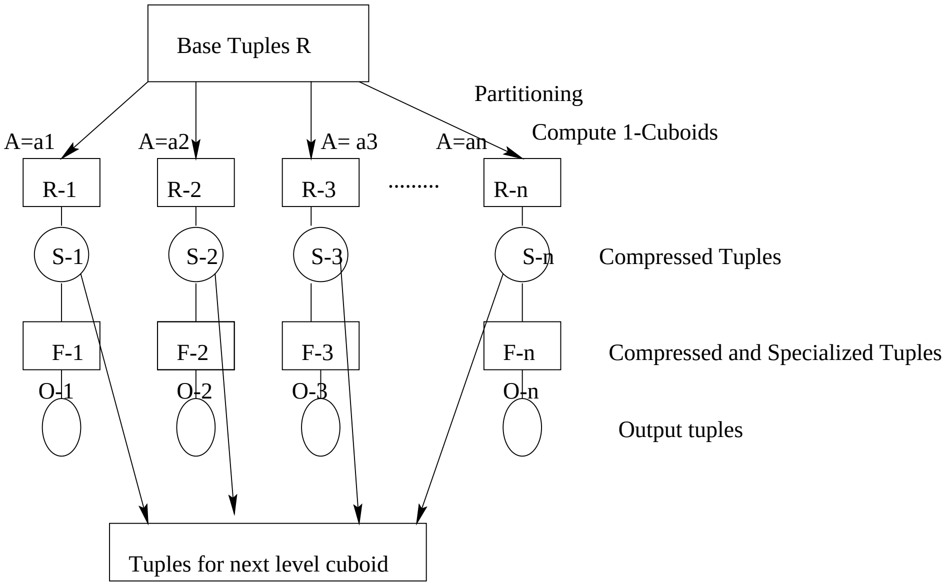


Figure 3.6: Modified Partitioned-Cube

The next issue we have to address is which tuples should be used for the sub datacube with $k-1$ attributes.

For concordant operators over MIN and MAX we only need to pass on those base tuples which are themselves large. For other cases we illustrate the algorithm used by means of the following example.

Example 3.5.1: We are computing a datacube with 5 `CUBE BY` attributes A, B, C, D, E. As indicated in Figure 3.4 we choose a partitioning attribute and split the data into sets depending upon the value of this attribute. We invoke `Memory-Cube` on each set of data. In this example we partition by attribute A (Figure 3.6) to get sets of tuples

$R_1, R_2 \ldots R_n$. We compress each R_i by collapsing tuples with the same CUBE BY attributes to obtain S_i's. We modify `Partitioned-Cube` here to compute all the large tuples of the 1-cuboids which can be done while we are partitioning the data itself. Since we have this information we can now carry out specialization on each set of S_i tuples *before* invoking `Memory-Cube`. Corresponding to each S_i we now have a set of tuples,F_i. We carry out our `Memory-Cube` computation on each F_i to obtain the sub datacube results O_i. In `Partitioned-Cube` the tuples output in each computation of `Memory-Cube` are used for computing cuboids not containing the partitioning attribute. This would be equivalent to using the O_i tuples which we can't do since this contains just those datacube tuples which meet the selection condition. The F_i tuples cannot be used since they contain the $\sqcup$'s produced by specialization. Using the S_i tuples is superior to using the uncompressed R_i's only if the gain due to collapsing tuples offsets the I/O cost of writing and then rereading the S_i tuples. $\Box$

Generalization is unaffected by whether the relational operator is concordant or discordant. For discordant operators we can carry out specialization only for MAX and MIN.

3.6 From Syntax to Semantics

Earlier we identified certain syntactic properties of selection conditions and associated them with corresponding optimizations for enhanced performance. We now try to isolate what semantic properties of selections make these optimizations applicable. [NLHP98] introduced the notion of *anti-monotonicity* of constraints.

Definition 3.6.1: A 1-variable constraint C is *anti-monotone* iff for all sets S, S': $(S \supseteq S')$ & $(S$ *satisfies* $C) \Rightarrow (S'$ *satisfies* $C)$ $\square$

Example 3.6.1: Constraint $MIN(S) \geq T$ is a *anti-monotone* constraint. $MIN(S) \geq T$ implies that every element e in S satisfies $e > T$. $S' \subseteq S$, thus every element in S' is also less than T and the constraint holds for S' too. Thus we have an *anti-monotone* constraint. $\square$

Similarly we introduce the notion of *monotone* constraints.

Definition 3.6.2: A 1-variable constraint C is *monotone* iff for all sets S,S': $(S \supseteq S')$ & $(S'$ *satisfies* $C) \Rightarrow (S$ *satisfies* $C)$ $\square$

Example 3.6.2: $MAX(S) \geq T$ is an *monotone* constraint.
Consider a subset of S, S' which satisfies the constraint. $MAX(S') \geq T$ implies that there exists an element a in S which satisfies $a \geq T$. Since $S' \subseteq S$, a is also present in S, and hence the constraint holds for S too. $\square$

Our generalization optimization holds for *monotone* constraints where our predicate had a concordant operator and we were dealing with projected datacubes (e.g, $SUM(S) \geq t$). The reason this applies is that once we know that once a set of tuples satisfies such a constraint, all supersets of this set will also satisfy the constraint. This is what enables us to skip computations and output tuples without knowing the exact value of the aggregate. Generalization for regular datacubes where we display the aggregate holds for *anti-monotone* constraints like $SUM(S) \leq t$. The reason this applies is that once the constraint is false for a set of tuples, we know that the constraint will not hold for all supersets too. We simply skip computations in this case.

Specialization holds for montone constraints (e.g, $SUM(S) \geq t$) where we display the aggregate. In this case, we know that if the constraint doesn't hold for a set of tuples it will not hold for any subset of these tuples either. While implementing specialization we only consider sets of tuples corresponding to tuples from 1-cuboids of the datacube. Considering other tuples, would require the n-specialization optimization which does not necessarily enhance performance.

3.7 Range Queries and Hierarchies

We can extend these techniques to handle range queries over the datacube. Example of typical range queries are shown in Figure 3.7.

In both these examples, attribute `a` is constrained to take values between `x` and `y`. In a range query, we may have ranges specified on one or more attributes. A range query is distinct from a datacube query with a WHERE clause involving a variable which is not one of the CUBE BY attributes.

If our dataset is too large to fit in memory, we can exploit the range query by using an attribute on which a range is specified as the partitioning attribute. For those values which lie outside the acceptable range, we will not have to call algorithm `Memory-Cube` at all.

We can compute range queries efficiently within `Memory-Cube` by removing all tuples not satisfying the range constraints during the first run where we compute tuples of the finest granularity. These tuples will not be accessed during any further sort orders in this invocation of `Memory-Cube`. This is cheaper than the naive approach of evaluating the query by scanning the entire relation, writing the satisfying tuples to a separate region on disk, and computing the datacube based

```
SELECT     a,b,c,d
FROM       relation
WHERE      x < a < y
GROUP BY   CUBE a,b,c,d
HAVING     aggregate(G) relop threshold

SELECT     a,b,c,d,aggregate(G)
FROM       relation
WHERE      x < a < y
GROUP BY   CUBE a,b,c,d
HAVING     aggregate(G) relop threshold
```

Figure 3.7: Example Range Queries

on these tuples, since it does not require any additional disk I/O.

A similar approach is adopted while dealing with hierarchies. In a query involving a hierarchy, for one or more of the attributes we will not output the value of the attribute but the mapping of that value to a node in the hierarchy. If we assume a hierarchy as shown in Figure 3.8, a possible query could include `division(team)` as one of the CUBE attributes rather than just `team`. In a similar fashion, we can carry out the transformation of `team` to `division` in the preliminary stage of `Memory-Cube` to avoid the additional overhead of disk I/O.

3.8 Holistic Aggregates

For holistic aggregates like the median or other quartiles, data cube computation algorithms cannot compute a datacube tuple from its parents, so each tuple has to be computed directly from the base data. In `Memory-Cube` this means that we have to omit the preprocessing step where we replace the base data with the finest level

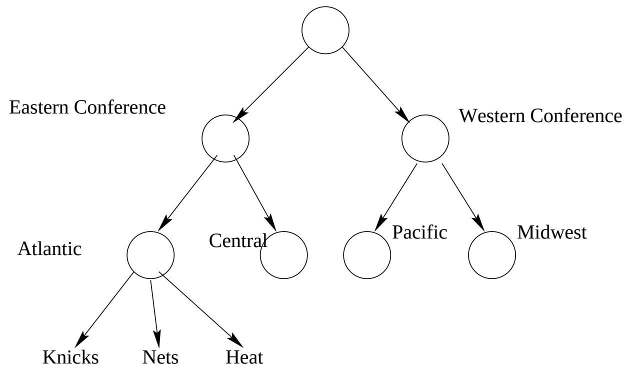


Figure 3.8: A partial hierarchy for the NBA

aggregates.

Surprisingly, for holistic projected-datacube queries of the kind shown in Figure 3.9, we can evaluate the query in a fashion similar to that of distributive aggregates. Instead of maintaining the value of the median in each data cube tuple we maintain two counts: c_1 which corresponds to the `count` of the tuple and c_2 which corresponds to the number of contributing tuples for which the measure attribute satisfies the clause `v relop threshold`. We can use these two aggregates to check if any data cube tuple satisfies the `HAVING` clause by computing c_2/c_1. If we are computing the median and the relational operator is $>$, we have to check if this number is less than 0.5 . In this fashion we can compute any quantile both for $>$ and $<$. We can similarly compute the median for $=$ by maintaining three

```
SELECT      a,b,c,d
FROM        relation
GROUP BY    CUBE a,b,c,d
HAVING      median(v) relop threshold
```

Figure 3.9: Example Holistic Query

counters instead of two.

Lemma 3.8.1: We can check if a quantile is greater than or less than a threshold by maintaining two counts and computing these counts distributively. □

This also works for `Partitioned-Cube`: we change to the count representation during the first partitioning step. This is a big win over computing the exact median since we cannot use `Partitioned-Cube` for holistic aggregates when the data exceeds memory capacity. As demonstrated in Section 3.10, distributive computation is much more efficient than a nondistributive one. However, we can't apply specialization and generalization in the holistic case.

3.9 Multiple Selection Conditions

In our previous examples there has always been a single selection condition in the `HAVING` clause. In the most general case we can have any combination of aggregation functions connected by any combination of boolean operators. The aggregation functions may be distributive, algebraic or holistic and the boolean operator may be `OR` or `AND`.

Let us consider the case where our selection conditions are of the form `distributiveAggregate relop threshold` (where relop is a concordant operator) and are connected only by `AND`'s (the expression is in CNF). In this case, we can carry out 1-Specialization whenever any of the aggregation functions is not satisfied. In the compressed tuple we will have to maintain all the aggregates being checked. Thus, we cannot exploit this optimization if we have a mixture of distributive and holistic aggregates in the `HAVING` clause.

Example 3.9.1: Consider the following query:

```
SELECT     Supplier, Customer, Item, Month,
           SUM(Sales),MEDIAN(Sales)
FROM       SUPPLIES
WHERE      Year = 1996
GROUP BY   CUBE  Supplier, Customer, Item, Month
HAVING     (SUM(Sales) > threshold) AND
           (MEDIAN(Sales)  > threshold2)
```

Since we are computing the median sales in addition to the sum of sales, we cannot carry out 1-Specialization since computing the median requires the actual base data to be computed. □

If we consider the case where we have multiple selections with aggregate functions in CNF, we still have to decide the order in which to evaluate different conditions. This is particularly important if the aggregation operation is expensive. This problem has been studied extensively [Smi56, HS93, KMPS94, CS96] in the database literature. A major issue tackled by recent papers is when expensive predicates should be *pushed* or *pulled* before a join. Since we do not have joins in our scenario our approach will be different. The approach first proposed in [Smi56] orders predicates based on their *rank*, where the rank of a predicate is based on its cost and its selectivity:
$rank = selectivity * (cost - per - application)$ and $selectivity = \frac{cardinality-of-output}{cardinality-of-input}$. Predicates are then evaluated in order of ascending rank. We cannot use this scheme without modification for data cubes since the selectivity of a predicate differs for each cuboid and we are processing more than a single cuboid in a run.

Example 3.9.2: We are computing the aggregation function $\bigcap$ over a set which we represent as a bitmap in each individual tuple. Let the predicate be $\bigcap(bitmap) \supset \emptyset$. If the set cardinality is n and in each bitmap an average of $n/2$ bits are set, the selectivity of the predicate and hence its rank will be high. If we aggregate k tuples for each tuple in the coarser cuboid, the expected number of bits set in a tuple will be $n * (1/2)^k$. This number is considerably smaller than the $n/2$ we started with leading to a very different rank. $\square$

To handle this we propose the following modification to the predicate ordering algorithm if the selection consists of distributive range predicates connected conjunctively. If cuboids $a_1, a_2, \ldots, a_n$ are being computed in a particular run, we compute the global selectivity as $\frac{\sum_{i=1}^{n}(selectivity_i * size_i)}{\sum_{i=1}^{n}(size_i)}$ where $size_i$ denotes the size in tuples of cuboid a_i. We can estimate the size of a cuboid using techniques described in [SDNR96]. Estimating the selectivity of a predicate is normally carried out over the base data by using histograms.

In Example 3.9.2 we use a bitmap representation for a set and the cost of each $\bigcap$ operation is a constant. If we used a linked list representation instead of a bitmap representation the cost of each operation would not be value independent.

We outline how to estimate the selectivity of a predicate for different aggregate functions.

1. COUNT: If we assume that the data is uniformly distributed we can estimate the average count of a tuple by computing the size of the cuboid, taking its reciprocal and multiplying by the size of the base data (n). We use this value to obtain an estimate of 0 or 1 for predicates of the form $COUNT(x) > t$ or $COUNT(x) < t$.

2. MAX/MIN: Calculate the fraction ($frac = base/n$) of the base dataset for which the measure attribute is greater (less) than the threshold. We can use the estimate of the COUNT to determine the selectivity of the predicate by computing $1 - (1 - base/n)^{count}$ for $MAX(x) > t$.

3. SUM: If we assume that the measure attribute takes values drawn from a random distribution with a mean of μ and a standard deviation of σ, the distribution of the sum of *count* such random variables will be given by $(count * \mu, count * \sigma)$. We can now use Chebyshev's theorem which states that the probability of any random variable X will assume a value within k standard deviations of the mean is at least $1 - 1/k^2$. We can compute how many standard deviations away from the threshold the mean is, and then use this theorem to estimate the selectivity.

4. $\bigcap$: Assume the base dataset has an average of a bits set out of a possible total of b for a bitmap based set representation. The expected number of bits set is given by $b * (a/b)^{count}$. We can compute a similar estimate for $\bigcup$.

We use the selectivities in conjunction with the execution costs to order predicate evaluations. We can summarize the applicability of our optimizations for multiple selection conditions in the following table.

Connectives(Type)	Generalization	Specialization
Disjunctive	Yes	No
Conjunctive(Concordant)	No	Yes
Conjunctive(Discordant)	No	No

3.10 Experimental Results

We have implemented the modified version of the `Memory-Cube` algorithm presented in Figure 3.5. We implemented 1-Specialization. We implemented generalization only for tuples in the same path so that performance is not worse than the unmodified version. Specialization will have the maximum effect when we have lots of small tuples in the 1-cuboid while generalization will help us benefit when we have many large tuples in the finer cuboids of the datacube. Thus, our optimizations potentially provide gains in both of these complementary cases.

We implemented this algorithm in `C++` and it computes the datacube of a partition that fits in memory. The data is read in or synthetically generated internally. Data is assumed to consist of 4-byte integer values on all grouping and aggregated attributes, and it is assumed that no extraneous attributes are present. We report results for the SUM and COUNT operators over a single attribute.

We ran the datacube algorithm on a 200 MHz UltraSparc single processor with 256MB of RAM. The algorithm was run when no other processes were active on the system. We measured both the CPU time and the elapsed time. In all our experiments these two measurements were within three percent; we use the CPU time in our results. The time was measured from the point after the input relation was read or generated until the end of the datacube computation.

A large fraction of the time computing the datacube is spent sorting since the number of paths required to cover the search lattice is $\begin{pmatrix} n \\ \lceil n/2 \rceil \end{pmatrix}$ for n `CUBE BY` attributes and each path requires a sorting step. To minimize the sorting cost we adopt a scheme described in detail in [RS98]. We modify the input tables in such a way as to reduce the sorting cost, and then reconstitute the original values at the end. This is done by applying Huffman coding independently to each attribute domain to get bit strings corresponding to each attribute value. We then carry out a counting sort using a composite surrogate key formed by concatenating the Huffman codes from each of the attributes in a sort order. The counting sort operation counts the frequency of each b-bit prefix of this surrogate key for some b. A cumulative histogram is created, and another pass through the data puts it in the order of its most significant b bits. Values that share the same b-bit prefix are then ordered using quicksort. This scheme leads to substantial speedup over quicksort.

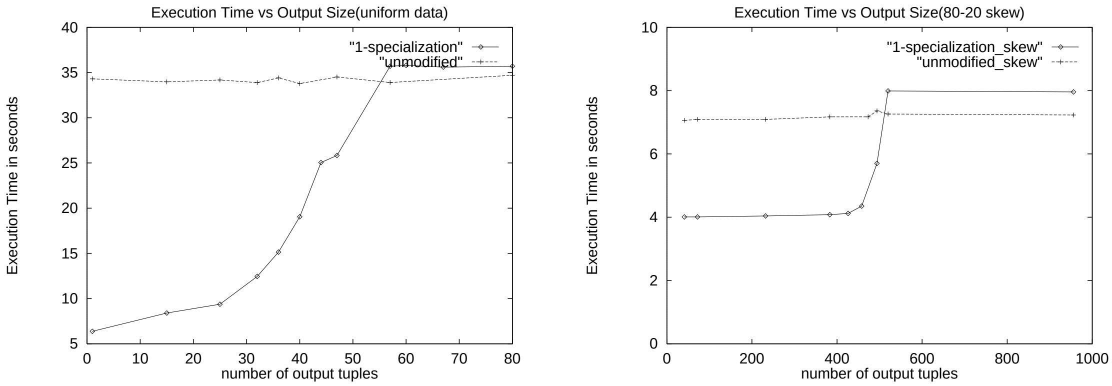


Figure 3.10: Varying the size of the output

In Figure 3.10 we investigate the effect on performance time when we vary the threshold in the selectivity condition. We use a synthetic dataset with 500000 tuples and 6 `CUBE BY` attributes and aggregate function COUNT. The left figure has uniformly distributed data while the right hand figure has 80/20 skew. We see a considerable improvement in performance when the selection condition is restrictive. The degradation of our algorithm when we do not have any opportunity for 1-Specialization is small, the difference in performance is caused by the code introduced for handling $\sqcup$'s. Note that the selectivity of the condition in terms of the number of output tuples cannot be made across data sets. With a particular set of data, we might have many fine output tuples and a few coarse tuples. With a different dataset we might have the same number of output tuples but with a different breakup between coarse and fine tuples.

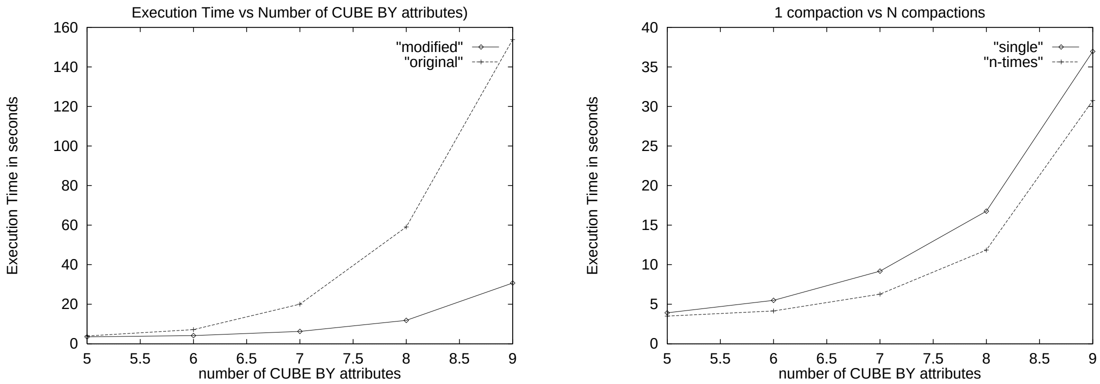
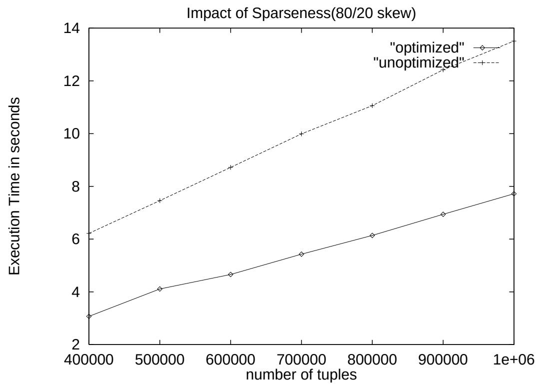


Figure 3.11: Varying number of CUBE BY attributes with constant $|R|$

In Figure 3.11, we vary the the number of `CUBE BY` attributes for a fixed number of base tuples. In each case we generate 500000 tuples with a 80/20 skew and a fixed output size. We can fix the output size by appropriately choosing the threshold for each case. The left hand graph shows how the modified algorithm compares to the original one. In the right hand graph, we study the impact of having a single compression step against multiple compression steps in the `Memory-Cube` algorithm. We can have a single step in which we compress tuples after carrying out all possible 1-Specializations. Alternatively, whenever we carry out 1-Specialization we can compress the tuples. The graph indicates that the second alternative is better.

In Figure 3.12 we study the impact of varying the size of the base relation with a fixed output size and a constant number of `CUBE BY` attributes. We see



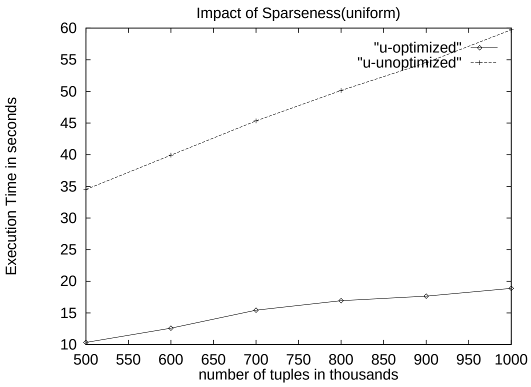


Figure 3.12: Varying $|R|$ with fixed output, `CUBE BY` attributes

that the performance gap between the two versions of `Memory-Cube` widens as the number of tuples increases for both skewed and uniformly generated data.

Example 3.10.1: We also experiment upon real-world data on cloud coverage [HWL94]. The data corresponds to measurements of the amount of cloud coverage throughout the globe over a period of one month, September 1985. We use a data set containing 117,635 tuples for measurements made over the ocean. We have chosen 9 `CUBE BY` attributes out of a possible 20 fields. We have carried out experiments with SUM as the aggregate function. In the graphs of Figure 3.13 we show the difference in performance between the unmodified and modified versions of `Memory-Cube` as well as the difference in performance between having a single compression step and n compression steps. We see that the modified version of `Memory-Cube` does better than the unmodified version for restrictive selections. The right graph indicates that for restrictive selections it is better to have many compression steps, but there is little to be gained in doing so when that is not the case. □

We carry out experiments to show the effect of generalization when we have a relatively expensive aggregation operation. An example of such a query follows.

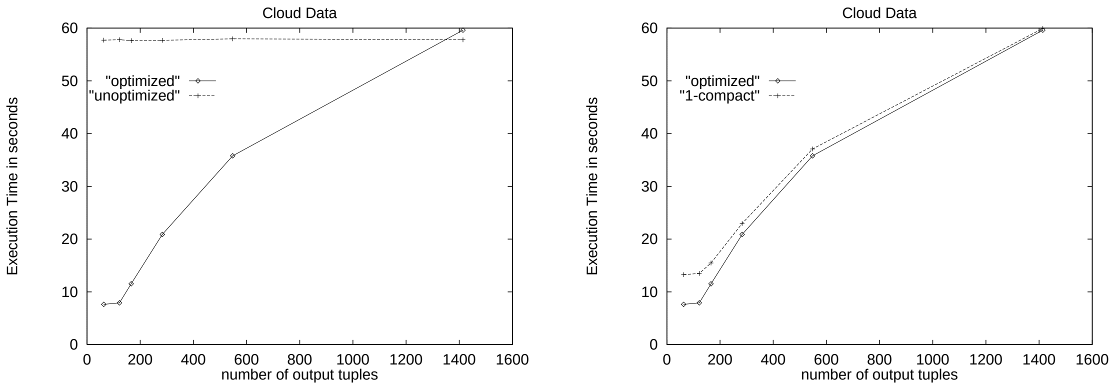


Figure 3.13: Cloud Data

Example 3.10.2: Returning to our example of cars, suppose that instead of using sales as the aggregate column, we use "color". We use a bitmap set representation for the set of colors used at a particular granularity. As discussed above, we also keep a count of the number of (distinct) colors. We can apply the generalization optimization above for a query like "Find combinations of manufacturers and models for which cars of more than a certain number of colors were sold." □

As we can see in Figure 3.14, generalization leads to great improvements in performance. Our function is a variation of the union function where along with the elements of a set we also maintain the count of the number of elements in a set. The set is stored as a bitmap using many aggregation columns. Note that the count is a *distinct* count, and could not be computed without the set being explicitly represented. We output only those datacube tuples which have more than a certain number of elements in a set. This experiment is carried out on a synthetically generated uniform dataset with 100000 tuples. The query used 6 `CUBE BY` attributes.

We carry out experimentation with the same parameters but this time using

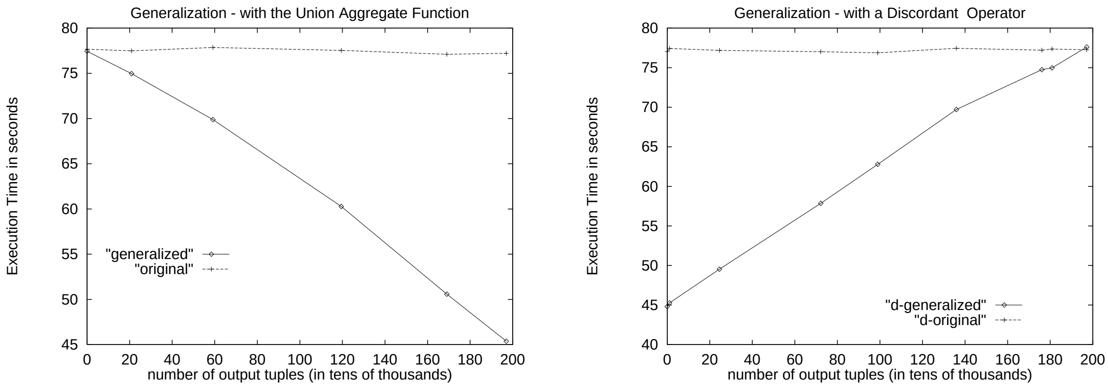


Figure 3.14: Effects of Generalization

a discordant relational operator (we output those datacube tuples with fewer than a certain number of elements in a set) in conjunction with our aggregate function. In this case, we can handle both datacubes and projected datacubes. This is because we do not need to output the tuples we generalize. The results in Figure 3.14(right graph) show that there is a substantial improvement in performance when the number of output tuples is small.

We compare the difference between the distributive and holistic versions of `Memory-Cube` for projected datacube computation. The left hand graph of Figure 3.15 indicates that the improvement in performance for the distributive case increases with the size of the input relation. In the holistic case the median is computed in a quicksort like algorithm that recursively processes one of the two partitions. This algorithm is linear in the average case.

In our final experiment, we show the improvement in performance when the aggregation operation is MIN or MAX and the relational operator is concordant. This is the case where we can drop all small tuples. In the right hand graph of Figure 3.15, we use the real-world cloud data set, MAX as the aggregate function and $\geq$ as the relational operator. We can see that the improvement in performance is the greatest when the number of output tuples is small.

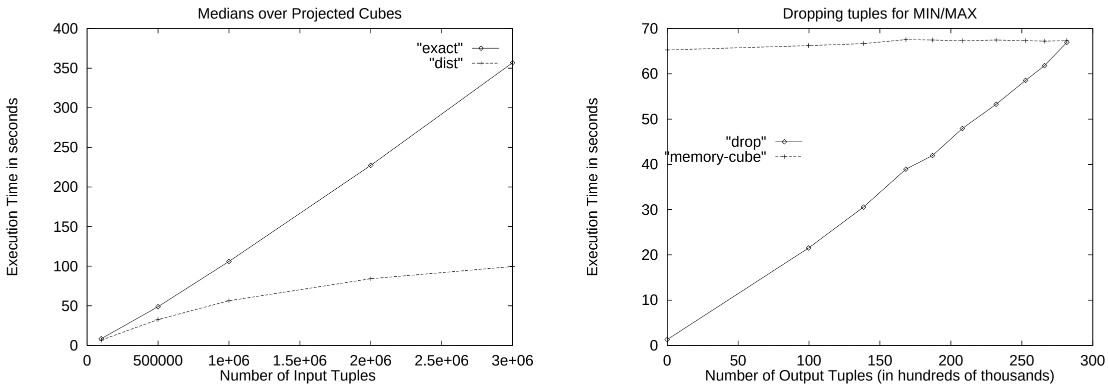


Figure 3.15: Medians and dropping tuples for MIN/MAX

3.11 Conclusions

Datacube queries with selections are important because in decision support environments we are often interested in knowing for which tuples in a datacube a certain condition holds. We have proposed two different ways by which we can use the selection condition internally during the computation of such queries. By making use of the selection condition within the datacube computation, we can safely prune parts of the computation and end up with a more efficient computation of the answer. Our first technique, called "specialization", uses the fact that a tuple in the datacube does not meet the given threshold to infer that all finer level aggregates cannot meet the threshold. We propose a scheme of specialization transformations on the underlying data sets, using properties of the aggregates and threshold functions. We investigated a variation on this idea called n-specialization which does not necessarily improve performance and hence is not worthwhile.

Our second technique is called "generalization", and applies for projected datacubes when the relational operator is concordant. When the relational operator is discordant it holds for datacubes too. Generalization uses the fact that a tuple meets the given threshold to infer that all coarser level aggregates also meet the threshold. We also propose a scheme of generalization transformations. Additionally, we show that for projected datacubes the median is easier to compute. The class of projected datacube queries is an important class of queries which hasn't been considered separately before.

We demonstrate the efficiency of these techniques by implementing them within the sparse datacube algorithm of Ross and Srivastava. We present a performance study using synthetic and real-world data sets. Our results indicate substantial performance improvements for queries with selective conditions.

Related work on datacube computation is described in Section 3.2. Work on reasoning with aggregation constraints is described in [LM96, RSSS98]. The idea of moving predicates for query optimization has been investigated in [LMS94]. The monotonic properties of aggregations has been studied in [RS97a].

Chapter 4

Serving Datacube Tuples from Main Memory

4.1 Introduction

Most OLAP systems precompute some or all of these aggregates to answer queries as quickly as possible. To answer all possible queries over the datacube, we could materialize the entire datacube and store it on disk. The tradeoff is that the datacube may be substantially larger than the base data and may require more space than available. This is especially true for large sparse datasets. In [HRU96], a greedy algorithm is proposed which attempts to maximize the benefit of the set of aggregates picked. In a subsequent paper [GHRU97] techniques for the selection of indices in conjunction with the aggregates were presented. In [SDN98] algorithms with faster running times but which achieve the same performance as [HRU96] are developed.

These previously mentioned schemes materialize datacube tuples on disk and do not exploit the available main memory. Rapidly decreasing main memory prices have lead to workstations with over a gigabyte of RAM. The Asimolar Report on database research [BBC+98] states:

Within ten years, it will be common to have a terabyte of main memory serving as a buffer pool for a hundred-terabyte database. All but the largest database tables will be resident in main memory.

Under the circumstances today, even if we can fit the base table in main memory, we probably will not be able to fit all the tuples of the datacube. In this chapter we develop a framework which enables us to efficiently answer queries under these circumstances by materializing a subset of datacube tuples and storing them in memory. An important benefit of utilizing main memory is that we can provide answers to queries rapidly without having to go to disk. Users are likely to require answers to their queries which are accurate up-to-the-minute. An advantage of our approach is that we do not have to update a large number of previously materialized tuples on disk whenever new tuples are available. We can, instead, efficiently update our in-memory data structures and provide the user with useful running aggregates.

The previous work cited has focussed on selecting datacube tuples for materialization at the cuboid level (we refer to the set of aggregates at a particular granularity as a cuboid). This means that either all the tuples of a cuboid are selected for materialization or none at all. In our framework we can materialize any number of tuples from any cuboid - our unit of materialization is a tuple. This enables us to efficiently answer queries which require a *slice* of a cuboid.

We use a two-level materialization scheme. Our level-1 store contains datacube tuples. Since we cannot expect to materialize all datacube tuples, we store only "high value" tuples in the level-1 store. We analyze what constitutes a "high value" tuple, and demonstrate that tuples with many ALLs and tuples with high query probabilities are good candidates for materialization.

Our level-2 store contains all tuples at the finest granularity. We store the data in a structure that allows a form of partial match query to answer queries without scanning the entire finest granularity dataset. We interrogate the level-2 store only if we find no match in the level-1 store. Our data structures enable fast incremental updates in response to new data, thus allowing the datacube server to supply up-to-date results. Note that only tuples from the finest granularity cuboid are used to populate the level-2 store.

We show how to optimize space between the level-1 and level-2 stores, and how to prioritize tuples for materialization in the level-1 store. Our experimental results show that if the available amount of memory is very small, priority is given to the level-2 store. Once we have an adequate amount of memory the level-2 store tends towards its natural size. The experimental results indicate that for skewed data the choice of a tuple as the unit of materialization is a good one. Our experimental results show that additional memory of a few megabytes enables us to reduce the response time to 2-4 ms for a typical example, which is considerably less than the time required to compute the tuple from the finest level data (194 ms) or to recover a materialized tuple from disk.

We believe that serving datacube tuples from RAM is feasible now for a variety of applications. Further, as main memories increase in size, our techniques will be applicable for more and more applications.

4.2 Notation, Terminology and Cost Models

We assume that the aggregate function(s) to be computed are *distributive* or *algebraic* as defined in Chapter 2. (For simplicity of presentation, we will describe the computation for a single distributive aggregate function; the extension to multiple aggregate functions is straightforward. In practice, one would probably compute several aggregates, such as sum, count, min, max and sum-of-squares, and derive other aggregates from these.)

4.2.1 Queries

For now, a "datacube query" (or just "query") is a request for a single tuple that may be in the datacube. The user specifies values for some of the attributes, and ALLs for the remaining attributes. The answer to the query is the datacube tuple with that combination of attribute values/ALLs.[1] We will consider more general notions of query later in Section 4.4.1.

There are a number of assumptions we could make about the distribution of queries over the datacube. Queries could select all the tuples from a particular cuboid or alternatively we could have *slice queries* [GHRU97] that select a particular tuple from a cuboid. Let the ith cuboid in the datacube have probability p_i of being queried where for a d attribute dataset $\sum_{i=1}^{2^d} p_i = 1$.

We take the most general model of query probability distribution, namely that the jth tuple in cuboid i has a probability $t_{i,j}$ of being queried, where $\sum_j t_{i,j} = p_i$. Our only restriction is that these $t_{i,j}$ probabilities can be calculated either

[1]If the particular combination of attributes in the query does not correspond to an actual datacube tuple, then we can either return a special flag indicating "no tuple," or return a tuple with default aggregate values (e.g., 0 for a count aggregate).

during or immediately after the datacube computation itself, before any queries are actually posed.

Several kinds of query distribution can be justified as reasonable. Both [HRU96, SDN98] make the assumption that each cuboid has the same probability of being queried, i.e., the p_i values are equal. For each i, the $t_{i,j}$ values would also be equal. We call this probability model the *uniform cuboid* distribution.

A second kind of query distribution would assume query probabilities are proportional to a tuple's *count*. We define the *count* of a datacube tuple to be the number of tuples from the base relation R which would need to be aggregated to compute the aggregate associated with the tuple. For example, the count of the tuple $< ALL, ALL, \ldots, ALL >$ would be $|R|$, the number of tuples in the base relation R. (For any cuboid in the datacube, the sum of counts of tuples in it will be equal to $|R|$.) We call this the *count based* distribution.

The motivation for a count based distribution is that users might be most interested in values of tuples for which the most data is available. If an example attribute *state* takes the value of one of the 50 states of the USA, queries will be more likely to specify those states which have substantial amounts of data rather than ones for which little information is available. More detailed motivation for a count based distribution is provided in Appendix A.2.

A third kind of distribution would be to specify a precise weighted query workload. Based on the workload, appropriate probabilities can be assigned to each tuple in the datacube at materialization time. We call this a *workload-based* distribution. Note that, unlike the other distributions considered so far, a workload-based probability distribution might give nonzero query probability to tuples that are not in the datacube. For example, the workload might contain a query like "Give the total sales in Idaho of green BMW cars, irrespective of month sold," even when there are no actual sales of green BMWs in Idaho. Further, as we shall see, it may still be beneficial to explicitly store a tuple indicating this fact in the level-1 store, rather than requiring a search of the level-2 store to answer this query.

A special kind of workload-based distribution is one composed of full granularities. A *full granularity* is a collection of equiprobable queries at a granularity of the datacube where all the non-ALL attributes range over all values from the appropriate domain. We call this a *full-granularity* distribution. This distribution is different from a uniform cuboid distribution that assigns equal probabilities to datacube tuples at a granularity; there may be fewer tuples in the cuboid than queries in the corresponding full granularity.

For example, a full granularity such as "For each car in domain C of cars, and for each state in domain S of states, give the total sales of that car in that state" defines a workload of $|C| \cdot |S|$ equiprobable queries. If a particular car was not sold in some state, then the uniform cuboid distribution would have fewer than $|C| \cdot |S|$ queries. Full granularity workloads are likely to be common, because they correspond to filling multidimensional grids that are frequently used in data analysis.

Each of these three distributions satisfies the condition that "probabilities can be calculated either during or immediately after the datacube computation itself, before any queries are actually posed." In the case of the count-based distribution, we can compute the count aggregate within the datacube computation. On the other hand, a query distribution that becomes apparent only over time as queries are posed does not satisfy our condition. We need to know tuple probabilities in order to properly configure our system.[2]

In what follows, we shall abuse terminology slightly by talking about query tuples "in the datacube" or "in a particular cuboid." For workload-based distributions, we must interpret these statements as including queries at the appropriate granularity that may not actually be in the datacube itself.

4.2.2 Cost Model

In [HRU96], the cost model is a linear cost model where the cost of answering a query Q is the number of rows which need to be read in order to answer the query. This cost model is simplistic and does not take into account whether we have materialized the aggregate in memory or on disk. If the aggregate is to be computed from the base relation on disk, the number of disk I/O's will depend upon which attributes the relation has been clustered. In [GHRU97] the same cost model is used for indices too. A more appropriate cost model is to express the cost of answering a query in terms of the number of memory accesses and the number of disk I/O's required. Since our framework does not involve accessing disk, our analysis will be in terms of number of memory accesses required.

Another assumption made in earlier work is that the cost of answering a query on a cuboid which has not been materialized is equal to the cost of scanning the base table, i.e the number of rows in the table. This assumption will not be true if (as we shall do in this chapter) we organize the base data in such a way that

[2]If the query distribution is unknown, then our framework could be used with the level-1 store being a cache of previously asked queries. Dynamic configuration of such a system is beyond the scope of this work.

we can treat each query for a non materialized tuple as a partial match retrieval query against the base data and a subsequent aggregation of the result tuples.

4.3 The Tuple Serving Framework

4.3.1 Problem Description

Our setting of the problem is as follows. Given main memory M, and a base relation R, how can we appropriately utilize memory to minimize the average cost of executing a query over the d dimensional datacube D constructed over R according to a defined cost model while simultaneously providing update performance at worst $O(log|R|)$. Our problem addresses the selection of tuples to materialize, the definition of appropriate data structures, and the optimization of the system's configuration.

The problem is relatively simple for smaller values of d with moderate attribute cardinalities. In this case we can store a d dimensional array in memory where each dimension takes each possible value (including ALL) of the corresponding attribute. This solution does not scale with larger values of d and high attribute cardinalities since the size of the array will become prohibitively large.

4.3.2 Overall Approach

The overall approach is to exploit the following key points which hold for sparse skewed data.

- We store the finest granularity cuboid, since it cannot be computed from other cuboids. One should use a data structure that allows the computation of coarser granularity cuboids without a complete scan of the finest-granularity data.

- The coarsest granularity cuboids (i.e., those with many ALLs) are the most expensive to compute from the base data. On the other hand, the space needed to explicitly store these datacube tuples is relatively small.

- Skewed datasets and/or query probability distributions produce cuboids where the tuple probabilities are skewed. In such situations it is beneficial to selectively materialize high probability tuples.

We have a two tier framework (Figure 4.1) for handling this problem. We consider a tuple to be a $< K, V >$ pair where K is a composite key constructed from the d attribute values and V is the value to be aggregated. The tuples in the level-1 store may be from any cuboid in the datacube. The level-1 store is represented as a hash table. Given a query q, we first check if the result tuple has been materialized in the level-1 store by hashing on the composite key specified in q. If not, we can compute it from the finest cuboid tuples in the level-2 store.

Our level-2 store shown in Figure 4.1 is similar to that used for partial match retrieval. Intuitively, partial match retrieval is well suited for this problem since partial match retrieval with wild cards corresponds to finding all contributing base tuples ifor a query with ALL's. We have an array of n slots in memory. A set of functions $h_i, i = 1, \ldots, d$ together map each possible tuple to one of these slots. Each finest level tuple is represented as an element in a linked list whose head pointer is in the slot array.

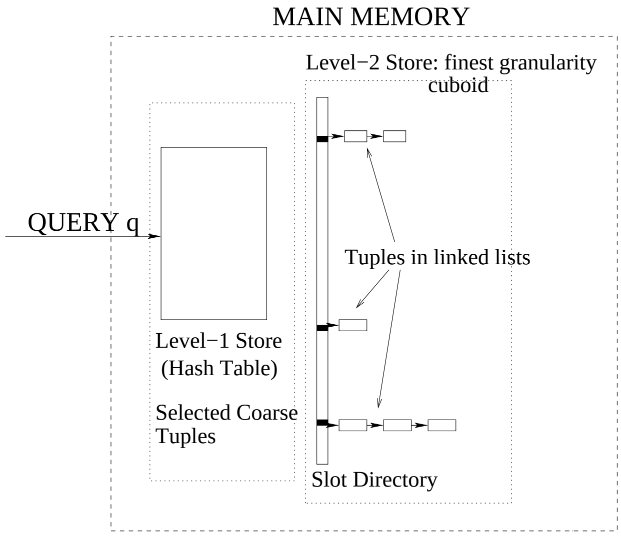


Figure 4.1: Framework for answering queries

Note that tuples in the level-1 store can have ALLs as attribute values, while tuples in the level-2 store cannot.

When a query is answered using the level-2 store, we want to examine just a fraction of the slots. If the query has all its attributes specified we would like to be able to look in exactly one slot. If some of the attributes in the query are not specified, we would like to use the known attribute values to limit the set of slots we need to search. Note that the number of slots is likely to be many orders of magnitude smaller than the product of the cardinalities of the attribute domains.

Suppose that we have space for a slot array with up to T entries. (We'll see how to choose T later.) Then the 1-dimensional slot array can be visualized as a d-dimensional array $\mathtt{S}$ with array bounds of $b_1, \ldots, b_d$ in each of the d dimensions, as long as the product $b_1 b_2 \cdots b_d \leq T$.

We construct d domain mappings $h_1, \ldots, h_d$, where h_i maps values in the domain of attribute i into the range $1, \ldots, b_i$. A tuple $(v_1, \ldots, v_d, x)$ with attribute values v_i and aggregate value x maps to the slot $\texttt{S[}h_1(v_1)\texttt{][}h_2(v_2)\texttt{]}\ldots\texttt{[}h_d(v_d)\texttt{]}$. This tuple is placed in the list whose head pointer is stored in that entry of the slot array. All finest-granularity tuples are placed in the appropriate list.

Consider a query that specifies all but two of the attributes (suppose the first two attributes are ALLs in the query). To answer this query we need to examine only $b_1 b_2$ lists. On average, that means we can ignore $1/(b_3 \cdots b_d)$ of the finest-level data, a significant improvement over a full scan.

There are a number of design issues to be addressed before this framework can be implemented. We haven't yet specified how the b_i values are chosen, nor how the h_i functions are constructed. We also haven't yet shown how to derive T, the preferred size of the slot array, nor how to choose the tuples for the level-1 store. Each of these issues will be addressed in the following sections.

4.3.3 Optimizing the Level-1 Store

If there is sufficient space, one could conceivably store all of the datacube in the level-1 store. Given sufficient memory, we would do just that. However, we expect that we won't have sufficient memory to store the complete datacube since (as can be seen in Example A.2.1 for instance) the datacube is orders of magnitude bigger than the original dataset given sufficiently large attribute cardinalities and a sufficiently large number of dimensions.

Our intuition suggests two guidelines for choosing tuples for the level-1 store. The first guideline says "Materialize tuples with the most ALLs first." The reasoning behind this guideline is that (a) the size of these cuboids will be smaller than the size of cuboids with fewer ALLs, so we get better space utilization, and (b) queries with more ALLs are more expensive in the level-2 store: if we can satisfy these queries in the level-1 store, then we won't need to interrogate the level-2 store.

The second guideline says "Materialize tuples with the highest probabilities first." The main reason for this guideline is that those tuples are more likely to be queried according to our cost model.

The two guidelines are often compatible, since datacube tuples with more ALLs will (in several query distribution models) tend to have higher probabilities. However, we do need to prioritize them in order to determine whether or not to materialize a high-probability tuple with fewer ALLs in favor of a low-probability tuple with more ALLs. We shall address this issue in Section 4.3.5.

We notice that if a tuple is in the level-1 store, it is very likely (or necessary for the count-based distribution described in Appendix A.2) that all coarser tuples are in the level-1 store. Thus, it doesn't pay to try to aggregate tuples in the level-1 store in response to a query. This is borne out by experiments described in Section 4.6.

4.3.4 Optimizing the Level-2 Store

Our level-2 store has T slots. Given a partial match query where some attributes are unspecified, we can efficiently reference the slot addresses which have to be checked by using an appropriate number of nested for-loops. Within the for loops, we traverse the list and accumulate all tuples that match the query. Note that the h_i functions map attributes to a smaller domain, so there may be nonmatching tuples in the list.

Example 4.3.1: Consider a query on an 8-dimensional dataset in which the first two dimensions are ALLs, and the other six are specified. Using the terminology of Section 4.3.2, suppose the specified attributes $v_3, \ldots, v_8$ are mapped to `y3`, ..., `y8` via $h_3, \ldots, h_8$ respectively. Let `b1` and `b2` be b_1 and b_2. Then the query can be answered according to the pseudo-code in Figure 4.2 that assumes arrays start at index 1. We have a total of $b_1 b_2$ lists to traverse. □

Choosing the b_i Values

We now address the question of how to choose the b_i values. At one extreme, we could give high values to some attributes and low values to others. At the other extreme, we could try to balance all b_i values among the various attributes. Which is more appropriate?

An answer to this question appears to require a weighted average of the cost of all queries that reach the level-2 store. However, this weighted sum has a special form that makes it amenable to analysis. All terms in the sum include a product of some of the b_is. Consider b_1 and b_2. The terms fall into four categories: those that include both b_1 and b_2, those that include neither, those that include just b_1, and those that include just b_2. In the case that queries on all cuboids are equiprobable (e.g., the uniform cuboid distribution or the count based distribution) there is symmetry among all the b_is. By symmetry, there is a one-to-one correspondence between terms including just b_1 and terms including just b_2. Thus the overall weighted sum has the form

$$b_1 b_2 X_1 + (b_1 + b_2) X_2 + X_3$$

```
initialize(total); /* for sum, would be "total=0" */
for ( x1=1; x1<=b1; x1++ ) {
     for ( x2=1; x2<=b2; x2++ ) {
        for each tuple in the list starting at S[x1][x2][y3]...[y8] {
            if the tuple matches the query
               accumulate(total,aggregate-value);
           /* for sum, would be "total += aggregate-value" */
        }
     }
}
```

Figure 4.2: Pseudo-code for computing tuples from the level-2 store

for some expressions X_1, X_2 and X_3. If the product $b_1 b_2 \cdots b_d$ is fixed, but b_1 and b_2 are allowed to vary subject to that constraint, which configuration minimizes the expression above? The only term that actually changes is the middle term $(b_1+b_2)X_2$. On the real numbers, this term is minimized (subject to $b_1 b_2$ remaining fixed) when $b_1 = b_2$. If $b_1 \neq b_2$ then we can reduce the term by bringing b_1 and b_2 closer together. Since the choice of b_1 and b_2 was arbitrary, we can apply the same reasoning to all of the b_i values.

Thus we should aim to distribute the b_i values in a balanced fashion, with $b_i \approx T^{1/d}$. If an attribute cardinality is actually less than this computed b_i, then we can use the attribute cardinality and adjust the other b_j values upwards.

The intuition behind the choice of balanced b_i values is the following: Suppose we were to start with a balanced set of b_is, and then doubled b_1 while halving b_2. Then queries depending on b_2 would take twice as long, while queries depending on b_1 would take half as long. Adding up the cost of queries that depend on exactly one of b_1 and b_2 we get a cost equal to $(2+1/2)/2 = 1.25$ times the original cost, while the other queries do not change in cost.

In the general case, where queries on all cuboids are not equiprobable, the optimal set of b_is is obtained by solving a nonlinear integer programming problem. Because of the intractability of finding an optimal solution, heuristic approximations will usually have to suffice.

Choosing powers of 2 for the b_i's might lead to hash functions that are quicker to compute using bitwise operations. However, rounding up/down to powers of 2 makes the hashing less balanced among the attributes, which contributes indirectly to overall performance by requiring more tuples to be scanned in the level-2 store. Further, for most queries the hashing cost will be small compared to the scanning/aggregation cost in the level-2 store.

Choosing the h_i Functions

The properties desirable for the h_i functions are that they should uniformly distribute tuples among the range $1, \ldots, b_i$ as much as possible. By distributing tuples close to uniformly, we avoid the problem of having some very long lists and many empty lists in the slot array: the empty lists would hardly be used by queries, and the long lists would be traversed by many independent queries, leading to poor performance. Like any hash function, there is a limit to how good a job h_i can do in the presence of skew. If some value of attribute i occurs fifty percent of the time, then the image of that attribute under h_i must occur at least fifty percent of the time.

We can be a little more informed than an ordinary hash function, because we can precompute the single-attribute distributions in advance, and adjust the h_i functions accordingly. Consider a domain of size N for attribute i, with $N \geq b_i$

and let $f_1, \ldots, f_N$ be the frequencies of each of the N attribute values. We wish to partition the domain into subsets $S_1, \ldots, S_{b_i}$ such that each S_j is of roughly the same cardinality, and so that $\sum_{x \in S_j} f_x$ is as balanced as possible.

By choosing S_js of the same cardinality, we reduce the number of bits needed to distinguish members of each S_j. By balancing the frequencies, we make the mapping into the slot array more uniform. An exact solution to this subproblem is beyond the scope of this work. A useful heuristic is to allocate domain elements to sets S_j in decreasing frequency order, putting a domain element in the set with lowest total frequency, stopping when the cardinality of S_j is larger that N/b_i.

Some b_i values may be slightly higher than others: some will equal $\lfloor T^{1/d} \rfloor$ and others will equal $\lceil T^{1/d} \rceil$. We can analyze each of the attributes to see which would benefit the most (in terms of spreading the data more uniformly) and choose those attributes as recipients of the higher b_i values.

Once the mappings to sets S_j has been determined, our hash functions can be implemented by storing each (domain-value,j) pair (for a domain value mapping to S_j) itself in a hash table or sorted array.

Optimizing the Size of the Slot Array

It seems that there should be a natural size for the slot array. A slot array that was too big would give poor time performance because too many (mostly empty) slots would need to be checked. A slot array that was too small would give poor time performance because most lists would be very long, with few actual matches, and time would be wasted traversing the lists.

Thus we seek the time-optimal slot array size. Since queries with the most

ALLs will dominate the average cost, we can approximate the overall time-optimum by optimizing the slot array size for queries at the first level not substantially present in the level-1 store; call this the "critical level". Let k denote the number of ALLs at the critical level. In the following analysis, we assume a perfectly balanced distribution of b_i values, and a uniform distribution of tuples among lists.

If T is the slot array size, then we will examine $T^{k/d}$ slots for a critical-level query. Thus the overall cost is equal to

$$T^{k/d}(s + nl/T)$$

where n is the number of finest granularity tuples, s is the cost of a slot access, and l is the cost of a list element access. This function is minimized when

$$T = \frac{nl(d-k)}{ks}$$

So, for example, if $k = d/2$, and $l = s$, we would expect a number of slots equal to the number of finest-granularity tuples.

So far we haven't considered any limitations on memory availability. As we'll see in Section 4.3.5 we will probably settle for a slot array size smaller than the time-optimal size when the memory budget is limited. We may not have sufficient memory for a time-optimal choice. Further, it may be more beneficial to spend memory on the level-1 store than to spend it on reaching the time-optimal slot array size.

4.3.5 Space Issues and Tradeoffs

We will clearly have a limited space budget, and so we need to specify how to allocate that space among the level-1 and level-2 stores.

List Representation in the Level-2 Store

The list representation can be made compact by observing that we don't need to store complete key information in the list nodes. We already have partial information about the key based on the slot that the list emanates from. For attribute i, if g_i is the maximum number of elements that map to a single element via h_i, then we need to store a number in the range $1, \ldots, g_i$ for attribute i. Thus we need to store a number in the range $1, \ldots, g_1 g_2 \cdots g_d$ in each list element. Each list element also needs a pointer to the next list element.[3]

Given the size p of a pointer in bits, we can estimate the total size of the lists in the level-2 store as $n(p + \log(g_1 g_2 \cdots g_d))$ bits. To make the matching more efficient within a list, using bit operations typically found in machine instruction sets, we could specify separate bit ranges for each attribute. In that case the space needed is $n(p + \lceil log(g_1) \rceil + \ldots + \lceil log(g_d) \rceil)$

Assuming that the finest-granularity data fits in memory, this amount is smaller than the available memory. The remaining memory is used for the level-1 store and the slot array in the level-2 store.

Space in the Level-2 Slot Array

Consider a datacube tuple t from cuboid i with k_i ALLs. The cost of looking in the level-2 store for t is $T^{k_i/d}(s + nl/T)$. Thus the average cost of a lookup to the level-2 store is given by

$$A = \sum_{t \text{ not in level-1 store; all cuboids } i} t_{i,j} T^{k_i/d}(s + nl/T)$$

[3] One could optimize the list so that a node contains more than one element, saving some space for pointers, but we do not pursue such optimizations here.

where $t_{i,j}$ is the probability that t is queried, Suppose we keep a running sum of the $t_{i,j}$ values of *unmaterialized* tuples at each value of i in the level-1 store, and denote the sum as C_i for $i = 1, \ldots, 2^d$. Then the equation above reduces to

$$A = \sum_{i=1}^{2^d} C_i T^{k_i/d}(s + nl/T).$$

The derivative of A with respect to T is given by

$$A' = \sum_{i=1}^{2^d} C_i(k_i s T^{k_i/d-1} - nl(d - k_i)T^{k_i/d-2})/d.$$

We will use these expressions to evaluate the tradeoff between the level-1 and level-2 stores below.

Space in the Level-1 Store

Suppose that we've computed the datacube, and are trying to decide which tuples to place in the level-1 store. Materializing a tuple t from cuboid i with k_i ALLs and a probability of $t_{i,j}$ will benefit us by an amount equal to

$$B_t = t_{i,j} T^{k_i/d}(s + nl/T).$$

This amount corresponds to the expected reduction in cost of searching the level-2 store.

Thus we should materialize datacube tuples in strictly decreasing order of their B_t values. Observe that the formula for B_t is exponential in k_i while linear in $t_{i,j}$. We would materialize a level $k-1$ tuple in preference to a level k tuple only if the probability of the level $k-1$ tuple was (roughly) a factor of $T^{1/d}$ higher than that of the level k tuple. As a result, our materialization pattern will likely include a good coverage of the top levels of the cube, with much more scattered coverage of the remainder of the cube.

The Tradeoff Equation

We can measure the cost of storing one level-1 tuple as the product of the tuple size by a hash fudge factor (typically 1.2). Let's suppose that the storage cost per tuple at level-1 is given by q. The storage cost of one slot in the level-2 store is equal to the slot size. Let's denote the slot size by z.

We have some memory left over after accounting for the lists, and we need to decide how to balance these needs among level-1 storage (materializing additional tuples) and level-2 storage (increasing the size T of the slot array). Our approach is to give a unit of memory to the piece of storage the "needs" it the most. We measure the need of the level-2 store as $-A'/z$. We measure the need of the level-1 store as B_t/q where t is the next tuple in line to be materialized. Which of these quantities is greater determines which store will be allocated the next unit of storage.

After such an allocation, the parameters change, and B_t and A' need to be recalculated. We continue this process until we have exhausted all of the available memory. (In practice, we could save computation by allocating memory in batches, rather than one unit at a time.)

4.3.6 Computing the Materialized Tuples

We can use a datacube computation algorithm [AAD+96, GBLP96, RS97b, ZDN97], to materialize datacube tuples on disk. In this cube computation, we compute both the original aggregate as well as any additional aggregates (such as the count) needed to compute the tuple probabilities. We materialize the cube on disk in d files, one for each number of ALLs. An external sort is performed on each of these files to sort them into decreasing probability order.

The finest-granularity file gives us n, the total number of finest-granularity tuples, which we need in order to configure the stores appropriately. We can also calculate the number of distinct values of each attribute, and their frequencies from the tuples with $d-1$ ALL's. This information will help us choose the b_i values and construct the h_i functions.

We then treat each of the d files as a sorted run to be "merged" according to the formula for B_t (Section 4.3.5). Tuples are read and inserted into the level-1 store until all of the memory is accounted for. (We could use an extendible hashing technique such as linear hashing [Lit80] to avoid having to rebuild the hash table as more tuples are read.)

In the case of workload-based distributions, we need to also consider the probabilities of tuples at each granularity that are *not* in the datacube result. In general, we would have to explicitly list all possible combinations of attributes that yielded a nonzero probability, and consider them for materialization in the level-1 store. However, for full granularity distributions, there is a particularly efficient solution that does not require the storage of nonexistent tuples.

Recall that in a full granularity distribution, all combinations of non-ALL attributes at a particular granularity are equally likely. As a result, once one tuple at that granularity is chosen for materialization, all of the remaining tuples at that granularity must be next in line for materialization, according to our formula for B_t. Thus, given sufficient space to materialize the whole granularity, the complete cuboid will be materialized. Rather than explicitly materializing the nonexistent tuples, we can simply switch on a bit indicating that the granularity has been *completely* stored in the level-1 store. Thus, if the bit is on and a query to that granularity finds no match in the level-1 store, we can immediately conclude that the tuple doesn't exist, without consulting the level-2 store. The overhead (2^d bits) is negligible.

At this point we have derived a value for T, the slot array size.

We allocate the slot array of size T, derive corresponding balanced b_i values, and construct the h_i functions. Finally, we read in the finest granularity data and store it in the level-2 store.

4.3.7 Maintaining Materialized Tuples

In our framework, we may have new data being frequently appended to the existing data. In such a situation, we have to update or add a tuple to our level-2 store as well as update any materialized tuples in the level-1 store.

If the tuple t has a matching tuple in the level-2 store, we update its value. Otherwise we add this tuple to the level-2 store. Note that a new tuple will map to exactly one slot, so we traverse just one list on an update.

In general, we may need to check 2^d slots in the level-1 store for all possible tuples that might need to be updated. For a count based distribution, however, if a tuple t is materialized in the level-1 store, any generalization [RZ98] (the tuple formed by replacing one or more attribute values by ALL's) of the tuple has at least as high a count and more ALL attributes, and so must also be present in the level-1 store. When a tuple t is added, the first tuple we update is the tuple with ALL in every attribute position. We then search for generalizations of the tuple on a level by level basis starting from the level of the cuboid where all but one attribute is ALL. If a tuple is not found, we know that no specialization of this tuple would also be found. This breadth first traversal of the cube allows us to cut down on the 2^d possible cube updates.

Another potential optimization of the update process involves keeping a bit for each cuboid indicating whether there is at least one tuple from that cuboid in the level-1 store. One can then cheaply exclude from the 2^d cuboids any cuboid that has no tuples materialized. (We shall see experimentally that this optimization improves update times.)

If the number of appended tuples is large we may need to periodically rebuild our data structures for one of several reasons:

1. There is no space remaining in memory.
2. The underlying data distribution and hence the best choice of h_i has changed.
3. The number of slots may no longer be ideal since the average number of tuples in a slot becomes too large.
4. New domain values appear for some attributes.

We would not expect this reorganization to take place very often, since the base data contributing to the stored datacube is probably orders of magnitude larger than the size increments accumulated over a short period. To plan for future updates we could allow some spare memory for new records, and preallocate some extra attribute domain space for yet-unseen attribute values.

4.4 Extensions

4.4.1 Range Queries

We can extend our framework to efficiently answer range queries. A range query specifies bounds on attribute values rather than specifying an ALL. The simplistic way of answering a range query would be to treat a range query as the equivalent query with ALL's over the level-2 store, and then check if each matching tuple falls within the range. Essentially we are *pulling up* the selection.

We adopt a more sophisticated approach where we construct the h_i functions such that they preserve the domain ordering on the attributes. In other words, if x and y were values in the domain of attribute i such that $x < y$ in the domain ordering, then it must be the case that $h_i(x) \leq h_i(y)$. Then to answer a query in the level-2 store for attribute i between x and y, we need only check offsets between $h_i(x)$ and $h_i(y)$ in the ith index of the slot array. Our hash function h_i now performs a special kind of hashing known as range-partitioning.

Example 4.4.1: Consider a modification of Example 4.3.1 in which the second attribute is used in a range constraint instead of as an ALL. Suppose that we wish to compute the aggregate for values of the second attribute between v_2 and v_2'. Let `low2` equal $h_2(v_2)$ and `high2` equal $h_2(v_2')$. Then the query would be answered using the pseudo-code in Figure 4.3(using the same terminology as Example 4.3.1). We traverse `b1`$*$(`high2 - low2 + 1`) lists. □

The cost of such an approach is that the best h_i functions obtainable under the ordering constraint may be less uniform than an unconstrained choice.

```
initialize(total); /* for sum, would be "total=0" */
for ( x1=1; x1<=b1; x1++ ) {
  for ( x2=low2; x2<=high2; x2++ ) {
      for each tuple in the list starting at S[x1][x2][y3]...[y8] {
          if the tuple matches the query
                  accumulate(total,aggregate-value);
          /* for sum, would be "total += aggregate-value" */
      }
  }
}
```

Figure 4.3: Pseudo-code for Range Queries

A similar technique could be used in the level-1 store if we use a hash function that is similar to the composition of the h_is in the level-2 store. We would need to be certain that all tuples in the query's domain (i.e., with the right number of ALLs) actually appear in the level-1 store. Thus, to support range queries, we have an extra incentive to materialize *all* tuples in a datacube level, rather than leaving out some tuples with very small counts.

4.4.2 Hierarchies

In real life we frequently have hierarchies on attributes. For example, in a dataset on the American National Basketball Association the teams fall into a natural hierarchy (Figure 4.4). In general, the hierarchy may not be a balanced tree. One branch of this hierarchy would be `Knicks`, `Atlantic Division`, `Eastern Conference` specifying the team name, division and conference.

While one user may be interested in calculating the total points scored at the division level, another might be interested in the same statistic at the conference

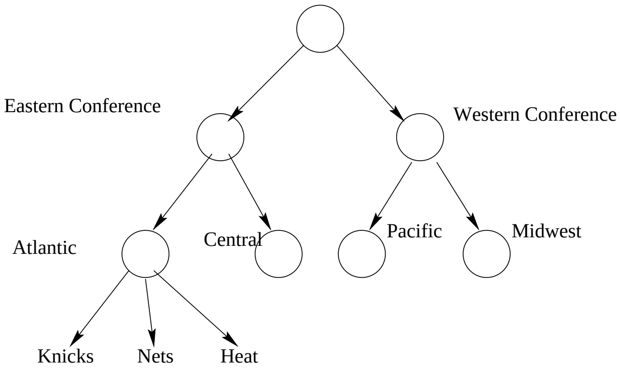


Figure 4.4: A partial hierarchy for the NBA

level. In our base data, we would only be storing the information about the team; it would be a waste of space to store the division and conference since these are dependent upon the team.

We can adapt our scheme to handle hierarchies by mirroring the structure of the hierarchy in our domain ordering. We order our domain lexicographically from higher levels in the hierarchy down to the bottom level in the hierarchy. In the example above, we order the teams according to (conference,division,team). Queries on the hierarchy now correspond to range queries, which we would handle as in Section 4.4.1.

4.5 Optimality and Alternative Frameworks

Our proposed two level scheme performs very well for many data and query distributions. Experiments verifying this are described in Section 4.6. However, it is not possible to make the claim that this framework is optimal for any combination of data distribution and workload.

We illustrate this by constructing certain pathological cases which work best for specially designed frameworks.

Example 4.5.1: Consider a scenario where the number of datacube tuples with a non-zero probability of being queried is equal to the number obtained by dividing the total amount of memory available by the size of a datacube tuple. In this case, it would be optimal to materialize *all* the queried datacube tuples in the level-1 store and not have a level-2 store at all. Every query can be answered in constant time by checking the level-1 store. □

Example 4.5.2: Consider the case where we have a 8 dimensional datacube where all cuboids containing a particular attribute (A) have zero probability of being queried. Rather than using the finest granularity cuboid tuples for populating the level-2 store, we can now use the tuples of the finest granularity cuboid (7 dimensional) which does not containing A. This will reduce the time spent examining tuples in the level-2 store as well as free additional memory for the level-1 store. □

Example 4.5.3: For a 10 dimensional datacube consider the following query distribution. The finest granularity cuboid has a probability of 10^{-6} of being queried. The 1-cuboids of the cube have probability 10^{-2} of being queried where each tuple's probability is proportional to its count. All remaining queries are on the cuboids $ABCD$,$ABCE$,$BCDE$,$ACDE$ and $ABDE$. Each of these cuboids has the same probability of being queried and each tuple of a cuboid has the same probability of being queried. Since the finest granularity cuboid is queried we cannot dispense with the level-2 store. However, we can imagine a "level 1.5" store where the slot array is populated with tuples from cuboid $ABCDE$. This three level framework would be particularly effective if we had sufficient space for materializing cuboid $ABCDE$ but not the 5 cuboids being queried. The level-1 one store would still be useful for the queries for the tuples of the 1-cuboids. □

These examples have been explicitly constructed to demonstrate that different frameworks would be optimal for certain special query distributions. In general, it is unlikely that we will encounter such distributions. Additionally, designing a special framework for each distribution would require identifying the special characteristics of each distribution, a non trivial task in itself. Our two level framework is likely to perform reasonably even for these examples.

4.6 Experimental Results

We experiment with an 8 dimensional subset of cloud coverage data [HWL94] which has 1015367 base tuples. The total number of datacube tuples is 102745662. Each tuple can be represented by 64 bits for the CUBE BY attributes and 32 bits for the aggregate. The finest-granularity data would occupy 12 megabytes, while the full datacube would occupy 1.2 gigabytes.

We also carry out experiments on a uniform dataset. This uniform dataset has the same dimensionality and number of tuples as the cloud coverage data. The cardinality of the attributes are fixed to be the same as those of the cloud data. This enables us to compare results across both datasets. Though the base data contains 1015367 tuples, the datacube itself is larger and consists of 195206797 tuples.

We have implemented the algorithms presented here in `C` and the experiments were carried out on a 300 MHz Sun Ultra-2 running Solaris 5.6. The implementation consists of three distinct modules. The first module analyzes the dataset and computes the optimal size of the level-1 and level-2 stores. The second module chooses the hash functions and bucket sizes for each of the attributes. The third module is the core of the system where any datacube query is answered using the level-1 or the level-2 store. The timing measurements are computed by measuring the total execution time taken for a set of queries and averaging over the size of the set. In the following graphs the horizontal axis shows the number of megabytes available beyond the space needed for the finest-granularity data.

Example 4.6.1: Consider the cloud coverage dataset. Figure 4.5 shows various measures of the performance of our algorithm for a range of available memories. Figure 4.5(a) shows the chosen size of the slot array in megabytes. Figure 4.5(b) shows the space required for the level-1 store in megabytes. Figure 4.5(c) shows the overall query cost to the level-2 store. (The cost of checking the level-1 store is negligible.) Figure 4.5(d) shows the number of tuples from each level in the cube that are resident in the level-1 store. Figure 4.5(e) shows the percentage of tuples of each level in the cube (the number of tuples from a level present in the level-1 store divided by the total number of tuples present in that level in the complete cube) resident in the level-1 store. Figure 4.5(f) shows the average update cost of a tuple for two different strategies. The first strategy checks for each of the 2^d generalizations of a tuple in the level-1 store while the second strategy only looks for generalizations for which at least one tuple has been materialized (as described in Section 4.3.7). The queries are generated according to the count based distribution.

These results show that as we increase memory there is a rapid increase in

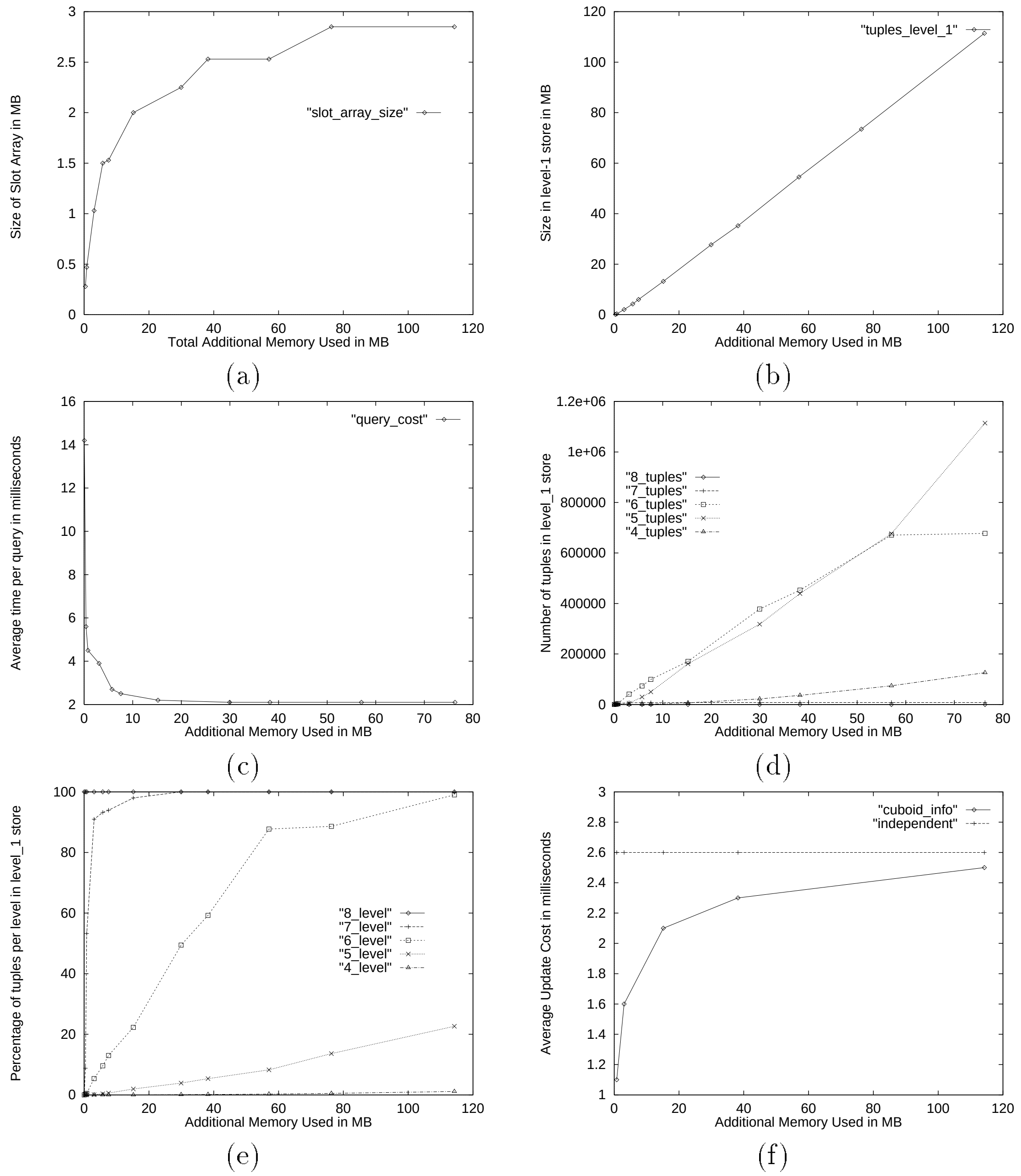


Figure 4.5: Experimental results for Example 4.6.1.

the number of slots which subsequently tapers off. The graph is flat in some places because there is no distinct integer solution which corresponds to the predicted slot array size. In contrast the number of tuples in the level-1 store increases linearly with memory. There is initially a rapid decrease in query cost. As we increase the amount of memory available the decrease becomes less rapid. The initial gain is very sharp since we are first materializing the very expensive tuples with a high number of ALL's. Figure 4.5(d) shows that we do not proceed strictly in a level by level fashion while populating the level-1 store. Figure 4.5(e) illustrates the impact of materializing the high count tuples; the normalized counts of tuples stabilize quicker than the number of tuples in the previous figure. Figure 4.5(f) shows that checking the bit for cuboids with no materialized tuples allows us the save greatly on the update cost when the size of the level-1 store is small. □

Example 4.6.2: Consider the uniform dataset with a count based query distribution. Figure 4.6 shows various measures of the performance of our algorithm for a range of available memories. Figure 4.6(a) shows the chosen size of the slot array in megabytes. Figure 4.6(b) shows the space required for the level-1 store in megabytes. Figure 4.6(c) shows the overall query cost to the level-2 store. Figure 4.6(d) shows the number of tuples from each level in the cube that are resident in the level-1 store. Figure 4.6(e) shows the percentage of tuples from each level in the cube that are resident in the level-1 store. Figure 4.6(f) shows the average update cost of a tuple for two different different strategies outlined earlier.

The results for the slot array size, number of tuples in the level-1 store and the expected query cost are similar to those of the cloud dataset. As indicated in Figure 4.6(d) and Figure 4.6(e), for uniform datasets, we materialize tuples from

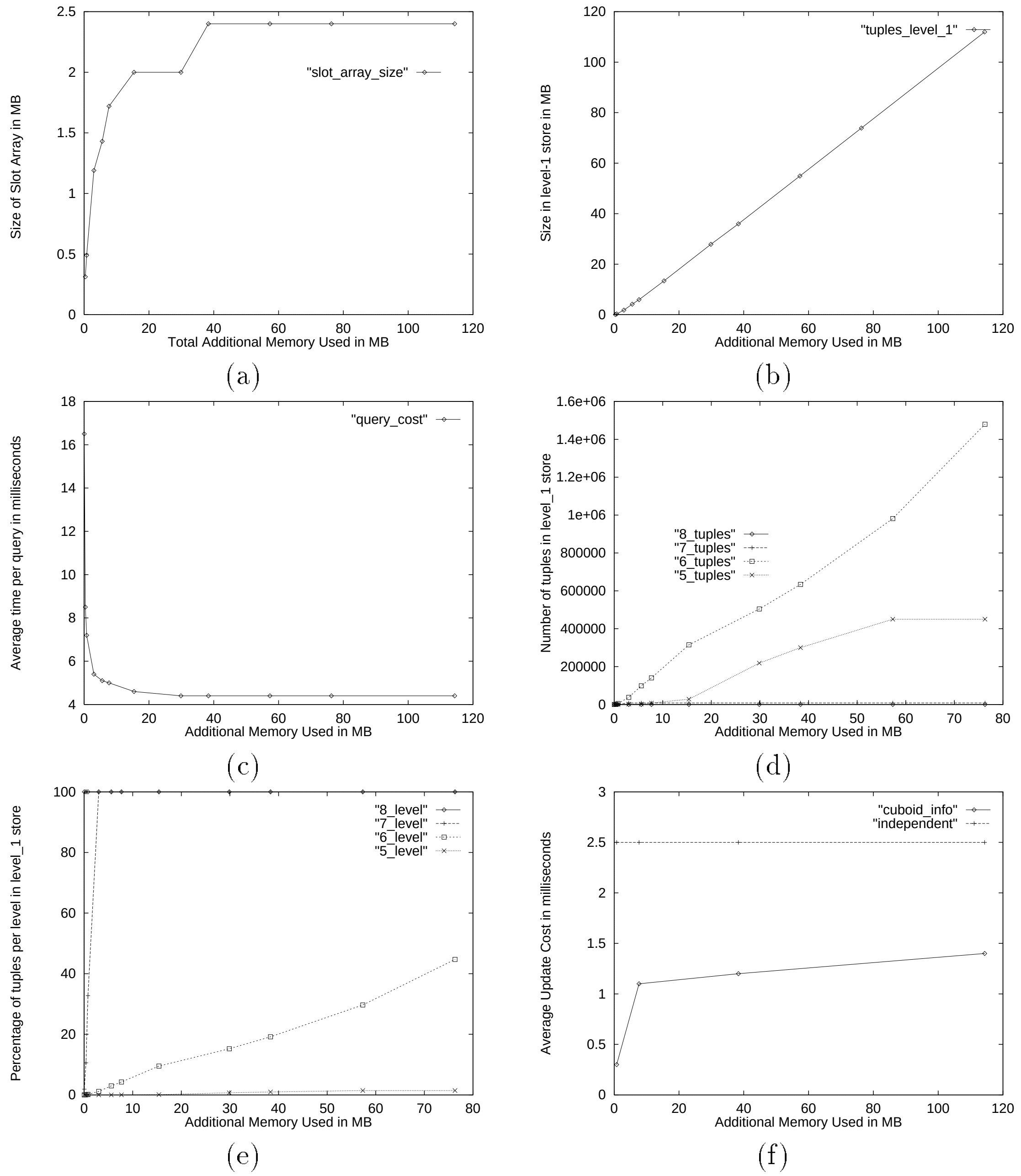


Figure 4.6: Experimental results for Example 4.6.2.

one level only after materializing a considerable fraction of the earlier levels. Each level has a distinct starting point on the X axis (we do not reach the starting points for some of the levels) unlike the cloud coverage dataset where we materialize the high benefit tuples from each of the level at the beginning. For uniform data we tend to materialize whole cuboids (since all tuples in a cuboid have counts close to each other) which is reflected in the fact that the curves are less smooth. Figure 4.6(f) shows that the difference between strategies is significant even when the size of the level-1 store is large. This is because even with a large level-1 store many cuboids are still untouched. □

Some interesting features of our results are the tradeoffs between the slot array size and that of the level-1 store. Initially a slight increase in the slot array size causes a great reduction in the partial match cost (the average number of list elements is greatly reduced), with large amounts of memory the slot-array tends to its natural size. The difference between the uniform and real world data in terms of probability distribution within a level illustrates that for non-uniform datasets it is profitable to choose the tuple over the cuboid as the unit of materialization.

Example 4.6.3: We now experiment with the cloud coverage dataset according to the uniform cuboid query model. The graphs corresponding to this experiment are shown in Figure 4.7. The sizes of the level-2 store and the level-1 vary in a similar fashion to the graphs of the high count query model. The main distinction lies in the composition of tuples materialized in the level-1 store. Since each tuple in a cuboid has the same benefit, the effective unit of materialization becomes a cuboid rather than a tuple. In the graphs of Figure 4.7 we see that all the tuples with 7 ALLS are materialized rapidly while no tuples with 4 ALLS are materialized across the entire range of additional memory shown. This levelwise progression is characteristic of this query model. We see that the update cost difference between strategies is even more pronounced than in the uniform dataset case. □

We see that for all the examples, with just tens of megabytes of additional memory, our response time is between 2 and 4 ms. The time taken to compute a tuple by scanning an array containing the finest granularity data would have been 194 ms. Compared with a disk based approach, our techniques provide query performance faster than a disk lookup on a conventional disk with latencies of at least 10 ms per access.

While we clearly beat disk-based systems for single datacube tuple queries, we may not beat a disk-based materialization for some queries returning sets of tuples. For example, consider a dataset with three dimensions, namely state, year, and model, for a car sales database. A query such as "Find total sales for each combination of year and model in New York" would be particularly efficient in a disk-based system if the data were clustered lexicographically by (state,year,model). The answers would reside on a small number of disk pages, so the I/O cost could be amortized over many tuples. Our approach, on the other hand, would pay a computation cost for each answer tuple, which could add up to more than the I/O cost.

Despite the scenario above, we still expect to beat disk-based materializations for most queries returning sets of tuples. If the query above had instead been "Find total sales for each combination of state and year for Taurus cars," then the clustering by (state,year,model) would not help, and many more disk pages would

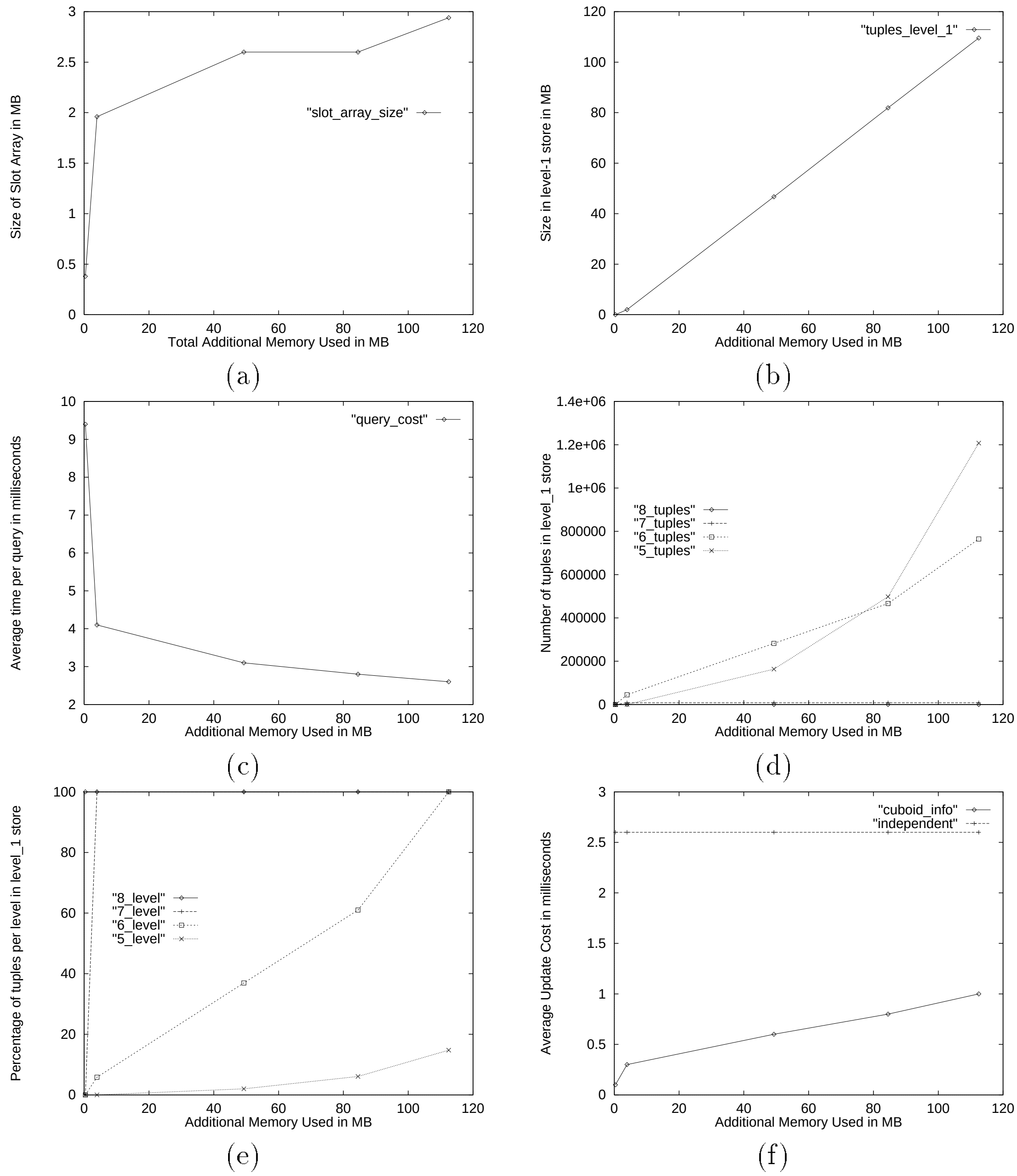


Figure 4.7: Experimental results for Example 4.6.3.

need to be accessed. In the absence of redundancy,[4] there can be just one physical organization. That organization will benefit some queries, but many queries will still require essentially random I/O.

In Figure 4.8 we show the number of buckets assigned to each attribute of the cloud dataset for a estimated level-2 size of 90000 slots. The solution which assigns 86400 slots tries to balance the b_i values, but uses some additional heuristics. For example, any attribute that has a single value occurring more than fifty percent of the time is assigned just two buckets.

Once the b_i values for each of the functions have been assigned we use the methods described in Section 4.3.4 to compute the hash functions themselves. The images of each attribute value for attribute number 1 in the cloud data set are shown in Figure 4.9. The above techniques are used for computing the b_i's and h_i's in all the previously described experiments.

4.7 Related Work

Most related work [HRU96, SDN98] has focussed on the problem of materializing cuboids on disk with a constraint on the amount of disk space available. In [KR98] the problem of efficiently storing and querying the materialized tuples is addressed by using *cubetrees*. An alternative to materialized datacubes called small materialized aggregates (SMA) is introduced in [Moe98].

In [GHRU97], views and indices are chosen to be materialized in terms of a set of queries Q. The algorithms introduced operate upon a data structure called

[4] Redundancy is not a good option, because of the size of the datacube and the number of copies needed to cover all attribute prefixes.

Attribute Number	Cardinality	Buckets Assigned
1	30	6
2	8	4
3	152	5
4	352	5
5	7037	6
6	101	2
7	10	2
8	179	6

Figure 4.8: Number of buckets assigned per attribute

Attribute Value	Hash Image
1, 10, 11, 23, 29	1
5, 6, 16, 17, 26	2
7, 8, 9, 15, 18	3
12, 19, 24, 27, 28	4
2, 13, 14, 21, 25	5
3, 4, 20, 22, 30	6

Figure 4.9: Hash function for attribute 1

a query view graph which contains nodes corresponding to each query as well as each possible view. Note that the total number of possible queries over a datacube is even larger than the datacube itself. Hence, data structures like a query view graph are not scalable for a large set of queries.

Most typical partial match retrieval schemes [AU79, RLT83] are hash based. In these schemes each attribute in a record maps to a bitstring and the we obtain a signature by concatenating the bitstrings. Each record is mapped into a bucket based on its signature. A partial match query is processed by producing the bitstrings for the specified attributes. Based on these bitstrings we can compute a set of candidate signatures by filling in the unspecified bits in all possible ways. We check the buckets corresponding to each candidate signature for matching records. These schemes optimize the number of bits assigned to the bitstring of an attribute based on the probability of that attribute being specified in a query. Our technique is more efficient than partitioning by bits since we are not limited to b_i values being powers of 2.

Our work is distinctive in that we focus on making appropriate use of available main memory. Our unit of materialization is a tuple rather than cuboid, a decision that is appropriate when dealing with skewed rather than uniformly distributed data. We also investigate how to best organize the finest granularity cuboid tuples to best answer queries which require an unmaterialized datacube tuple as the answer.

Chapter 5

Conclusions

We now discuss the contributions of this thesis and potential directions suggested by this work.

5.1 Contributions

In this thesis we have developed approaches dealing both with the computation and querying of datacubes.

Computing selections over datacubes efficiently is very important now that the `CUBE` syntax is well on its way towards being incorporated into the ANSI SQL standard. All major database vendors will soon be supporting these queries, some vendors support them already. To the best of our knowledge, generalization and specialization are among the first approaches proposed for optimizing `CUBE` queries with a `HAVING` clause. We demonstrate the efficiency of these optimizations by implementing them within the `Memory-Cube` and `Partitioned-Cube` algorithms. We present a performance study using synthetic and real-world data sets. Our results indicate substantial performance improvements for queries with selective conditions.

With continual advances in memory capacity, PC's on the desktops of users now commonly have close to half a gigabyte of RAM. Currently, the cost of one gigabyte of RAM is in the neighborhood of $2,000. Even small companies can afford to buy hundreds of gigabytes of RAM. With the motivation of making the most of these resources, we have introduced a main memory based framework for answering datacube queries efficiently. We exploit the fact that while the entire datacube is much larger than available memory, there is often enough space for the finest granularity cuboid. By materializing a well chosen set of tuples in memory, it is possible to compute any datacube tuple efficiently without having to go to disk. Apart from the reduced time for processing queries this also requires considerably less maintenance cost than a disk based scheme in an append-only environment. Our experimental results show query performance in the 2-4 ms range for a typical example.

5.1.1 Broader Applicability of Proposed Techniques

Techniques developed in this thesis could conceivably be used outside the domain of datacubes. [NLHP98] discusses how anti monotone constraints can be integrated into a framework for mining association rules. The techniques of generalization and specialization could be adapted into such frameworks. The idea of pruning unnecessary computations is fundamental in Computer Science. Branch and bound techniques are frequently used to reduce the search space explored in search problems. One difference which should be noted is that we are not searching for a single optimal solution; any datacube tuple which satisfies the constraint is a solution. We aim to output *every* valid solution. In the future we are likely to see expensive operations like querying a search engine for metadata integrated into the database system. Identifying which operations could be skipped would greatly enhance query performance.

The idea of multiple levels stores containing items based upon the frequency with which they are queried originates from caching both at the operating system and architectural levels. In recent years the problem of web caching has also received considerable attention. The primary difference in our framework is that any desired item can be *computed* from the level-2 store. Rather than having a fixed miss penalty, the cost incurred is the number of items that we have to examine from the level-2 store. The idea of using certain data items in the cache to compute items not present there has been examined in the context of ROLAP systems [DN00]. We can envision the level-2 store or other partial match based schemes being used in web based publish-subscribe system where a given event has to match subscriptions along many possible dimensions.

5.2 CUBE: Effects on Query Optimization and Evaluation

We now discuss how the CUBE operator would fit into the query optimization and evaluation framework of a database system.

In a data warehousing environment, we frequently have a *star schema* with a central fact table and auxiliary dimension tables which containing additional information about the dimensions of the fact table. For example, details of a hierarchy on one of the dimensions would be contained in a dimensional table. Users may be interested in joins between the dimensional tables and a datacube constructed over the fact table.

One standard technique used while evaluating GROUP BY's in conjunction with a join is to carry out the GROUP BY first (when possible) since it results in the reduction of the number of tuples. Thus, the input to the join is smaller in size which is beneficial since join processing is expensive. This is not likely to be the case for the CUBE operator, since datacubes typically tend to be larger in size than the base table. When there is a selection over a datacube, pushing the selection may lead to a reduction in the number of tuples. This will be most pronounced for highly selective conditions. In many cases, evaluation of expensive selection conditions is delayed to reduce the overall process cost. If the selection is over a datacube, the cost of evaluating the selections over each tuple has to be weighed against the amount of computation saved by optimizations like specialization and generalization. [ZCL+00] describe how materialized datacubes can be represented in a *query graph* model and used by the query optimizer where applicable. The *query graph* model is the internal representation used for a query in IBM's DB2 Universal Database.

Query evaluation frameworks often have different algorithms for implementing the same operations. For example, a database system may implement nested loops, sort-merge and hash joins. It would be useful to have multiple datacube computation algorithms present in a system since there is a wide range in performance depending upon the nature of the dataset. Statistics maintained over the base relation should be used to pick the appropriate cube computation algorithm. For example, if the dataset is very sparse `Partitioned-Cube` and `Memory-Cube` would be a good choice. If the data is dense and the total size of the datacube is less than available memory, the algorithm proposed in [GCB+98] would be very efficient. The applicability of specialization and generalization would depend would depend upon the algorithm chosen. In Chapter 3 we have discussed the particulars of how these optimizations apply for various cube computation algorithms. The benefits of these optimizations should be incorporated into the cost estimates of a plan by the query optimizer.

5.3 Future Work

We discuss potential extensions to the work described in this thesis.

5.3.1 Main Memory based Tuple Serving

There are a number of interesting problems arising from our main-memory based tuple serving framework. In some cases the finest-level data may not fit in main memory, but will be somewhat larger than main memory. How might we gracefully transition into a hybrid system with part of the data on disk, and part of the data in RAM ?

What could we do for nondistributive aggregates in RAM? We now have to materialize the base data, not the finest aggregates, although we could probably materialize one tuple of dimension values with a list of corresponding aggregate values instead of many tuples.

It seems that the cost should now also include the cost of copying the aggregates to some temporary array and calling an aggregate function on the array. The aggregate function may have nonlinear complexity in the size of the array. In that case, it seems that the coarser tuples are even more valuable to cache. Can the cost functions be modified in a way that takes the cost function of the aggregate as a parameter, so that our overall approach is still valid with different cost functions?

5.3.2 2 Variable Constraints

Consider a selection condition of the form `HAVING sum(x) > sum(y)`. This predicate is different from the selection conditions we have discussed earlier, since it has a relational operator between functions of two variables. Our techniques of specialization and generalization do not apply for two variable constraints. Specialization does not apply since a non qualifying 1-tuple does not allow us to draw any conclusions about either of the attributes occurring in the output. Dealing with two variable constraints is hard in other related problems too [LNHP99]. It would be useful to come up with applicable optimizations rather than having to resort to the naive strategy of checking if the predicate holds for each computed datacube tuple.

Appendix A

A.1 Table of Symbols

Number of dimensions	d
Number of slots in the level-2 store	T
Number of tuples in the finest granularity cuboid	n
Cost of examining a slot	s
Cost of examining a tuple in a linked list	l
Benefit of materializing tuple t	B_t
Average cost of a lookup to the level-2 store	A
Size of a tuple	q
Size of a slot	z
Count of a tuple t	c_t
Pointer Size	p

A.2 Motivating a Count Based Model

In [HRU96] the assumption is made that choosing a set of datacube tuples to materialize is equivalent to deciding which cuboids to materialize. This assumption is not always justified, particularly in the case where the data does not follow a uniform distribution but is skewed and we are considering the universe of slice queries over the datacube.

Example A.2.1: Consider a 9 dimensional subset of cloud coverage data taken from [HWL94]. We contrast the behavior of this dataset with a uniform synthetic dataset having the same cardinality and the same attribute cardinalities as the cloud coverage data.

In Figure A.1 we examine how the number of base tuples contributing to a datacube tuple vary. We also see how the number of datacube tuples vary for each level of the lattice. We notice that for sparse datasets like this one the size of the datacube is larger for the uniformly generated synthetic dataset (46536156 tuples to 28171031). This is expected since skew in the dataset leads to a smaller number of distinct tuples. If the density of the datacube (the ratio of the number of distinct tuples to the product of the cardinalities of the attributes) is greater than 1, we would expect the size of the datacubes to be the approximately the same.

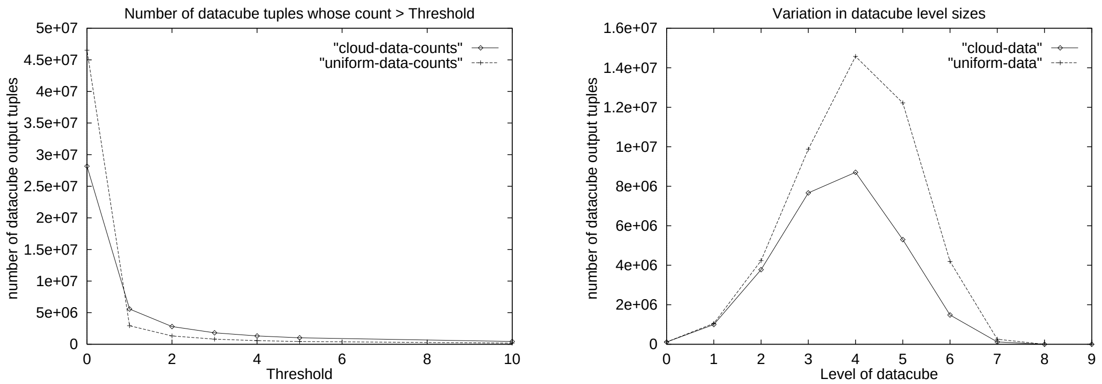


Figure A.1: Counts of datacube tuples and size of datacube levels

We observe that in both cases the majority of the datacube tuples have counts less than 5. For the cloud data only 3.63 percent of the datacube tuples have a count greater than 5. This implies that even if the cost of an aggregation is expensive, for the majority of the tuples the cost of computation is heavily dominated by I/O costs.

A key difference is how the *counts* of tuples in a cuboid are distributed. If the base data is uniformly distributed there will be a low variance in the counts of tuples in the cuboid. For both random and skewed data the variance of the counts in a cuboid are low at the lower levels of the cube (cuboids with a larger number of attributes). The reason for this is that since the data is sparse most tuples in the cube have a count of 1. At the higher levels of the datacube the skewed data exhibits greater variance in the counts of a cuboid than uniform data.

We define *ccount* as the percentage of tuples in a cuboid with a *count* greater than a threshold. For uniform data distribution we see that there are relatively few cuboids where *ccount* lies between 10 and 90 percent. In this case it is a reasonable approximation to treat the whole cuboid as a single unit.

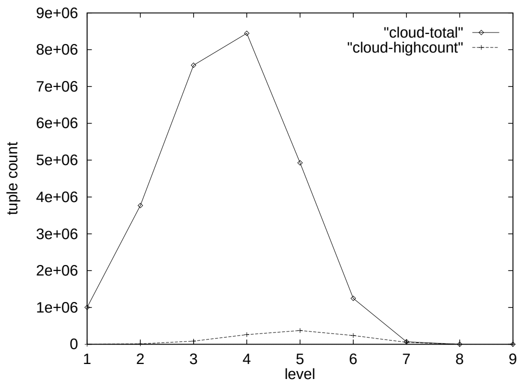
However, this is not the case for the cloud data. Here we have a large number of cuboids where the percentage of tuples exceeding the threshold lies between 10 and 90. This implies that by treating a cuboid as a single unit we are giving the same treatment to tuples with widely varying counts. This is the motivation for choosing a model with different probabilities for tuples within the same cuboid.

Uniformly distributed data:

Threshold	$ccount < 10$	$10 < ccount < 90$	$ccount > 90$
1	341	80	90
2	384	45	82
5	415	25	71
10	436	13	62

Cloud data:

Threshold	$ccount < 10$	$10 < ccount < 90$	$ccount > 90$
1	101	376	34
2	199	284	28
5	287	207	17
10	354	144	13



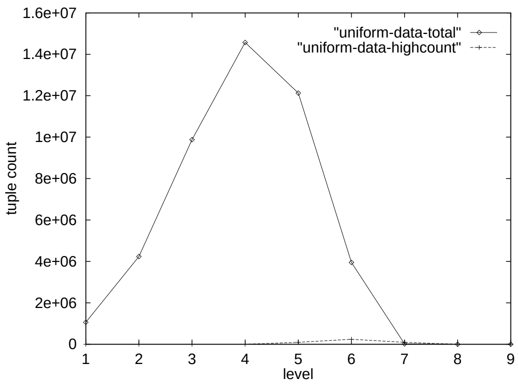


Figure A.2: Total number of tuples and number of tuples with a count exceeding a threshold count of 5 per level for both skewed and uniform data

Bibliography

[AAD+96] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22nd International Conference on Very Large Databases*, Mumbai,India, 1996.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, Santiago, September 1994.

[AU79] A. Aho and J. Ullman. Optimal partial-match retrieval where fields are independently specified. *ACM Transactions on Database Systems*, 4(2):168–179, 1979.

[BBC+98] Phil Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, H. V. Jagadish, Michael Lesk, Dave Maier, Jeff Naughton, Hamid Pirahesh, Mike Stonebraker, and Jeff Ullman. The Asilomar report on database research. *ACM Sigmod Record*, 27(4), 1998.

[BR99] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proceedings of the 1999 ACM SIGMOD Conference on Management of Data*, Philadelphia, 1999. Association for Computing Machinery.

[Cha98] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Seattle, 1998.

[CI99] C. Chan and Y. Ioannidis. Hierarchical cubes for range-sum queries. In *Proceedings of the 25th International Conference on Very Large Databases*, Edinburgh, Scotland, 1999.

[CS96] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Proceedings of the 22nd International Conference on Very Large Databases*, Mumbai,India, 1996.

[DHRC01] F. Dehne, S. Hambrusch, and A. Rau-Chapin. Selection of views to materialize under a maintenance cost constraint. In *Database Theory - ICDT '01, 8th International Conference, Proceedings*, 2001.

[DN00] P. Deshpande and J. Naughton. Aggregate aware caching for multi-dimensional queries. In *Proceedings of the Conference on Extending Database Technology, (EDBT-00)*, 2000.

[DRSN98] P. Deshpande, K. Ramasamy, A. Shukla, and J. Naughton. Caching multidimensional queries using chunks. In *Proceedings of the 1998*

ACM SIGMOD Conference on Management of Data, Washington, Seattle, 1998. Association for Computing Machinery.

[FSGM+98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. In *Proceedings of the 24th International Conference on Very Large Databases*, New York, August 1998.

[GAA00] S. Geffner, D. Agrawal, and A. El Abbadi. The dynamic data cube. In *Proceedings of the Conference on Extending Database Technology, (EDBT-00)*, 2000.

[GAAS99] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Prefix sums: An efficient approach for querying dynamic olap data cubes. In *Proceedings of the 15th IEEE Conference on Data Engineering*, Sydney, Australia, 1999. IEEE Computer Society.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Datacube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the IEEE Conference on Data Engineering*, pages 152–159. IEEE Computer Society, 1996.

[GC98] S. Goil and A. Choudhary. High performance multidimensional analysis of large datasets. In *DOLAP '98, ACM First International Workshop on Data Warehousing and OLAP, November 7, 1998, Bethesda, Maryland, USA, Proceedings*, pages 34–39. ACM, 1998.

[GCB+98] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Riechart, M. Vekatrao, F. Pellow, and H. Pirahesh. Data Cube: A relational aggregation operator generalizing group-by, cross-tab and sub-total. *Data Mining and Knowledge Discovery*, 1(1), 1998.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for OLAP. In *Proceedings of the 13th ICDE*. IEEE Computer Society, 1997.

[GM99] H. Gupta and I. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, 1999.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[HAMS97] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *Proceedings of the 1997 ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997. Association for Computing Machinery.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD Conference on Management of Data*, Montreal,Canada, 1996. Association for Computing Machinery.

[HS93] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD Conference on Management of Data*. Association for Computing Machinery, 1993.

[HWL94] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. Available from http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html, 1994.

[JS97] T. Johnson and D. Shasha. Some approaches to index design for cube forest. *Data Engineering Bulletin*, 20(1):27–35, 1997.

[KMPS94] A. Kemper, G. Moerkotte, K. Peithneri, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, Minneapolis, Minnesota, 1994. Association for Computing Machinery.

[KR98] Y. Kotidis and N. Roussopoulos. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *Proceedings of the 1998 ACM SIGMOD Conference on Management of Data*, Seattle, Washington, 1998. Association for Computing Machinery.

[KR99] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *Proceedings of the 1999 ACM SIGMOD Conference on Management of Data*, Philadelphia, 1999. Association for Computing Machinery.

[Lit80] W. Litwin. Linear hashing: a new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Databases*, 1980.

[LM96] A. Levy and I. Mumick. Reasoning with aggregation constraints. In *Proceedings of the Conference on Extending Database Technology, (EDBT-96)*, Avignon, France, 1996.

[LMS94] A. Levy, I. Mumick, and Y. Sagiv. Query optimization by predicate movearound. In *Proceedings of the 20th International Conference on Very Large Databases*, Santiago, September 1994.

[LNHP99] L. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proceedings of the 1999 ACM SIGMOD Conference on Management of Data*, pages 157–168, Philadelphia, 1999. Association for Computing Machinery.

[Moe98] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the 24th International Conference on Very Large Databases*, New York, August 1998.

[NLHP98] R. Ng, L. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proceedings of the 1998 ACM SIGMOD Conference on Management of Data*. Association for Computing Machinery, 1998.

[PG99] V. Poosala and V. Ganti. Fast approxiamte answers to aggregate queries on a data cube. In *11th International Conference on Scientific and Statistical Database Management*, 1999.

[RLT83] K. Ramamohanrao, J. Lloyd, and J. Thom. Partial-match retrieval using hashing and descriptors. *ACM Transactions on Database Systems*, 8(4):552–576, 1983.

[RS97a] K. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.

[RS97b] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proceedings of the 23rd International Conference on Very Large Databases*, Athens,Greece, 1997.

[RS98] K. Ross and D. Srivastava. Fast computation of sparse datacubes. *Full Version*, 1998.

[RSSS98] K. Ross, D. Srivastava, P. Stuckey, and S. Sudarshan. Foundation of aggregation constraints. *Theoretical Computer Science*, 193(1):149–179, 1998.

[RZ98] K. Ross and K. Zaman. Optimizing selections over data cubes. Technical Report CUCS-011-98, Department of Computer Science, Columbia University, USA, December 1998.

[RZ00a] K. Ross and K. Zaman. Optimizing selections over datacubes. In *Proceedings of the IEEE International Conference on Scientific and*

Statistical Database Management, Berlin, July 2000. IEEE Computer Society.

[RZ00b] K. Ross and K. Zaman. Serving datacube tuples from main memory. In *Proceedings of the IEEE International Conference on Scientific and Statistical Database Management*, Berlin, July 2000. IEEE Computer Society.

[SAM98] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In *Proceedings of the Conference on Extending Database Technology, (EDBT-98)*, 1998.

[Sar99] S. Sarawagi. Explaining differences in multidimensional aggregates. In *Proceedings of the 25th International Conference on Very Large Databases*, Edinburgh, Scotland, 1999.

[SDN98] A. Shukla, P. Deshpande, and J. Naughton. Materialized view selection for multidimensional datasets. In *Proceedings of the 24th International Conference on Very Large Databases*, New York, August 1998.

[SDN00] A. Shukla, P. Deshpande, and J. Naughton. Materialized view selection for multi-cube data models. In *Proceedings of the Conference on Extending Database Technology, (EDBT-00)*, 2000.

[SDNR96] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proceedings of the 22nd International Conference on Very Large Databases*, Bombay, India, 1996.

[SLCJ98] J. Smith, C. Li, V. Castelli, and A. Jhingran. Dynamic assembly of views in data cubes. In *Seventeenth ACM Symposium on Principles of Database Systems*, 1998.

[Smi56] W. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[SQL] Amendment 1 to iso/iec 9075-1:1999, on-line analytical processing (SQL/OLAP). Available from ftp://jerry.ece.umassd.edu/.

[VW99] J. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the 1999 ACM SIGMOD Conference on Management of Data*, Philadelphia, 1999. Association for Computing Machinery.

[ZCL+00] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *Proceedings of the 2000 ACM SIGMOD Conference on Management of Data*, Dallas, 2000. Association for Computing Machinery.

[ZDN97] Y. Zhou, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD Conference on Management of Data*, pages 159–170, Tucson, Arizona, May 1997. Association for Computing Machinery.