Properties and Applications of Diamond Cubes

by

Hazel Jane Webb

Bachelor of Science in Data Analysis, UNB, 2001
Master of Computer Science, UNB, 2005

A Dissertation Submitted in Partial Fulfilment of the Requirements for the Degree of

Doctor of Philosophy

in the Graduate Academic Unit of Computer Science

This dissertation is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

August 2010

Abstract

Queries that constrain multiple dimensions simultaneously are difficult to express, in both Structured Query Language (SQL) and multidimensional languages. Moreover, they can be inefficient. We introduce the diamond cube operator to facilitate the expression and computation of one such important class of multidimensional query, and we define the resulting cube to be a diamond.

The diamond cube operator identifies an important cube such that all remaining attributes are strongly related. For example, suppose a company wants to close shops and terminate product lines, whilst meeting some profitability constraints *simultaneously* over both products and shops. The diamond operator would identify the maximal set of products and shops that satisfy those constraints.

We determine the complexity of computing diamonds and investigate how pre-computed materialised views can speed execution. Views are defined by the parameter k associated with each dimension in every data cube. We prove that there is only one $k_1, k_2, \ldots, k_d$-diamond for a given cube. By varying the k_i's we get a collection of diamonds for a cube and these diamonds form a lattice.

We also determine the complexity of discovering the most-constrained diamond that has a non-empty solution. By executing binary search over theoretically-derived bounds, we compute such a diamond efficiently.

Diamonds are defined over data cubes where the measures all have the same sign. We prove that the more general case—computing a diamond where measures include both positive and negative values—is NP-hard.

Finding dense subcubes in large data is a difficult problem. We investigate the role that diamonds play in the solution of three NP-hard problems that seek dense subcubes: Largest

Perfect Cube, Densest Cube with Limited Dimensions and Heaviest Cube with Limited Dimensions.

We are interested in processing large data sets. We validated our algorithms on a variety of freely-available and synthetically-generated data cubes, whose dimensionality range from three to twenty-seven. Most of the cubes contain more than a million facts, and the largest has more than 986 million facts. We show that our custom implementation is more than twenty-five times faster, on a large data set, than popular database engines.

Dedication

I dedicate this dissertation to my family, whose love and support were invaluable. Most of all, this is for my best friend and greatest supporter, my husband Phillip. Without his faith in me during times of self-doubt, I would never have completed this work.

Acknowledgements

Obviously, this work was not done in isolation, and it is with pleasure and gratitude that I acknowledge the support of many individuals. I have, first and foremost, to thank my supervisors, Dr. Owen Kaser and Dr. Daniel Lemire, for their mentoring, guidance and encouragement. I am particularly grateful for their patience when I took a long time to understand a particular concept, or stubbornly went off on a tangent that they felt was fruitless. Thanks to them this dissertation "tells a story" in a precise, and I hope, interesting way. I hope that I will be able to provide the same kind of support and guidance to future graduate students.

I must thank the members of the examining board; in particular Dr Tim Alderson for his suggestions of how to simplify some of the proofs and Dr Andrew Rau-Chaplin for his kind comments and suggestions for improvements.

Special thanks must also go to Dr Ruth Shaw, my MCS supervisor. Her positive attitude and confidence in my abilities encouraged me to progress. She is a great role model and mentor to all female students.

Every journey begins with a single step and this journey would never have begun if I had not taken a course in Discrete Mathematics from Dr Larry Garey, now professor emeritus at UNB Saint John. He introduced me to interesting and exciting topics and made me realise that studying Computer Science was for me. Thankyou, Larry, for inspiring me and many generations of other students.

Table of Contents

List of Tables

List of Figures

Notation

$\leq_\diamond$	the $k_1, k_2, \ldots, k_d$-diamond is less than or equal to ($\leq_\diamond$) the $k'_1, k'_2, \ldots, k'_d$-diamond if and only if $k_1 \leq k'_1, k_2 \leq k'_2, \ldots, k_d \leq k'_d$	33
C	a data cube	23
$\kappa(C)$	maximum carats in C	36
σ	aggregator such as COUNT or SUM	25
d	the number of dimensions	25
k_i	a cube has k_i carats over dimension i if for every slice x_i of dimension i $\sigma(x_i) \geq k_i$	28
n_i	the number of distinct attribute values in dimension D_i	23
$\|\mathbf{C}\|$	$\|C\| = \sum_j C_{1,j}$, the number of allocated cells in cube C	35

Chapter 1

Introduction

Queries that constrain multiple dimensions simultaneously are difficult to express and compute efficiently in both Structured Query Language (SQL) and multidimensional languages. We introduce the diamond cube operator to facilitate the expression of one such class of multidimensional query. For example, we compute the (400 000,100 000)-diamond to solve the following diamond query from the business analysis domain.

Example 1.1. *A company wants to close shops and terminate product lines simultaneously. The CEO wants a maximal set of products and shops such that each shop would have sales greater than \$400 000, and each product would bring revenues of at least \$100 000—assuming we terminate all other shops and product lines.*

Diamonds are intrinsically multidimensional and, because of the interaction between dimensions, the computation of diamond cubes is challenging. We determine the complexity of computing diamonds and investigate how pre-computed materialised views can speed execution. We also determine the complexity of discovering the most-constrained diamond that has a non-empty solution. By executing binary search over theoretically-derived bounds we can compute such a diamond efficiently.

Diamonds are defined over data cubes where the measures all have the same sign. We prove that the more general case—computing a diamond where measures include both positive and negative values—is NP-hard.

We present diamond cube experiments on large data sets of more than 100 million rows. Our custom implementation generates diamonds more than twenty-five times faster, on a large data set, than popular database engines.

In Chapter 2, background and related work provide a context for the proposed diamond cube operator. The diamond cube is formally defined in Chapter 3 and we show that diamonds are nested and there is a unique diamond for any integer value $k \geq 0$. In Chapter 4, we discuss the complexity of various algorithms designed to compute diamonds. In Chapter 5, we present the results from our experimentation with both real and systematically-generated synthetic data sets. In Chapter 6 we investigate the role that diamonds might play in the solution of three NP-hard problems:

- Largest Perfect Cube
- Densest Cube with Limited Dimensions
- Heaviest Cube with Limited Dimensions

A summary of our contribution to multidimensional data analysis is presented in Chapter 7 together with several potential avenues for future work.

1.1 Motivation

The desire to explore and discover new information from within the vast quantities of data available in disparate fields has motivated research in many areas: On-Line Analytical Processing (OLAP) [Morf 07], databases [Hale 01,Ilya 08] and data mining [Chak 06,Stav 07] amongst others. The motivation for this dissertation is a desire to find a time-efficient implementation for a particular type of multidimensional query: one that can be executed simultaneously against each dimension of a large data set. For example, Hirsch [Hirs 05] introduced a measure—the h-index—to characterise the scientific output of a researcher.

The h-index for a given author is the maximum h such that she has h papers each cited at least h times. The h-index provides a balanced view of an author's work in that it neither rewards authors with large publication lists of papers that no-one cites, nor authors with only one or two widely-cited papers. Calculating the h-index of a researcher is a two-dimensional example of the kind of query in which we are interested. This type of query is difficult to express in both Structured Query Language (SQL) and multidimensional query languages, such as MultiDimensional eXpressions (MDX). Similarly, sub-sampling queries as discussed in the next section are also difficult to express. We, therefore, introduce the diamond cube operator to fill that void.

1.1.1 Sub-sampling

Dealing with large data is challenging. One way to meet this challenge is to choose a sub-sample—select a subset of the data—with desirable properties such as representativity, conciseness, or homogeneity. In signal and image processing, software sub-samples data [Poli 99] for visualisation, compression, or analysis purposes: commonly, images are cropped to focus the attention on a particular segment. In databases, researchers have proposed similar sub-sampling techniques [Babc 03, Gant 00], including iceberg queries [Fang 98, Pei 05, Xin 03] and top-k queries [Loh 02a, Loh 02b]. Such reduced representations are sometimes of critical importance to get good on-line performance in Business Intelligence (BI) applications [Aoui 09b, Fang 98]. Even when performance is not an issue, browsing and visualising the data frequently benefit from reduced views [Ben 06a].

Often, business analysts are interested in distinguishing elements that are most crucial to their business, such as the k products jointly responsible for 50% of all sales, from the *long tail* [Ande 06]—the lesser elements. The computation of icebergs, top-k elements, or heavy-hitters has received much attention [Care 97, Corm 05, Corm 04]. This type of query can be generalised so that interactions between dimensions are allowed. For ex-

Table 1.1: Sales (in million dollars) with a 5,10 sum-diamond shaded: stores need to have sales above \$10 million whereas product lines need sales above \$5 million.

	Chicago	Montreal	Miami	Paris	Berlin	Totals
TV	3.4	0.9	0.1	0.9	2.0	7.3
Camcorder	0.1	1.4	3.1	2.3	2.1	9.0
Phone	0.2	6.4	2.1	3.5	0.1	12.3
Camera	0.4	2.7	5.3	4.6	3.5	16.5
Game console	3.2	0.3	0.3	2.1	1.5	7.4
DVD player	0.2	0.5	0.5	2.2	2.3	5.7
Totals	7.5	12.2	11.4	15.6	11.5	58.2

ample, a business analyst might want to compute the smallest set of stores and business hours jointly responsible for over 80% of the sales. In this new setting, the heads and tails of the distributions must be described using a multidimensional language; computationally, the queries become significantly more difficult. Hence, analysts often process dimensions one at a time: perhaps they would focus first on the most profitable business hours, and then aggregate sales per store, or perhaps they would find the most profitable stores and aggregate sales per hour. This dissertation proposes a general model, of which the uni-dimensional analysis is a special case, that has acceptable computational costs and a theoretical foundation. In the two-dimensional case, the proposal is a generalisation of ITERATIVE PRUNING [Kuma 99], a graph-trawling approach used to analyse social networks.

To illustrate the proposal in the BI context, consider the following example. Table 1.1 represents the sales of different items in different locations. Typical iceberg queries might be requests for stores having sales of at least 10 million dollars or product lines with sales of at least 5 million dollars. However, what if the analyst wants to apply both thresholds simultaneously? He or she might contemplate closing both some stores and some product lines. In our example, applying the constraint on stores would close Chicago, whereas applying the constraint on product lines would not terminate any product line. However, once the shop in Chicago is closed, we see that the product line TV must be terminated which causes the closure of the Berlin store and the termination of two new product lines (Game console and DVD player). This multidimensional pruning query selects a subset of attribute values from each dimension that are simultaneously important. For all the attribute values remaining in the diamond, every column (slice) in the location dimension sums to at least 10 and every row (slice) in the product dimension sums to at least 5. There are many other similar applications:

- In Web traffic analysis, one could seek predominant traffic sources linking to pages benefiting substantially from these sources.
- A corporation could be provided with a list of services bringing in substantial revenues from large consumers of these same services.
- A list could be generated of all movies highly popular among individuals watching many of these movies.
- One could find products with sales above 10 000 units during months where 100 000 units of these products were sold, to customers who bought at least 5 000 such units during the selected months.
- The ITERATIVE PRUNING algorithm could be expanded so that it includes geography or textual content in addition to the inbound/outbound link information.

The proposed operation is a *diamond dice* [Webb 07], which produces a *diamond*, as formally defined in Chapter 3. The removal of attributes from Table 1.1 in this example illustrates an algorithm for computing the diamond, which is formally discussed in Chapter 4.

Other approaches that seek important attribute values, e.g. the Skyline operator [Borz 01, Mors 07], Dominant Relationship Analysis [Li 06], and Top-k dominating queries [Yiu 07], require dimension attribute values to be ordered, e.g. distance between a hotel and a conference venue, so that data points can be ordered. The proposed operator requires no such ordering.

1.2 Contribution

This dissertation establishes properties of the diamond cube together with a non-trivial theory. In every $d-$dimensional data cube whose measure values are all non-negative, given a set of positive integers $\{k_1, k_2, \ldots k_d\}$, there exists a unique maximal subcube where every slice of dimension i has sum at least k_i. For example, we could choose $k_1 = 5$ and $k_2 = 10$ for the two-dimensional cube of Table 1.1. The diamond cube operator extracts this structure. We prove that diamonds are not only unique but they are nested. By varying the k_i's we get a collection of diamonds for a cube and these diamonds form a lattice. A lattice is a partially ordered set in which all finite subsets have a least upper bound and greatest lower bound. For example, Table 1.1 contains the 5×7, 5×8 and 6×10 diamonds as illustrated in Figure 1.1.

Algorithms to extract diamonds have been designed, implemented and tested on real data sets as well as on artificially constructed data with predefined distributions. Our hand-crafted implementation generates diamonds more than fifty times faster than using generic database queries on data cubes containing in excess of forty million facts.

Finding dense subcubes in large data is a difficult problem [Lee 06, Barb 01, Ben 06a]. We investigate the role that diamond cubes might play in the solution of three NP-hard problems that seek dense subcubes. We provide evidence that the diamond operator is a sensible heuristic for two of them and can provide insight into a potential solution for the third.

	Chicago	Montreal	Miami	Paris	Berlin	Totals
TV	3.4	0.9	0.1	0.9	2.0	7.3
Camcorder	0.1	1.4	3.1	2.3	2.1	9.0
Phone	0.2	6.4	2.1	3.5	0.1	12.3
Camera	0.4	2.7	5.3	4.6	3.5	16.5
Game console	3.2	0.3	0.3	2.1	1.5	7.4
DVD player	0.2	0.5	0.5	2.2	2.3	5.7
Totals	7.5	12.2	11.4	15.6	11.5	58.2

(a) 5×7 diamond (original cube)

	Montreal	Miami	Paris	Berlin	Totals
Camcorder	1.4	3.1	2.3	2.1	8.9
Phone	6.4	2.1	3.5	0.1	12.1
Camera	2.7	5.3	4.6	3.5	16.1
DVD Player	0.5	0.5	2.2	2.3	5.5
Totals	11	11	12.6	8	42.6

(b) 5×8 diamond

	Montreal	Miami	Paris	Totals
Camcorder	1.4	3.1	2.3	6.8
Phone	6.4	2.1	3.5	12.0
Camera	2.7	5.3	4.6	12.6
Totals	10.5	10.5	10.4	31.4

(c) 6×10 diamond

Figure 1.1: Diamonds form a lattice.

Chapter 2

Background and Related Work

In this chapter we present the two major paradigms for storing and extracting information from large data—the relational and multidimensional models. For both the relational and multidimensional database environments, we provide examples of how a single query can be applied to multiple dimensions simultaneously.

Diamonds are related to other work in the following ways:

- generalisation—trawling the Web for cyber-communities
- sub-sample extraction—skyline and top-k queries and Formal Concept Analysis
- application—tag clouds and dense sub-cubes.

2.1 The Relational Model

Since its introduction by E. F. Codd in June 1970 [Codd 70], the relational model has become the standard for storing transactional data. It is implemented in many commercial Relational Database Management Systems (RDBMSs) such as Oracle Database, Microsoft SQLServer and MySQL. The base operators of relational algebra and their symbols are given in Figure 2.1. Although the natural join operator appears in Codd's original paper [Codd 70], this operator can also be expressed as a Cartesian product followed by a

σ	selection
$\prod$	projection
$\times$	Cartesian product
$\cup$	set union
$-$	set difference
$\cap$	set intersection
$\bowtie$	natural join

Figure 2.1: Relational algebra operators.

selection. The operations of relational algebra, except selection, projection and Cartesian product, require that the relations be **union-compatible**, i.e. having the same number and type of attributes; the attributes must be presented in the same order in each relation.

Structured English Query Language [Cham 74], which later became Structured Query Language (SQL), was developed at IBM as the language to query relational data. SQL provides a structured set of English templates with which to obtain information from tables. Since its inception, SQL has undergone many changes and refinements. For instance, the language based on the original algebra did not include aggregate functions, which were first proposed by Klug [Klug 82]. Agrawal [Agra 87] introduced a recursion operator to the relational model. He suggested that a database system that has special algorithms and data structures to directly implement an operator may outperform a system that relies on an iterative procedure to execute the same function. The current standard [ISO 08] makes SQL Turing complete by specifying the syntax and semantics for, among others, loops, decisions and recursive queries—making it much more powerful than the original relational model. However, not all commercial vendors support all the SQL language features and they sometimes include their own proprietary extensions.

Some queries that can be expressed simply in plain English are difficult to express in SQL, despite the original design goals to provide a simple English-like interface for non-programmers to interact with the database. Suppose we have a chain of department stores and record information about our sales, inventory and employees in a typical RDBMS. The relations Sale, Employee and Product would be linked with appropriate use of foreign

prodID	type	...
1	camcorder	⋮
2	ipod	
3	camera	
4	cell phone	

(a) Product relation

prodID	salesPersonID
1	a
1	b
2	c
2	d
3	a
3	b
3	d
4	c
4	e

(b) Sale relation used for Queries 2.1, 2.2 and 2.3.

empID	name	...
a	Paul	⋮
b	Carol	
c	Phillip	
d	Julie	
e	Elaine	

(c) Employee Relation

Figure 2.2: Example relational database. Sale relation has foreign keys: salesPersonID → empID of the Employee table, and prodID → prodID of the Product table.

keys. (See Figure 2.2.) We could ask straightforward queries like: *What were the total sales for employee Julie last month?* as in Query 2.1.

```
SELECT Sum(s.sales)
FROM Sale s, Employee e
WHERE s.date BETWEEN '2009-07-01' AND '2009-08-01'
     AND e.name = 'Julie'
     AND e.empID = s.salesPersonID;
```

Query 2.1: Simple SELECT . . . FROM . . . WHERE . . . example.

If, however, we wish to compare all sales persons over a particular time period and sales of a particular product, the query becomes more complex, as shown in Query 2.2.

When we are interested in mining information **simultaneously** on two attributes, the query is much more difficult to express in standard SQL. Suppose our sales data has the form of Table 2.2b and the required query is: *Find a group of sales persons S and a group of products P, such that each sales person in S sold at least n of the products in P; moreover, each product in P had a least n sales people selling it. We want n to be maximum, and we want both S and P to be as large as possible.* We are interested in finding the sales people who have been successful in selling the most varied products and at the same time

```
SELECT e.name, Sum(s.sales)
FROM Sale s, Employee e, Product p
WHERE s.date BETWEEN '2009-07-01' AND '2009-08-01'
     AND p.type = 'ipod'
     AND e.empID = s.salesPersonID
     AND s.prodID= p.prodID
GROUP BY name;
```

Query 2.2: Simple GROUP BY example.

interested in the most popular products sold by a variety of sales personnel. The solution to this query is two sales persons, Paul and Carol, sold two products, camcorder and camera.

```
SELECT COUNT(s.salesPersonID) AS numberSalesPersons,e.name,
       COUNT(s.prodID) AS numberProducts, p.type
FROM Sale s, Product p, Employee e
WHERE e.empID = s.salesPersonID AND p.prodID = s.prodID
GROUP BY s.salesPersonID, s.prodID
HAVING numberSalesPersons >= 2 AND numberProducts >= 2;
```

Query 2.3: An attempt to constrain two attributes simultaneously. Unfortunately, only an empty set is returned.

We might try to express this query in SQL using Query 2.3. We constrain the number of sales personnel and products to be greater than two, although we might not have an idea of the total number of sales personnel and products that constitute the "largest" set. In any case, the result from Query 2.3 is an empty set—not what we are looking for.

2.2 The Multidimensional Model

On-line analytical processing (OLAP) is a database acceleration technique used for deductive analysis, which the OLAP Council [OLAP] describes in five key words: fast analysis of shared multidimensional information. It is used in decision support systems to provide answers to "big picture" queries such as

What were our average quarterly sales in 2006 for each region?

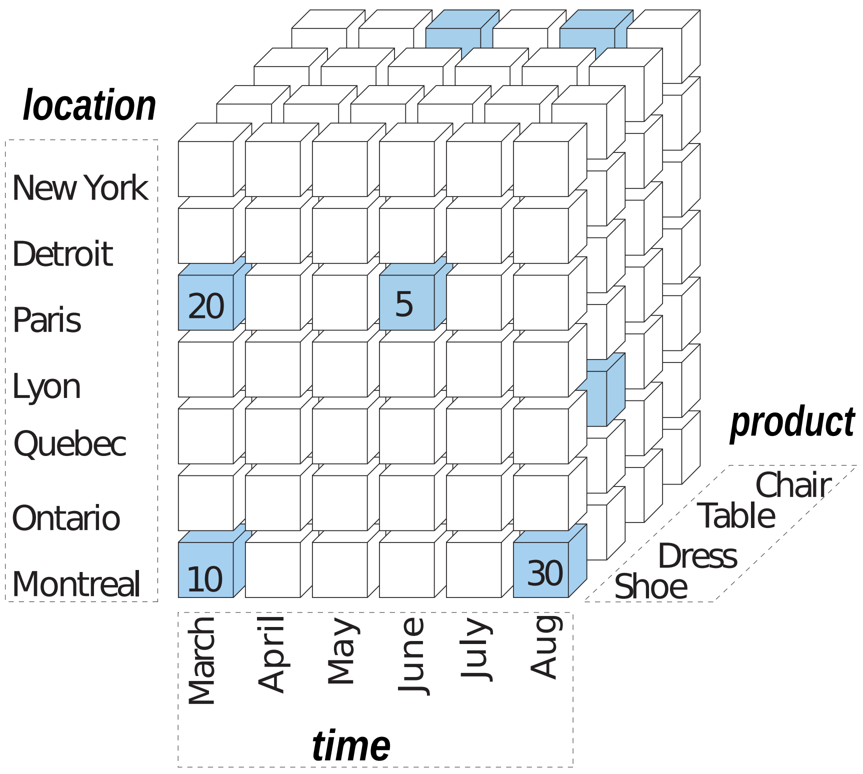


Figure 2.3: Three-dimensional data cube.

Researchers and developers have yet to agree on a single multidimensional model for OLAP [Rizz 06]. Our model is formally described in Section 3.1.

The main objective of OLAP is to have constant-time, or near constant-time, answers for many typical queries. To achieve such acceleration one can create a *cube* [Rumm 70] of data, a map from attribute values in every dimension to a given measure ($v_1 \times v_2 \times \cdots \times v_d \rightarrow M$). Figure 2.3 illustrates such a cube in three dimensions, *time, location* and *product*, built from the RDBMS relations in Section 2.1. The measure stored at the intersection of the axes represents a sales total: for example, there were 10 units of shoes sold in Montreal in March. A business analyst might be interested in seeking trends of sales across products and time—location is not relevant in this query. In this case, the cube could be *rolled-up*, collapsing the dimension hierarchy from three to two. The measure stored at the intersection of the axes represents the total sales of a product in a given month and thus would reflect that 30 units of shoes were sold in March. Similarly, if time was

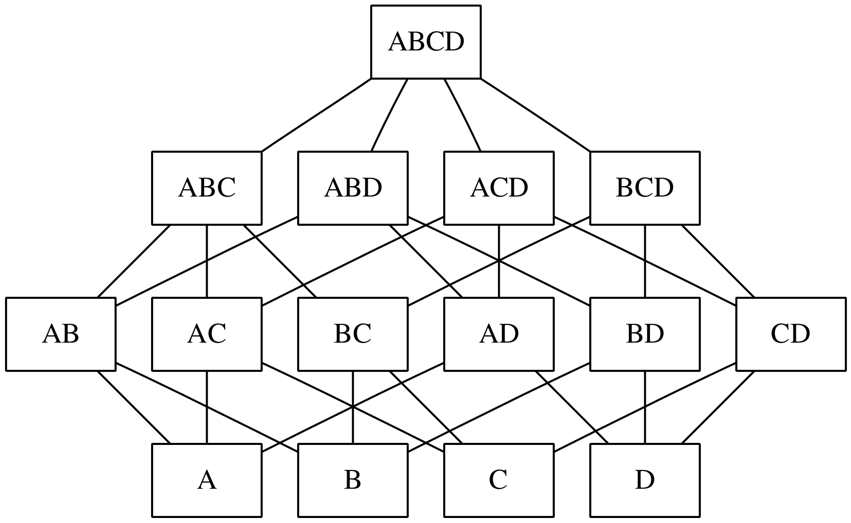


Figure 2.4: Four-dimensional lattice under roll-ups.

not important, but location and product were, one could *roll-up* the cube by aggregating the sums over time, reflecting that 40 units of shoes were sold in Montreal. In this way a lattice of cubes can be generated. Figure 2.4 illustrates a four-dimensional data cube lattice under roll-ups. Data are stored at a coarser granularity in cube ABC than in cube $ABCD$, i.e. the measures do not distinguish different attribute values for dimension D, and a *roll-up* can derive ABC from $ABCD$.

It is possible to generate a diamond cube using a language such as Microsoft's MultiDimensional eXpression language (MDX) for OLAP. We can create subcubes by restricting values on each dimension in turn until no change is observed in the cube. For example using the cube from Figure 2.3, Query 2.4 executes the first iteration to compute a diamond: the query must be repeated until there is no change.

2.3 Trawling the Web for Cyber-communities

A specialisation of the diamond cube is found in Kumar et al.'s work searching for emerging social networks on the Web [Kuma 99]. Our approach is a generalisation of their two-dimensional ITERATIVE PRUNING algorithm. Diamonds are inherently multidimen-

```
create subcube [Sales] as
select {filter([location].children, Measures.[Measure] > 100)}
on 0,
[time].children on 1,
[product].children on 2
from Sales

create subcube [Sales] as
select {filter([time].children, Measures.[Measure] > 100)}
on 1,
[location].children on 0,
[product].children on 2
from Sales

create subcube [Sales] as
select {filter([product].children, Measures.[Measure] > 100)}
on 2,
[time].children on 1,
[location].children on 0
from Sales
```

Query 2.4: Query to compute the $100 \times 100 \times 100$ diamond. All three statements must be repeated in order, until there is no more change.

sional.

Kumar et al. [Kuma 99] model the Web as a directed graph and seek large dense bipartite subgraphs. Recall that a bipartite graph is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V: U and V are independent sets. A bipartite graph is dense if most of the vertices in U and V are connected. Kumar et al. hypothesise that the signature of an emerging Web community contains at least one "core", which is a *complete* bipartite subgraph with at least i vertices from U and j vertices from V. In their model, the vertices in U and V are Web pages and the edges are links from U to V. Seeking an (i, j) core is equivalent to seeking a **perfect** two-dimensional diamond cube (all cells are allocated). The iterative pruning algorithm is a specialisation of the basic algorithm we use to seek diamonds: it is restricted to two dimensions and is used as a preprocessing step to prune data that cannot be included in the (i, j) cores.

Although their paper has been widely cited [Redd 01, Yang 05, Ragh 03, Holz 06], it appears that this dissertation is the first to propose a multidimensional extension to their approach and to provide a formal analysis.

2.4 Sub-sampling with Skyline and Top-k Queries

RDBMSs have optimisation routines that are especially tuned to address both basic and more complex SELECT ... FROM ... WHERE ... queries. However, there are some classes of queries that are difficult to express in SQL, or that execute slowly, because suitable algorithms are not available to the underlying query engine. Skyline, top-k and top-k-dominating queries are examples of such queries, ones that seek a representative subsample of the data.

```
SELECT *
FROM Hotels h
WHERE h.city = 'Saint John' AND NOT EXISTS(
      SELECT *
      FROM Hotels h1
      WHERE h1.city = 'Saint John'
      AND h1.distance <= h.distance
      AND h1.price <= h.price
      AND( h1.distance < h.distance OR h1.price < h.price));
```

Query 2.5: A nested query that may execute slowly.

2.4.1 Skyline Operator

The Skyline query [Borz 01] seeks a set of points where each point is not "dominated" by some others: a point is included in the skyline if it is as good or better in all dimensions and better in at least one dimension. Attributes, e.g. distance or cost, must be ordered. Therefore, indexes can be beneficial. Query 2.5 is an example that seeks the set of dominating hotels in Saint John, i.e. they are cheaper or closer to some landmark (possibly a conference venue) than all others. Börzsönyi et al. [Borz 01] present algorithms to implement the skyline operator and the results of their execution on synthetic data. The overall performance of the skyline operator was consistently good across all data sets presented in their study.

Since the skyline was proposed as an extension to SQL, the authors discussed its inter-operation with other operators, e.g. join. Typically, a skyline operator would be applied after any joins or GROUP_BY operations. It can be pushed through a join—applied before the join is computed—if it is non-reductive as defined by Carey [Care 97]. Informally, such a join is between an attribute x, which cannot be null, and at least one element y in the joining relation for which $x = y$ holds. The skyline operation can also be pushed *into* a join. Consider the example query that asks for employees with high salary and who work in departments with low budget [Borz 01]. The employees with maximum salary from each department can be computed first (part of the skyline), then joined with the departments and the remaining part of the skyline—low budget—computed. Similarly, the computation of a diamond can be pushed into a join. Suppose we are seeking the employees who have made more than \$100 000 in sales, of high value products, and who have sold products whose sales total more than \$25 000 000. For the relation `Sale(empID,prodID,total)`, we can compute the diamond cube as follows:

1. Select all employees who have more than \$100 000 in sales. Save the result in `T1(empID, sumOfSales)`.
2. Select all products whose sales total more than \$25 000 000. Save the result in `T2(prodID, sumOfProductSales)`
3. Compute the join $\sigma_{\text{empId,prodId,sumOfSales,sumOfProductSales}}(\text{Sale} \bowtie \text{T1} \bowtie \text{T2})$.
4. Finalise the diamond computation

Executing Steps 1 and 2 before the join would result in fewer tuples over which to finalise the diamond computation than if the join was executed first. The employees and products discarded before the join do not meet the constraint of sales greater than \$100 000 or \$250 000, respectively, and thus would never be part of a solution to this query.

As with the skyline query, a diamond dice can be partially executed in SQL. It is, however, unwieldy to express and it runs slowly. (See Section 5.7.)

2.4.2 Top-k

Another query, closely related to the skyline query, is that of finding the "top-k" data points. Donjerkovic and Ramakrishnan [Donj 99] frame the top-k problem as one of finding the best answers quickly rather than all the answers. Their method is applicable to unsorted and unindexed attributes. (Otherwise the index would be used to provide an exact result.) The top-k query on attribute χ can be rephrased as: $\sigma_{\chi > x}$, where x is a cutoff parameter that is expected to return k items. The authors' example explains this: "List the top 10 paid employees in the sales department", which can be translated to "List the employees from sales department whose salary is greater than x".

System statistics are used to estimate x such that the query returns more than k items. If fewer than k items are returned the query must be restarted; x is an unknown, but it is sufficient that x is chosen so that more than k items are in the result set. The authors show that using their probabilistic approach is much more efficient than the simplest execution plan: execute the query, sort the result and discard all but k items.

Top-k dominating queries [Yiu 07] combine skyline and top-k to present a set of points that dominate others in all dimensions, but keep the result set to a user-defined size. The data is indexed with a tree where each node stores the bounding box delimiting its area, together with the number of points that fall within the bounding box. The number of other points dominated are computed with the top-k retained in the solution. The top-k query may return data points that are not in the skyline, but nevertheless, do dominate. The result can be obtained without any special domain knowledge or the need for the user to supply a ranking function, or weights for the dimensions. The objectives of skyline and top-k queries and diamond dicing are orthogonal. The diamond cube comprises the largest set of attributes that satisfy some multidimensional constraints, whereas top-k and skyline queries seek to reduce the number of data points in the result.

2.4.3 Formal Concept Analysis

In Formal Concept Analysis [Will 89] [Godi 95] a Galois (concept) lattice is built from a binary relation. It is used in machine learning to identify conceptual structures among data sets. A formal concept is a set of objects—extent—together with the set of attributes—intent—that describe each of the objects. For example, a set of documents and the set of search terms those documents match, form a concept. Given the data in Table 2.5a, the smallest concept including document 1 is the one with documents {1, 2} and search terms {A,B,C,D}. Concepts can be partially ordered by inclusion and thus can be represented by a lattice as in Figure 2.5b

Galois lattices are related to diamond cubes: a perfect 3×3 diamond is described in the central element of the lattice of Figure 2.5b. Formal Concept Analysis is typically restricted to two dimensions although Cerf et al. [Cerf 09] generalise formal concepts by presenting an algorithm that is applied to more than two dimensions. Their definition of a closed $n-$set—a formal concept in more than two dimensions—states that each element is related to all others in the set and no other element can be added to this set without breaking the first condition. It is the equivalent of finding a perfect diamond in n dimensions. For example, given the data of Table 2.1 $\{(\alpha, \gamma)(1, 2)(A, B)\}$ is an example of a closed 3-set and also a perfect 4,4,4-diamond.

2.5 Applications for Diamond Cubes

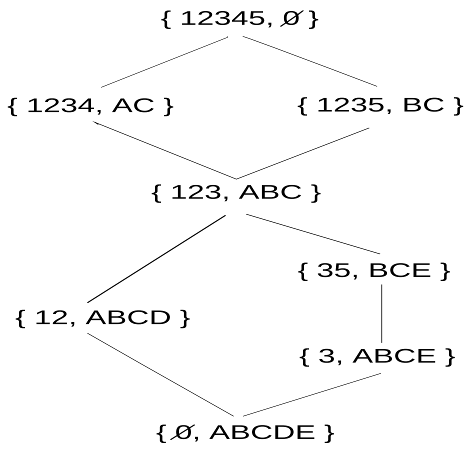
Diamonds have the potential to be useful in different areas and we present two specific applications as examples. First, we suggest they could be used when generating tag clouds—

		dimension 3			dimension 3			dimension 3		
		A	B	C	A	B	C	A	B	C
dimension 2	1	1	1	1	1	1	1	1	1	
	2	1	1		1	1		1	1	
	3		1				1	1		1
	4			1	1		1	1	1	1
		α			β			γ		
		dimension 1								

Table 2.1: A 3-dimensional relation with closed 3-set $\{(\alpha, \gamma)(1, 2)(A, B)\}$.

		Search Terms				
		A	B	C	D	E
Documents	1	1	1	1	1	0
	2	1	1	1	1	0
	3	1	1	1	0	1
	4	1	0	1	0	0
	5	0	1	1	0	1

(a) A 3×3 diamond is embedded in this binary relation.



(b) Galois lattice. Each element (concept) in the lattice is defined by its extent and intent.

Figure 2.5: Documents and search terms in an information retrieval system and the corresponding Galois lattice.

visual indexes that typically describe the content of a Web site. Tag clouds are defined and discussed in the next section. Second, one of the original motivations for this work was to identify a single large, dense subcube in any given data set. Once found, a dense region can be exploited to facilitate visualisation of the data or be used for further statistical analysis. Related work in this area is discussed in Section 2.5.2

2.5.1 Visualisation

It is difficult to visualise large multidimensional data sets [Mani 05, Ben 06a, Tech 05]. Aouiche et al. proposed tag clouds as an alternative [Aoui 09a]. Tag clouds are used

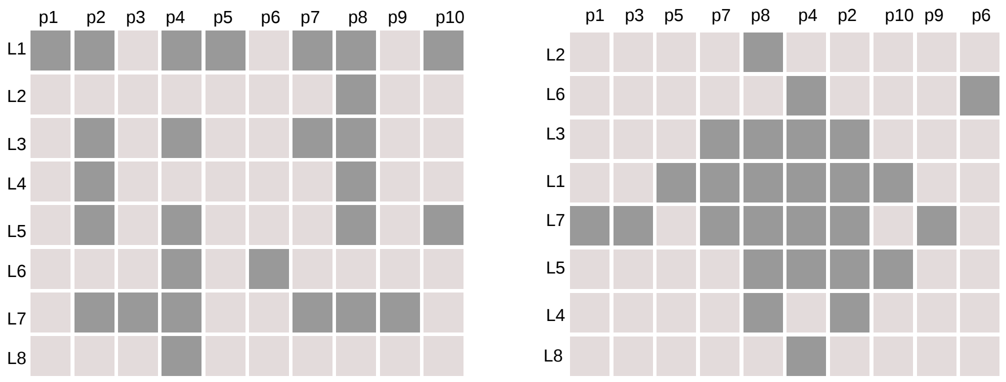
Figure 2.6: A tag cloud: more important words are expressed in larger fonts.

	Chicago	Montreal	Miami	Victoria	Saint John
Dog	4	2	2	1	1
Cat	2	4	1	6	1
Goldfish	10	3	4	1	3
Rabbit	3	11	1	2	1
Snake	1	1	1	1	9

Figure 2.7: Tags and their corresponding weights.

on Web sites to communicate the relative importance of facts or topics discussed on a particular page. A tag is defined [Aoui 09a] as a term or phrase describing an object with corresponding non-negative weights determining its relative importance, and it comprises the triplet (term, object, weight). Figure 2.6 illustrates a tag cloud [Kase 07].

For practical reasons, only the top values in each dimension are presented in a tag cloud. They can be selected by a top-k query on each dimension. However, if we seek significant values interacting together in a multidimensional way, then we can use the diamond operator. This would ensure that if the user slices on a value, some of the "top" values from other dimensions remain relevant. Consider Figure 2.7 and its corresponding tag cloud. Taking the top-2 from each dimension the cloud comprises goldfish, rabbit, Chicago and Montreal. If we slice on Chicago using a top-2 query, then *goldfish* and *dog* would be part of the cloud, which may very well confuse the user as *dog* was not part of the originally presented tags. If however, we slice on Montreal using a top-2 query we have *cat* and *rabbit*, also potentially confusing. Conversely, a 13×13 diamond retains the shaded items no matter which slice is chosen.


p1 p2 p3 p4 p5 p6 p7 p8 p9 p10
L1 L2 L3 L4 L5 L6 L7 L8
p1 p3 p5 p7 p8 p4 p2 p10 p9 p6
L2 L6 L3 L1 L7 L5 L4 L8

(a) Before reorganisation (b) After reorganisation

Figure 2.8: The allocated cells are in dark grey: they are moved together by the reorganisation.

2.5.2 Dense Subcubes

One motivation for the diamond dice is to find a single, non-trivial dense region within a data cube. If the region has sufficient density, then hybrid storage becomes possible: the dense region can be stored as a multidimensional array, whereas the tuples outside of the dense region can be stored sparsely, e.g. as a list of locations. Another benefit of partitioning dense and sparse regions is that it enables visual representation of data to be enhanced. The dense regions can be represented close together on a single screen. Ben Messaoud et al. [Ben 06a] employ Multiple Correspondence Analysis (MCA) to reorganise the cube so that dense subcubes are moved together for easy visualisation: A complete disjunctive table representing the entire cube is built and then MCA is applied to obtain factorial axes. Eigenvalues for each axis are computed and the user chooses a number of axes to visualise. We repeat the example from Ben Messaoud et al.'s paper in Figure 2.8. As with diamond dicing, this approach measures the contribution of a set of cells to the overall cube. No timings were given in their paper, so it is difficult to compare the efficiency of MCA and diamond dicing, although the outputs are quite similar.

Some researchers seek multiple dense regions rather than a single one: to more efficiently compute range sum queries [Lee 06] or as a preprocessing step before further statistical analysis [Barb 01]. Lee uses a histogram-flattening technique by computing three relative density measures for each attribute in each dimension. Dense regions in each dimension are then used to build multi-dimensional dense regions. The candidate results are then refined iteratively. Experiments conducted on synthetic data seeded with clusters, show that this method achieves good space reduction. However, there is no evidence of experimentation on real data or the time to compute their structure.

Barbará and Wu [Barb 01] seek to identify dense regions that can be further analysed by statistical methods. They shuffle attributes within dimensions based on a similarity measure. They do not mention whether rearranging attributes in one dimension has an adverse affect on the positioning of attributes in another dimension. Intuitively, one would expect there to be some issues with this process. The example provided in their paper identified four dense regions and showed how their algorithm would gather them together. The diamond dice applied to their example would find the largest dense region. If one then applied a diamond dice to the pruned area iteratively, all the regions identified by Barbará's method would also be identified by the diamond dice for this particular example.

Chapter 3

Properties of Diamond Cubes

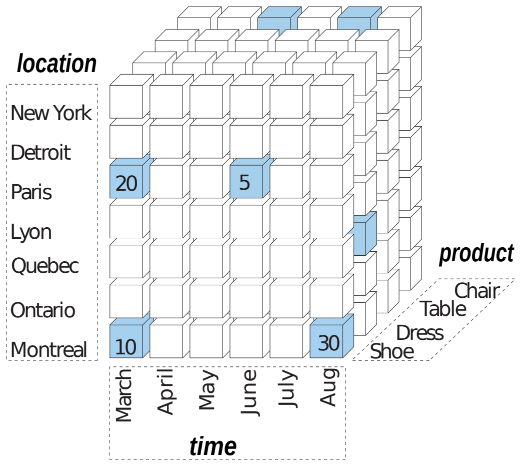
In this chapter a formal model of the diamond cube is presented. We show that diamonds are nested, much like Matryoshkas[1], with a smaller diamond existing within a larger diamond. We prove a uniqueness property for diamonds and we establish upper and lower bounds on the parameter k for both COUNT and SUM-based diamond cubes. One statistic of a cube C is its carat-number, $\kappa(C)$; its robustness is discussed in Section 3.6. We conclude this chapter by presenting the related NP-complete problem for cubes that contain both positive and negative measures.

3.1 Formal Model

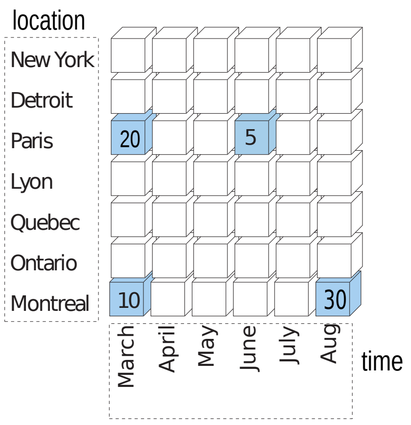
As stated in Chapter 2, researchers and developers have yet to agree on a single multidimensional model [Rizz 06] for OLAP. Our simplified formal model incorporates several widely-accepted definitions for the terms illustrated in Figure 3.1, together with new terms associated specifically with diamonds. For clarity, all terms are defined in the following paragraphs.

A **dimension** D is a set of **attribute values** that defines one axis of a multidimensional data structure. Each dimension D_i has a cardinality n_i, the number of distinct attribute

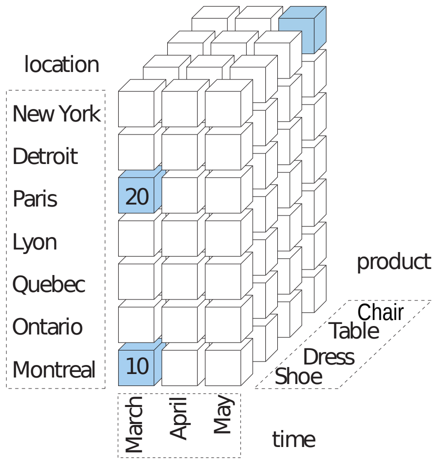
[1]Russian nesting dolls [Roos 09]



(a) 3-D cube



(b) Slice on product shoe



(c) Dice on months March–May

Figure 3.1: OLAP terms: cube, slice and dice.

values in this dimension. Without losing generality, we assume that $n_1 \leq n_2 \leq \ldots \leq n_d$. A dimension can be formed from a single attribute of a database relation and the number of dimensions is denoted by d. In Figure 3.1a the dimensions are time, product and location.

A **cube** is the 2-tuple (D, f) which is the set of dimensions $\{D_1, D_2, \ldots, D_d\}$ together with a total function (f) which maps tuples in $D_1 \times D_2 \times \cdots \times D_d$ to $\mathbb{R} \cup \{\perp\}$, where $\perp$ represents undefined.

A **cell** of cube C is a 2-tuple $((x_1, x_2, \ldots, x_d) \in D_1 \times D_2 \times \cdots \times D_d, v)$ where v is a measure and $v = f(x_1, x_2, \ldots, x_d)$).

The **measure** may be a value $v \in \mathbb{R}$, in which case we say the cell is an **allocated cell**. Otherwise, the measure is $\perp$ and we say the cell is empty—an **unallocated cell**. For the purposes of this dissertation, a measure is a single value. In more general OLAP applications, a cube may map to several measures. Also, measures may take values other than real-valued numbers—booleans, for example.

A **slice** is the cube $C' = (D', f')$ obtained when a single attribute value is fixed in one dimension of cube $C = (D, f)$. For example, if we slice on dimension three of C with value $v_3 \in D_3$, then we have $f'(x_1, x_2, x_3, \ldots, x_d) = f(x_1, x_2, v_3, x_3, x_4, \ldots, x_{d-1})$ where $x_1 \in D_1,\ x_2 \in D_2$ and for $3 \leq i \leq d-1$, $x_i \in D_{i+1}$, resulting in a cube with $d-1$ dimensions.

A **dice** defines a subcube S by specifying attribute values $\widehat{D}_1, \widehat{D}_2, \ldots, \widehat{D}_d$ where each $\widehat{D}_i \subseteq D_i$. The cells in the slices corresponding to the unspecified attribute values are deallocated—**deallocation** of a cell sets its value to $\perp$. The cube is considered to still have the same set of dimensions, although some may have attribute values whose slices contain only deallocated cells.

An **aggregator** is a function, σ, that assigns a real number to a slice of a cube. We choose aggregators such that they are commutative; their return value is not affected by any change in order of the values of the slice. For example, SUM is an aggregator: SUM(slice$_i$) = $v_1 + v_2 + \cdots + v_m$ where m is the number of allocated cells in slice$_i$ and the v_i's are the measures. A slice S' is a subslice of slice S (written $S' \sqsubset S$) if every allocated cell in S' is also an allocated cell in S and they have the same measures. An aggregator σ is **monotonically non-decreasing** if $S' \sqsubset S$ implies $\sigma(S') \leq \sigma(S)$. Similarly, σ is **monotonically non-increasing** if $S' \sqsubset S$ implies $\sigma(S') \geq \sigma(S)$. Monotonically non-decreasing operators include COUNT, MAX and SUM over non-negative measures. Monotonically non-increasing operators include MIN and SUM over non-positive measures. MEAN and MEDIAN are neither monotonically non-increasing, nor non-decreasing functions.

Many OLAP aggregators are distributive, algebraic and linear. The algorithms we used to compute the diamond cube require a linear aggregator. An aggregator σ is *distributive* [Gray 96] if there is a function F such that for all $0 \leq k < n-1$,

$$\sigma(a_0, \ldots, a_k, a_{k+1}, \ldots, a_{n-1}) = F(\sigma(a_0, \ldots, a_k), \sigma(a_{k+1}, \ldots, a_{n-1})).$$

For example, $\text{SUM}(a_0, \ldots, a_k, a_{k+1}, \ldots, a_{n-1}) = \text{SUM}(a_0, \ldots, a_k) + \text{SUM}(a_{k+1}, \ldots, a_{n-1})$. An aggregator σ is *algebraic* if there is an intermediate tuple-valued **distributive** function G from which σ can be computed. An algebraic example is AVERAGE: given the tuple (COUNT, SUM), one can compute AVERAGE by a ratio. In other words, if σ is an algebraic function then there must exist G and F such that

$$G(a_0, \ldots, a_k, a_{k+1}, \ldots, a_{n-1}) = F(G(a_0, \ldots, a_k), G(a_{k+1}, \ldots, a_{n-1})).$$

If G is (COUNT, SUM) then $F((c1, s1), (c2, s2)) = (c1 + c2, s1 + s2)$.

For example, $\text{COUNT,SUM}(a_0, a_1) = F((1, a_0)(1, a_1)) = (1 + 1, a_0 + a_1)$ and generally, $\text{COUNT,SUM}(a_0, \ldots, a_k, a_{k+1}, \ldots, a_{n-1}) = F((k+1, a_0+a_1+\ldots,+a_k), (n-1-k, a_{k+1}+a_{k+2} + \cdots + a_{n-1})) = (n, a_0 + a_1 + \cdots + a_{n-1})$

An algebraic aggregator σ is *linear* [Lemi 08] if the corresponding intermediate query G

movie	reviewer	date	rating
1	1488844	2005-09-06	3
1	822109	2005-05-13	5
1	885013	2005-10-19	4
1	30878	2005-12-26	4
1	823519	2004-05-03	3
1	893988	2005-11-17	3
1	124105	2004-08-05	4
1	1248029	2004-04-22	3

Figure 3.2: Part of the Netflix [Benn 07] fact table (cube). Attributes (dimensions) are movie, reviewer and date. Each row is a fact (allocated cell). The measure is rating.

satisfies

$$G(a_0 + \alpha d_0, \ldots, a_{n-1} + \alpha d_{n-1}) = G(a_0, \ldots, a_{n-1}) + \alpha G(d_0, \ldots, d_{n-1})$$

for all arrays a, d, and constants α. SUM and COUNT and AVERAGE are linear functions; MAX, MIN, ROOT-MEAN-SQUARE and MEDIAN are not linear.

Our formal model maps to the relational model in the following ways: (See Figure 3.2.)

- **cube** corresponds to a fact table: a relation whose attributes comprise a primary key and a single measure.
- **allocated cell** is a fact, i.e. it is a distinct record in a fact table.
- **dimension** is one of the attributes that compose the primary key.

3.2 Diamond Cubes are Unique

Diamonds are defined over data cubes where the measures all have the same sign. Cubes with mixed measures (positive and negative) are discussed in Section 3.7. We define a novel measure for diamonds, the carat, and show that given $k_1, k_2, \ldots k_d$ there is a unique $k_1, k_2, \ldots k_d-$diamond.

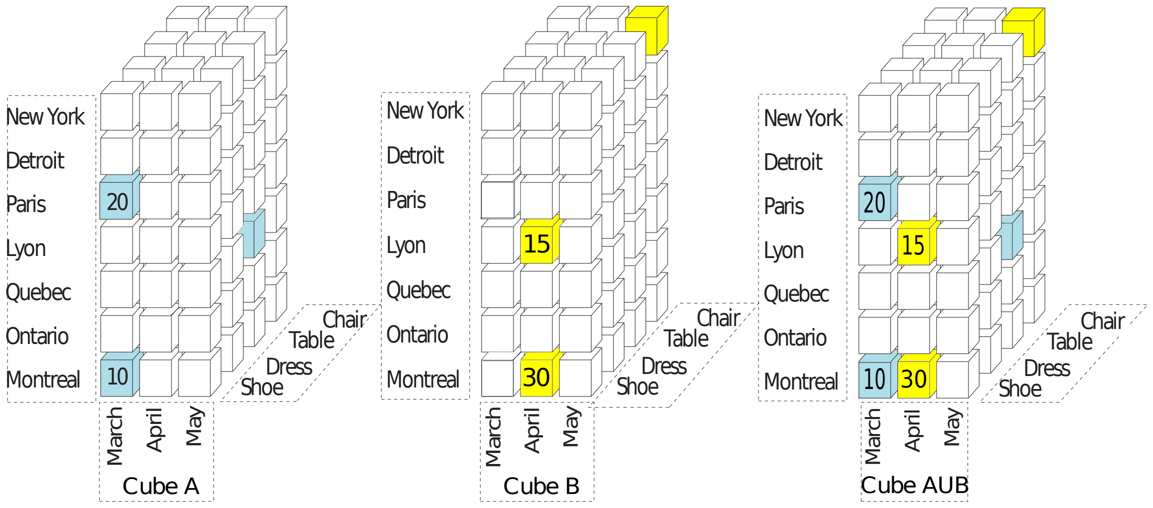


Figure 3.3: Union of two cubes.

Definition 3.1. *Let σ be a linear aggregator, such as* SUM *or* COUNT, *and $k_1, k_2, \ldots, k_d \in \mathbb{R}$. A cube has k_i* ***carats*** [2] *over dimension D_i, if for every non-empty slice x along dimension D_i, $\sigma(x) \geq k_i$, $i \in \{1, 2, \ldots, d\}$. Similarly, a cube has k_i negative carats over dimension D_i, if $\sigma(x) \leq -k_i$ for every non-empty slice. There is no restriction on the number of carats in a cube that contains only empty slices.*

Given cubes A and B and their respective measure functions f_A and f_B, A and B are **compatible** if $f_A(x) \neq \perp$ and $f_B(x) \neq \perp \implies f_A(x) = f_B(x)$.

Definition 3.2. *Let A and B be two compatible cubes with the same dimensions. The union of A and B is a cube denoted $A \cup B$. Formally, given $A = (D, f_A)$ and $B = (D, f_B)$, then $A \cup B = (D, f_{A\cup B})$ where*

$$f_{A\cup B}(x) = \begin{cases} f_A(x) & \textit{if } f_A(x) \neq \perp, \\ f_B(x) & \textit{otherwise} \end{cases}$$

Proposition 3.1. *If the aggregator σ is monotonically non-decreasing, then the union of any two cubes having k_i carats over dimension D_i has k_i carats over dimension D_i as well,*

[2] In actual diamonds, one carat is equal to 200 milligrammes. The price of a diamond rises exponentially with the number of carats [Ency 07]

for $i = \{1, 2, \ldots, d\}$. Similarly, if the aggregator σ is monotonically non-increasing, then the union of any two cubes having k_i negative carats over dimension D_i has k_i negative carats over dimension D_i as well, for $i = \{1, 2, \ldots, d\}$.

Proof. The proof follows trivially from the monotonicity of the aggregator. □

Definition 3.3. *Let A and B be two compatible cubes with the same dimensions. The intersection of two cubes A and B is a cube denoted $A \cap B$ containing the non-empty cells that are common to A and B. Formally, given $A = (D, f_A)$ and $B = (D, f_B)$, then $A \cap B = (D, f_{A\cap B})$ where*

$$f_{A\cap B}(x) = \begin{cases} f_A(x) & \text{if } f_B(x) \neq \perp, \\ \perp & \text{otherwise} \end{cases}$$

Suppose a cube C' having $k_1, k_2, \ldots k_d$ carats over dimensions $D_1, D_2, \ldots, D_d$ is specified by a dice on cube C. Cube C' is **maximal** when it is not contained in a larger cube having $k_1, k_2, \ldots, k_d$ carats over dimensions $D_1, D_2, \ldots, D_d$: the larger cube must, of course, also be specified by a dice on cube C.

As long as σ is monotonically non-decreasing (resp. non-increasing), there is a maximal cube having $k_1, k_2, \ldots, k_d$ carats (resp. negative carats) over dimensions $D_1, D_2, \ldots, D_d$, and we call such a cube a *diamond.* [3] Such a **diamond** is $\bigcup_{A\in\mathcal{A}} A$, where $\mathcal{A}$ is the set of all subcubes (of some starting cube C) having $k_1, k_2, \ldots, k_d$ carats. A diamond cube has $k_1, k_2, \ldots, k_d$-carats if it has k_i carats over dimension D_i. Any reference to a k-carat diamond cube implies $k_1 = k_2 = \ldots = k_d = k$.

We can think of the maximal cube, the diamond cube, as a dice. However, by a slight abuse of our definition, and for ease of exposition, we also consider diamond dicing as the removal of attribute values.

[3]When σ is not monotonically non-decreasing (resp. non-increasing), there may not be a unique diamond

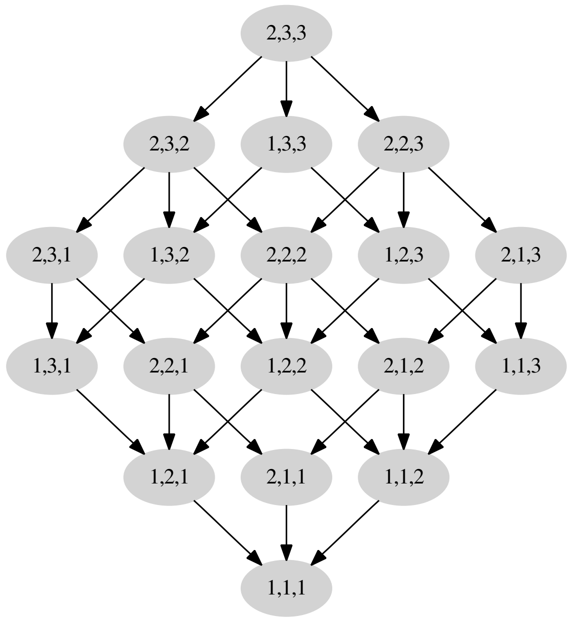


Figure 3.4: Part of the COUNT-based diamond-cube lattice of a $2 \times 2 \times 2$ cube.

3.3 Diamond Cubes Form a Lattice

The following propositions show that diamonds are nested and form a **lattice**. This is helpful because the $x_1, x_2, \ldots, x_d$-carat diamond can be derived from the $y_1, y_2, \ldots, y_d$-carat diamond when $y_i \leq x_i$ for all i. In an implementation where materialising the diamond lattice is considered, an analyst need only choose a subset of elements to store physically, the others can be derived.

Proposition 3.2. *The diamond having k' carats (resp. negative carats) over dimensions $i_1, \ldots, i_d$ is contained in the diamond having k carats (resp. negative carats) over dimensions $i_1, \ldots, i_d$ whenever $k' \geq k$.*

Proof. Let A be the diamond having k carats and B be the diamond having k' carats. By Proposition 3.1, $A \cup B$ has at least k carats, and because A is maximal, $A \cup B = A$; thus, B is contained in A. $\square$

Given an $n_1 \times n_2 \times \cdots \times n_d$ cube C, we can build the lattice of $(k_1, k_2, \ldots, k_d)$-value vectors from the Cartesian product of $(\{1 \text{ to } \frac{\prod_j n_j}{n_1}\}, \{1 \text{ to } \frac{\prod_j n_j}{n_2}\}, \ldots \{1 \text{ to } \frac{\prod_j n_j}{n_d}\})$ $j \in$

$\{1, 2, \ldots, d\}$. The ordering of k-value-vectors is such that $(a_1, a_2, \ldots, a_d)$ is smaller than or equal to $(a'_1, a'_2, \ldots, a'_d)$ if and only if $a_1 \leq a'_1, a_2 \leq a'_2, \ldots a_d \leq a'_d$. The full lattice contains $\prod_{i=1}^{d} n_i^{d-1}$ elements with its greatest element $(a_1, a_2, \ldots, a_{d-1}, a_d)$, where each a_i is $\prod_{j,j\neq i}^{d}(n_j + 1)$; part of the lattice of a $2 \times 2 \times 2$ cube is illustrated in Figure 3.4. A $(k_1, k_2, \ldots, k_d)$-value vector $\vec{v}$ **restricts** $(k'_1, k'_2, \ldots, k'_d)$-value vector $\vec{w}$ when $\vec{v} \geq \vec{w}$ under the ordering previously defined. The **most restrictive vector** of diamond Δ is the largest vector that is associated with Δ, and we now show that there is a unique most restrictive vector associated with every diamond of cube C.

Lemma 3.1. *Given a non-empty cube C and a lattice of k-value-vectors, for every non-empty diamond of C there is a unique most restrictive ($k_1, k_2, \ldots, k_d$)-value vector.*

Proof. Consider the k-value-vector $\vec{K}$ with each $k_i = \min(\sigma(S_i))$ where S_i are slices along dimension i, that corresponds to the $k_1, k_2, \ldots, k_d$-diamond Δ of C. Diamond dicing with $\vec{K}$ produces Δ. If we restrict the vector—by increasing at least one of the k-values k_i, $i \in 1, 2, \ldots d$—we produce $\vec{K'}$. If $\vec{K'}$ also corresponds to Δ then all attribute values x_i must have $\sigma(\text{slice}_i)$ greater than the corresponding element k_i of $\vec{K}$. This contradicts our original statement; therefore a diamond for $\vec{K'}$ is not Δ and there is a most restrictive $(k_1, k_2, \ldots, k_d)$-value vector associated with every diamond in C. □

We define the most restrictive vector of the empty cube as the $(m_1, m_2, \ldots, m_{d-1}, m_d)$-vector, where each m_i is $\prod_{j,j\neq i}^{d}(n_i + 1)$.

Diamonds often cover more than one lattice element and Figure 3.5 illustrates the lattice and diamonds of all possible cubes of shape $2 \times 3 \times 1$. The lattice elements coloured black are the most restrictive vectors for their respective diamonds. Figure 3.6e is a two-dimensional example of this phenomenon.

The lattice creates optimisation opportunities: given the $(2, 1)$-carat diamond X and the $(1, 2)$-carat diamond Y, then the $(2, 2)$-carat diamond must lie in both X and Y. Although intersecting X and Y does not necessarily yield the $(2, 2)$-carat diamond, the intersection is a good starting point for a $(2, 2)$-carat diamond computation. (See Figure 3.6d.)

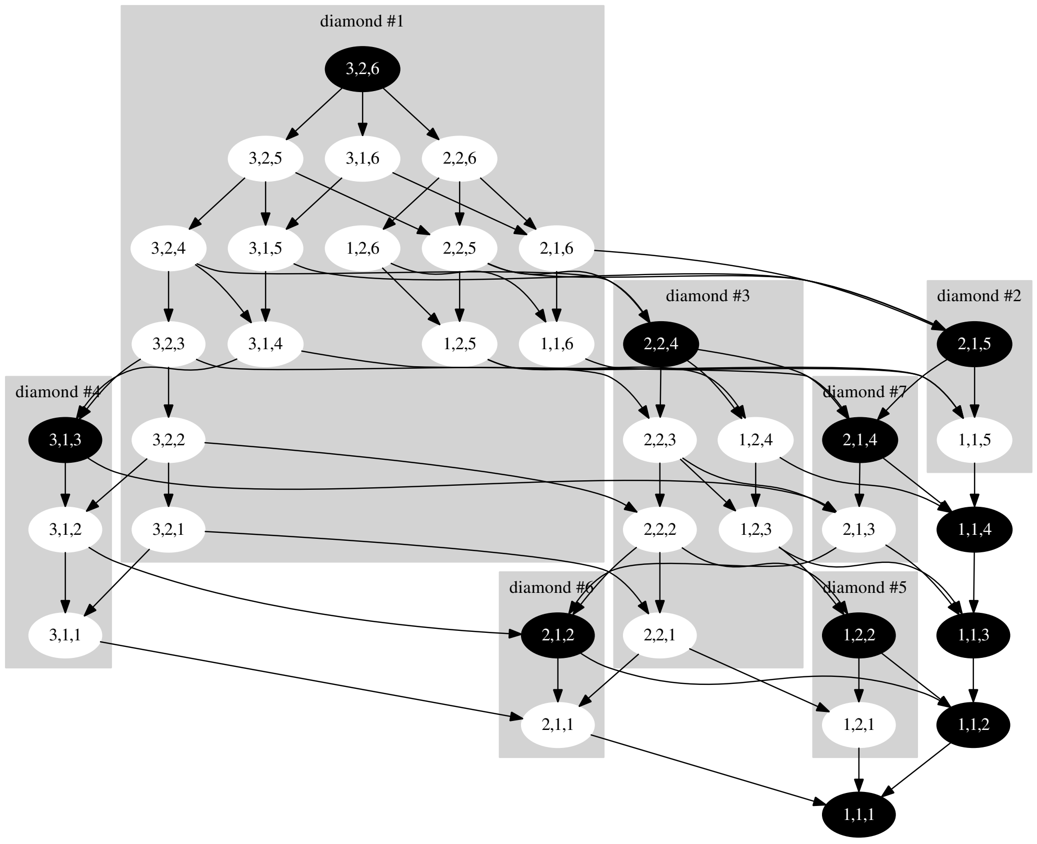


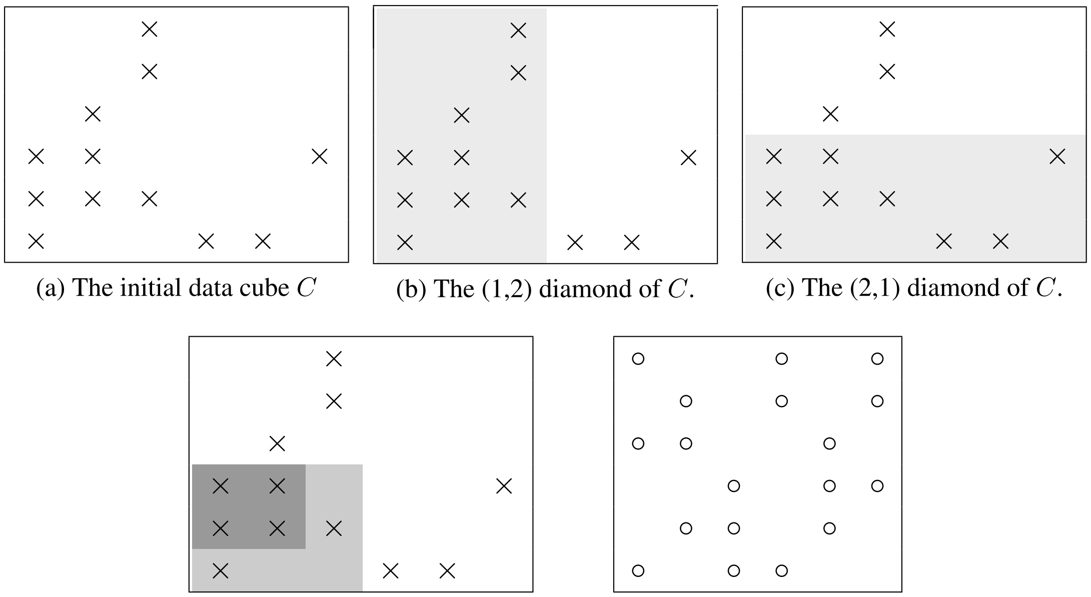
Figure 3.5: Lattice of all possible cubes that are diamonds of shape $2 \times 3 \times 1$. The shaded diamonds cover several lattice elements. Each diamond is associated with its most restrictive vector (shown in reverse print).

Not only do the unique diamonds of a non-empty cube C cover several elements of the k-valued-vector lattice, but they also form a lattice of their own as the next proposition shows.

Proposition 3.3. *The collection of all distinct diamonds of a cube C form a lattice.*

Proof. First, we need to define a partial ordering over diamonds since a lattice is a partial order. Second, we need to show that for every pair of diamonds of C there exists a supremum and an infimum.

First, from Lemma 3.1, we have that every diamond of C has a unique most restrictive $k_1, k_2, \ldots, k_d$-value vector. If we associate each diamond with its most restrictive vector $\vec{V}$ then we can use the ordering of k-value vectors to order the diamonds. Thus, the


(a) The initial data cube C

(b) The (1,2) diamond of C.

(c) The (2,1) diamond of C.

(d) The $(2,2)$ diamond (dark grey) is contained in the intersection (grey) of the $(2,1)$ diamond and the $(1,2)$ diamond of C.

(e) Several lattice elements may be the same cube: (1,1) (1,2) (1,3) (2,1) (2,2) (2,3) (3,1) (3,2) (3,3) are all this cube D.

Figure 3.6: Diamonds are nested.

$k_1, k_2, \ldots, k_d$-diamond C' is less than or equal to the $k'_1, k'_2, \ldots, k'_d$-diamond C''if and only if $k_1 \leq k'_1, k_2 \leq k'_2, \ldots, k_d \leq k'_d$. We write $C' \leq_\diamond C''$. The collection of diamonds in C form a partially ordered set since $\leq_\diamond$ is transitive, reflexive and anti-symmetric.

Second, we need to show that every pair of diamonds of C has a supremum and an infimum. Let L be the collection of diamonds in C. Let diamond Δ be associated with vector $\vec{V} = (v_1, v_2, \ldots v_d)$ and diamond Δ' be associated with vector $\vec{V'} = (v'_1, v'_2, \ldots, v'_d)$. From Proposition 3.2 both diamonds are contained in the diamond Δ'' associated with vector $\vec{V''}$, where $(v''_1, v''_2, \ldots, v''_d) = (\min(v_1, v'_1), \min(v_2, v'_2), \ldots \min(v_d, v'_d))$. So Δ'' is a lower bound. To show that it is the *greatest* lower bound, we need to show that any other diamond that contains both Δ and Δ' is less than Δ''. Suppose there is a diamond χ that also contains Δ and Δ' and further suppose that $\chi \not\leq_\diamond \Delta''$. The most restrictive vector $\vec{X} = (x_1, x_2, \ldots, x_d)$ associated with χ has $x_i > v''_i$ for some i. Without loss of generality, $v''_i = v_i$ and since $\vec{V}$ is most restrictive, Δ contains a slice whose σ is exactly

v_i. But since $x_i > v_i$, this slice is not in χ, a contradiction. Therefore, $\chi \leq_\diamond \Delta''$ and every pair of diamonds in L has an infimum.

The final step is to show that every pair of diamonds in L has a supremum. Let diamond Δ be associated with vector $\vec{V} = (v_1, v_2, \ldots v_d)$ and diamond Δ' be associated with vector $\vec{V}' = (v'_1, v'_2, \ldots, v'_d)$. From Proposition 3.2 both diamonds contain the diamond Δ'' associated with vector $\vec{V}''$, where $(v''_1, v''_2, \ldots, v''_d) = (\max(v_1, v'_1), \max(v_2, v'_2), \ldots \max(v_d, v'_d))$. We need to show that any other diamond that is contained by both Δ and Δ' is greater than Δ''. Suppose there is a diamond χ that is also contained by Δ and Δ'. Further suppose that $\chi \not\geq_\diamond \Delta''$. The most restrictive vector $\vec{X} = (x_1, x_2, \ldots, x_d)$ associated with χ has $x_i < v''_i$ for some i. Without loss of generality, $v''_i = v_i$ and χ contains a slice whose σ is less than v_i. This slice is not in Δ since $\vec{X}$ is most restrictive, a contradiction. Therefore, $\chi \geq_\diamond \Delta''$ and every pair of diamonds in L has a supremum. The collection of distinct diamonds of C form a lattice. □

The least element of the lattice, the $(1, 1, \ldots, 1, 1)$-diamond, is the diamond where every slice has at least one allocated cell, typically the initial cube. The greatest element is the $(m_1, m_2, \ldots, m_{d-1}, m_d)$-diamond, where each m_i is $\prod_{j,j\neq i}^{d}(n_i + 1)$, and is the empty diamond.

3.4 Properties of COUNT-based Diamonds

It is useful to know the ranges of values that $k_1, k_2, \ldots, k_d$ may take for any given cube. When the range of values for k_i is known, we can guarantee an empty diamond will result from a query whose k_i fall outside the range. Constraints on k when σ is COUNT are given in the following propositions. Without loss of generality we assume that $n_1 \leq n_2 \leq \ldots \leq n_{d-1} \leq n_d$, and n_i represents the number of *non-empty* slices in dimension D_i.

By considering a **perfect** cube, the following is immediate.

Proposition 3.4. *Given the sizes of dimensions of a* COUNT*-based diamond cube, n_i, an upper bound for the number of carats k_i for dimension i of a non-empty subcube is $\prod_{j=1,j\neq i}^{d} n_i$. An upper bound on the number of carats k of a non-empty subcube is $\prod_{i=1}^{d-1} n_i$.*

An alternate (and trivial) upper bound on the number of carats in any dimension is $|\mathbf{C}|$, the number of allocated cells in the cube. For sparse cubes, this bound may be more useful than that from Proposition 3.4.

Intuitively, a cube with large carats needs to have many allocated cells: accordingly, the next proposition provides a lower bound on the size of the cube given the number of carats.

Proposition 3.5. *For $d > 1$ and $\sigma =$ COUNT, the size $|C|$ of a d-dimensional non-empty cube C of $k_i, k_2, \ldots, k_d$ carats satisfies*

$$|C| \geq \max k_i n_i \geq (\prod k_i)^{1/(d-1)}, i \in \{1, 2, \ldots, d\}.$$

In particular, if C is a k-carat cube, then $|C| \geq k n_d \geq k^{d/(d-1)}$

Proof. For the first inequality we observe that for fixed i, C has precisely n_i slices along dimension i, each of which has at least k_i cells, so $|C| \geq k_i n_i$, whence $|C| \geq max\{k_i n_i\}$. For the second inequality, observe that by Proposition 3.4,$k_i \leq \prod_{j\neq i} n_j$. Therefore, we have

$$\prod_{i=1}^{d} k_i = k_1 \prod_{i=2}^{d} k_i \leq \prod_{i=2}^{d} n_i k_i \leq [\max(n_i k_i)]^{d-1}$$

The result follows. □

We calculate the *volume* of a cube C as $\prod_{i=1}^{i=d} n_i$ and its *density* is the ratio of allocated cells, $|C|$, to the volume ($|C| / \prod_{i=1}^{i=d} n_i$). We can exploit the relationship between the size of a diamond and its density to determine whether a diamond of particular carat exists in a given data cube.

Theorem 3.1. *For* COUNT*-based carats, if a cube C does not contain a non-empty $k_1, \ldots, k_d$-carat subcube, then $|C|$ is at most $1 + \sum_{i=1}^{d}(k_i - 1)(n_i - 1)$. Hence, it has density at most*

$(1 + \sum_{i=1}^{d}(k_i - 1)(n_i - 1))/\prod_{i=1}^{d} n_i$. *In particular, a cube that does not contain a non-empty k-carat subcube has size at most $1 + (k - 1)\sum_{i=1}^{d}(n_i - 1)$ allocated cells and density at most $(1 + (k - 1)\sum_{i=1}^{d}(n_i - 1))/\prod_{i=1}^{d} n_i$.*

Proof. Suppose that a cube of dimension at most $n_1 \times n_2 \times \cdots \times n_d$ contains no k-carat diamond. Then some slice must contain at most $k - 1$ allocated cells. Remove this slice. The amputated cube must not contain a k-carat diamond. Hence, it has some slice containing at most $k - 1$ allocated cells. Remove it. This iterative process can continue at most $\sum_i(n_i - 1)$ times at which point there is at most one allocated cell left: hence, there are at most $(k-1)\sum_i(n_i-1)+1$ allocated cells in total. The more general result follows similarly. $\square$

The contrapositive form of Theorem 3.1 is also useful.

Corollary 3.1. *A cube with $|C| > 1+\sum_{i=1}^{d}(k_i-1)(n_i-1)$ that is, having density greater than*

$$\frac{1 + \sum_{i=1}^{d}(k_i - 1)(n_i - 1)}{\prod_{i=1}^{d} n_i},$$

must contain a non-empty $k_1, k_2, \ldots, k_d$-carat subcube.
If $|C|$ is greater than $1 + (k - 1)\sum_{i=1}^{d}(n_i - 1)$ then C must contain a non-empty k-carat subcube.

3.4.1 Maximum number of carats

Given a cube C and σ, $\kappa(C)$ is the largest number of carats for which the cube has a non-empty diamond. Intuitively, a small cube with many allocated cells should have a large $\kappa(C)$, and the following proposition makes this precise.

Proposition 3.6. *For* COUNT*-based carats, we have $\kappa(C) \geq |C|/\sum_i(n_i - 1) - 3$.*

Proof. Solving for k in Corollary 3.1, we have a lower bound on the maximal number of carats:

$$\kappa(C) \geq |C| / \sum_i (n_i - 1) - 3$$

□

3.5 Properties of SUM-based Diamonds

For SUM-based diamonds, the goal is to capture a large fraction of the sum. The statistic, $\kappa(C)$, of a SUM-based diamond is the largest sum for which there exists a non-empty diamond: every slice in every dimension has sum at least $\kappa(C)$. Propositions 3.7 and 3.8 give tight lower and upper bounds respectively for κ.

Proposition 3.7. *Given a non-empty cube C and the aggregator* SUM*, a tight lower bound on κ is the value of the maximum cell (m).*

Proof. The κ-diamond, by definition, is non-empty, so it follows that when the κ-diamond comprises a single cell, then κ takes the value of the maximum cell in C. When the κ-diamond contains more than a single cell, m is still a lower bound: either κ is greater than or equal to m. □

Given only the size of a SUM-based diamond cube (in cells), there is no upper bound on its number of SUM-carats. However, given its sum, say S, then it cannot have more than S SUM-carats. We can determine a tight upper bound on $\kappa(C)$ as the following proposition shows.

Proposition 3.8. *A tight upper bound for κ is*

$$\min_i(\max_j(\text{SUM}(\text{slice}_j(D_i)))) \text{ for } i \in \{1, 2, \ldots, d\} \text{ and } j \in \{1, 2, \ldots, n_i\}.$$

Proof. Let $X = \{\text{slice}_j(D_i) \mid \text{SUM}(\text{slice}_j(D_i)) = \max_k(\text{SUM}(\text{slice}_k(D_i))\} \; i \in \{1, \ldots, d\}$ then there is one slice x whose SUM(x) is smaller than or equal to all other slices in X.

Suppose $\kappa(C)$ is greater than $\textsc{sum}(x)$ then it follows that all slices in this κ-diamond must have SUM greater than $\textsc{sum}(x)$. However, x is taken from X, where each member is the slice for which its SUM is maximum in its respective dimension, thereby creating a contradiction. Such a diamond cannot exist. Therefore, $\min_i(\max_j(\textsc{sum}(\text{slice}_j(D_i))))$ is an upper bound for κ. To show that $\min_i(\max_j(\textsc{sum}(\text{slice}_j(D_i))))$ is also a tight upper bound we only need to consider a perfect cube where all measures are identical. $\square$

We can also provide a lower bound on the sum of a k-carat diamond as the next lemma indicates.

Lemma 3.2. *If the diamond of size $s_1 \times s_2 \times \cdots \times s_d$ has k-carats, then its sum is at least $k \max(s_i)$ $i \in \{1, 2, \ldots, d\}$.*

Proof. For fixed i each of the s_i slices along dimension i have sum at least k, from which the result follows. $\square$

3.6 Robustness of $\kappa(C)$

One statistic of a cube C is its carat-number, $\kappa(C)$. Is this statistic robust? i.e. with high probability, can changing a small fraction of the data set change the statistic much? Of course, typical analyses are based on thresholds (e.g. applied to support and accuracy in rule mining), and thus small changes to the cube may not always behave as desired. Diamond dicing is no exception. For the cube C in Figure 3.7b and the statistic $\kappa(C)$ we see that diamond dicing is not robust against an adversary who can deallocate a single cell: deallocation of the second cell on the top row means that the cube no longer contains a COUNT-based diamond with 2 carats. This example can be generalised.

Proposition 3.9. *Let σ be* COUNT. *There is a cube C from which deallocation of any b cells results in a cube C' with $\kappa(C') \leq \kappa(C) - \frac{b}{2}$.*

$$\boxed{\begin{array}{cccccc} 7 & 2 & & & 1 & \\ 7 & 2 & & & & \\ & & 7 & 7 & & \\ & 5 & 2 & 2 & & \\ & 2 & & & & 7 \\ & & & & 8 & 2 \end{array}}$$

(a) A 9-carat SUM-diamond. It is not robust against the deletion of a single cell.

$$\boxed{\begin{array}{cccccc} 1 & 1 & & & & \\ 1 & 1 & & & & \\ & 1 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & & 1 & 1 \\ & & & & & 1 \quad 1 \end{array}}$$

(b) An $n \times n$ cube with $2n$ allocated cells and a 2-carat diamond in the upper left.

Figure 3.7: Diamonds are not necessarily robust.

Proof. Let C be a d-dimensional cube with $n_i = 2$. Then $\kappa(C) = 2^{d-1} = k$ and $|C| = 2^d$. Deallocate b cells of C, producing the cube C', so that $|C'| = 2^d - b$. Let $\kappa(C') = k'$. From Proposition 3.5 we have $|C'| \geq n_d k' = 2k'$. Therefore, we have $2^d - b \geq 2k'$, which implies $k' \leq 2^{d-1} - \frac{b}{2} = k - \frac{b}{2}$. □

Conversely, in Figure 3.7b we might allocate the cell above the bottom-right corner, thereby obtaining a 2-carat diamond with all $2n + 1$ cells. Compared to the original case, we see that a small change effects a very different result.

The statistic $\kappa(C)$ is no more robust when σ is SUM as the following example shows:

Example 3.1. *Consider Figure 3.7a. By deallocating the second cell in the fourth row* (5) *the 9-carat* SUM*-diamond becomes the 7-carat* SUM*-diamond and* $\kappa(C') < \kappa(C)$.

Conversely, one can add a single cell, with a value greater than the current $\kappa(C)$, to the cube C resulting in cube C' with $\kappa(C') > \kappa(C)$.

Diamond dicing is not, in general, robust. However, it is perhaps more reasonable to follow Pensa and Boulicaut's work on Formal Concept Analysis [Pens 05] and ask whether κ appears, experimentally, to be robust against random noise on realistic data sets. This is explored experimentally in Subsection 5.10.

	col1	col2	col3	col4
row1	1	1	1	1
row2	2	-4	1	1
row3	1	1	1	1
row4	1	1	1	1

Figure 3.8: Data cube with positive and negative measures.

3.7 The k-SUM Cube Problem

When we consider the SUM operator, the definition of a diamond cube is restricted to cubes with all positive (resp. all negative) measures. If one allows negative and positive measures in the same cube, there may no longer be a *unique* solution to the problem 'find the $k_1, k_2 \dots k_d$-carat cube'.

By way of illustration, consider Figure 3.8, a simple two-dimensional example. If the rows are processed first and k is 3, the result is the cube given in Figure 3.9. However, if the columns are processed first we are left with the cube in Figure 3.10. Both are maximal cubes with 3 carats under the SUM aggregator.

	col1	col2	col3	col4
row1	1	1	1	1
row3	1	1	1	1
row4	1	1	1	1

Figure 3.9: SUM-cube ($k = 3$) generated from Figure 3.8. Rows processed first.

	col1	col3	col4
row1	1	1	1
row2	2	1	1
row3	1	1	1
row4	1	1	1

Figure 3.10: SUM-cube ($k = 3$) generated from Figure 3.8. Columns processed first.

Intuitively, we can see that there may be cases when an attribute value is deleted early and, as a result of a subsequent deletion, it should be reinstated into the diamond. Consider Figure 3.11. If the rows are processed first, then row 1 and row 2 are deleted on the first pass. When the columns are processed, column 1 is deleted, which should result in the reinstatement of rows 1 and 2.

Data cubes, in general, are not restricted to positive (resp. negative) measures. Consider

row	col 1	col 2	col 3	col 4	col 5	col 6
1	-2	1	1	1	0	2
2	-30	4	1	0	1	0
3	2	2	4	0	2	1
4	0	2	3	1	0	2

Figure 3.11: There is no unique diamond for $\kappa = 5$. Rows 1 and 2 could be reinstated after deletion of column 1.

	Puppies	Dog Food	Kennels
Mr. Breeder	+20	-5	-6
Ms Butcher	-15	+4	-1
Ms Carpenter	-1	-2	+5
Mr. Pro	-3	+3	+2
$\Sigma \geq 0$	1	0	0

Figure 3.12: An instance of MTP. All trades can be carried out if a broker collects all the goods offered and distributes all the goods wanted. She would then reap a profit of one puppy.

Figure 3.12, a matrix representing people with items they wish to trade and items they require. Negative measures indicate the number of items required and positive measures the number of items available for trade. The Maximum Trade Problem (MTP) [Luo 94] takes just such a matrix and seeks the largest set of trades possible.

An MTP Instance: A matrix $A = [a_{i,j}]$ of size $m \times n$ where each $a_{i,j}$ is an integer—possibly negative—and p, a positive integer.

Question: Does there exist an index set $I \subseteq \{1, 2, \ldots, m\}$ of size at least p, that is $|I| \geq p$, so that $\Sigma_{i \in I} a_{i,j} \geq 0$,for all $j = 1, \ldots, n$?

The Maximum Trade Problem is NP-complete. A multi-dimensional version of this problem, one we call the k-SUM Cube Problem (kSCP) is also NP-complete as shown in the following discussion.

The k-SUM Cube Problem (kSCP) is formally described: Given a d-dimensional data cube, C, with measures in $\mathbb{Z}$, integers $k_1, k_2, \ldots, k_d$ and a shape $s_1 \times s_2 \times \cdots \times s_d$, does there exist a $k_1, k_2 \ldots k_d$-carat cube with at least the specified shape? i.e. the dimension cardinalities are no smaller than $s_1, s_2, \ldots, s_d$.

Theorem 3.2. *The k-SUM Cube Problem (kSCP) is NP-Complete.*

Proof. The first requirement is to show that kSCP belongs to the class NP. This is straightforward. If a cube $A = [a_{i,j}]$, $i = 1, \ldots, m$; $j = 1, \ldots, n$ defines a possible solution to the problem, then checking that this satisfies the constraints:

- $\Sigma a_{i,1}, \Sigma a_{i,2}, \ldots, \Sigma a_{i,n} \geq k_1$ for all $i = 1 \ldots m$
- $\Sigma a_{1,j}, \Sigma a_{2,j}, \ldots, \Sigma a_{m,j} \geq k_2$ for all $j = 1 \ldots n$
- $m \geq s_1$ and $n \geq s_2$

can all be done in polynomial time; moreover, it can be done in $O(mn)$.
If we consider a cube with more than two dimensions then checking that a potential solution satisfies the constraints for each dimension can also be done in polynomial time—Let N be the number of bits in the input. Clearly, the number of attribute values is O(N). To calculate the sum for a given slice is O(N), thus to check all sums against the threshold is O(N^2).
The second and final requirement is to show that this problem is NP-hard. The proof is by reduction from the Maximum Trade Problem (MTP).
An MTP instance has an explicit parameter, p (the size of the index set), and an implicit parameter 0 (each column must have a non-negative sum). An MTP instance can be transformed into a kSCP instance by setting

- $s_1 = p$
- $s_2 = n$
- $k_1 = -\infty$
- $k_2 = 0$

row sums of original	7	1	2	4	1	-1
	9	3	4	-1	2	1
	5	-1	3	1	1	1
	9	1	-1	2	2	5
	-1	1	-3	1	-1	1
(column sum - k_2) ≥ 0		0	0	0	0	0

Figure 3.13: An MTP instance $s_1 = 3, s_2 = n = 5$. The $k_1 = 7, k_2 = 5$-diamond is shaded.

Then MTP is equivalent to asking whether there is a $0, -\infty$-carat cube having at least size (p, n). The constructed kSDP instance has answer YES if and only if the original MTP instance has answer YES.

□

As stated in Section 3.2, we restrict diamonds to the cubes where all measures are positive (resp. negative). We have proved a uniqueness property for diamonds and established upper and lower bounds on $\kappa(C)$. The logical next step is to find efficient algorithms to compute diamonds. This is the topic of the next chapter.

Chapter 4

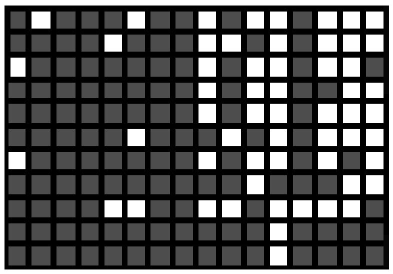
Algorithms

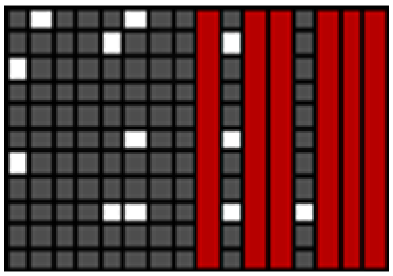
Computing diamonds is challenging because of the interaction between dimensions; modifications to a measure associated with an attribute value in one dimension have a cascading effect through the other dimensions. We need to consider what kind of data structures are needed to store the aggregated measures for each slice. The linearity of σ implies that it is enough to compute SUM diamonds; other linear aggregation problems can always be recast into sum, e.g. for COUNT we can replace each measure by 1 and then compute the sum.

Several different approaches were taken to develop algorithms for computing diamonds. Although there is no publicly available version of Kumar's algorithm [Kuma 99], introduced in Section 2.3, we have developed one that resembles the description provided in his paper. It has been extended to address more than two dimensions. An algorithm using SQL is presented in Section 4.2.4. An intuitive solution, that loops through the cube checking and updating the COUNT or SUM for all attribute values in each dimension until it stabilises, provides the basis for our most efficient algorithm.

The basic algorithm for computing diamonds is given in Algorithm Diamond Dice (DD). Its overall approach is illustrated by Example 4.1 that was first introduced in Chapter 1. The approach is to repeatedly identify an attribute value that cannot be in the diamond, and then remove the attribute value and its slice. The identification of "bad" attribute values is done conservatively, in that they are known already to have a sum less than required (σ is SUM), or insufficient allocated cells (σ is COUNT). When the algorithm terminates, only attribute values that meet the condition in every slice remain: a diamond.

Example 4.1. *An 8-COUNT–diamond is determined by processing the rows and columns of the two-dimensional cube in Figure 4.1. On a first pass, each row has at least eight allocated cells, so no deletions are executed. However, columns representing Switzerland, Austria, Luxembourg, Netherlands, Norway and Denmark are all deleted. On a second pass, the row representing letter bomb is deleted since it no longer has eight allocated cells. No further columns or rows are deleted; they all have at least eight allocated cells. The algorithm then terminates. The columns could have been processed first without affecting the final result.*


(a) Original data


(b) Switzerland, Austria, Luxembourg, Netherlands, Norway and Denmark are deleted

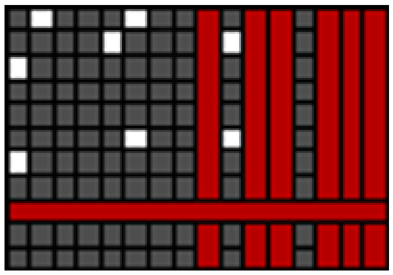

(c) Letter bomb is deleted. Remaining slices have ≥ 8 cells

Figure 4.1: Cube 5: Country $\times$ Action.

Algorithms based on this approach always terminate, though they might sometimes return

```
input: a $d-$dimensional data cube $C$, a monotonic linear operation $\sigma$ and
    $k_1 \geq 0, k_2 \geq 0, \ldots, k_d \geq 0$
output: the diamond cube $A$
stable ← false
while ¬stable do
    // major iteration
    stable ← true
    for dim $\in \{1, \ldots, d\}$ do
        for i in all attribute values of dimension dim do
            $C_{\texttt{dim,i}} \leftarrow \sigma$(slice i on dimension dim)
            if $C_{\texttt{dim,i}} < k_{\texttt{dim}}$ then
                delete attribute value i
                stable ← false
            end
        end
    end
end
return cube without deleted attribute values;
```
Algorithm DD: Algorithm to compute the diamond of any given cube by deleting slices eagerly.

an empty cube. By specifying *how* to compute and maintain sums for each attribute value in every dimension we obtain different variations. The correctness of any such variation is guaranteed by the following result.

Theorem 4.1. *Algorithm DD is correct, that is, it always returns the $k_1, k_2, \ldots, k_d$-carat diamond.*

Proof. Because the diamond is unique, we need only show that the result of the algorithm, the cube A, is a diamond. If the result is not the empty cube, then dimension D_i has at least value k_i per slice, and hence it has k_i carats. We only need to show that the result of Algorithm DD is maximal: there does not exist a larger $k_1, k_2, \ldots, k_d$-carat cube.

Suppose A' is such a larger $k_1, k_2, \ldots, k_d$-carat cube. Because Algorithm DD begins with the whole cube C, there must be a first time when one of the attribute values of dimension dim of C belonging to A' but not A is deleted. At the time of deletion, this attribute's slice cannot have obtained more cells so it still has value less than $k_{\texttt{dim}}$. Let C' be the cube at the instant before the attribute is deleted, with all attribute values deleted so far. We see that C' is larger than or equal to A' and therefore, slices in C' corresponding to attribute values of A' along dimension dim must have more than $k_{\texttt{dim}}$ carats. Therefore, we have a contradiction and must conclude that A' does not exist and that A is maximal. $\square$

```
input: a $d-$dimensional data cube $C$ and $k_1 \geq 0, k_2 \geq 0, \ldots, k_d \geq 0$
output: the diamond $A$
Q ← empty collection
loop
    Actions ← empty set of possible actions
    if there is attribute i of dimension dim with $\sigma(\text{slice}_{\texttt{dim,i}}) < k$ that is not already
    in Q then
        Actions ←{Add to Q any new i and dim with $\sigma(\text{slice}_{\texttt{dim,i}}) < k$}
    end
    if Q is not empty then
        Actions ← Actions ∪ {Delete arbitrary slice in Q and its corresponding
        slice information }
        // Q contains only pending deletions
    end
    if Actions is not empty then
        Arbitrarily choose and execute one action from Actions
    else
        return the modified $C$
    end
```

Algorithm QDD: Queued-deletion approach to compute the diamond of any given cube.

Implementations of Algorithm DD, and variations that can process data sets of an unrestricted size, are discussed in the next two sections. The chapter concludes with three alternate approaches to determine $\kappa(C)$.

4.1 Main Memory Algorithms

When the volume of the cube to be processed fits into main memory, there are some implementation choices to be made:

- An allocation table mapping cells to booleans can be stored in multidimensional binary arrays if a COUNT diamond is to be found. A '1' represents an allocated cell; a '0' an unallocated cell [Albe 06]. An issue that hinders performance of an array-based implementation is the number of zeros that are stored in sparse cubes. Typically data cubes are not dense; the large data cubes used in our experiments had densities of between 5.4×10^{-6} and 1.6×10^{-23}.

The overall performance of the array-based implementation was poor, taking hours to process a moderately-sized 4-dimensional cube of two thousand allocated cells with volume 102 608. Since every iteration runs in $\Omega(\prod n_i)$ time—every potential cell is counted in each iteration—it is very inefficient. It might work well on more dense cubes, but this approach was not explored further, because of the inefficiency and the fact that we wanted to process large sparse data sets.

- A multi-list [Knut 97] is a more efficient data structure for an in-memory implementation. It is a multiply linked list whose nodes belong to several lists, one per dimension. Each allocated cell is aware of all the lists it belongs to, as in Example 4.2. Only allocated cells are stored, removing the main obstacle to good performance of the array-based implementation. A multi-list was used in the implementation of the queue-based Algorithm QDD. Effectively, in each outermost loop, slices to be deleted are en-queued until all dimensions have been processed. Then, the queue is completely processed, and the next iteration of the outermost loop can begin. Let I be the number of iterations until the diamond stablises. Algorithm QDD runs in time O($I|C|d$); for each iteration we examine the remaining attributes in every dimension d. The SUMS can be maintained in a multidimensional array assuring a bound of O(1) for verifying $\sigma(\text{slice})$ and since order of insertion or deletion is not important we can assume a bound of O(1) for queue operations.

Example 4.2. *Suppose a 4-dimensional cube has an allocated cell at coordinates (1,9,2,6). It has a link to some element at* 1xxx*, a second link to some element at* $\text{x}9\text{xx}$*, a third link to some element at* $\text{xx}2\text{x}$ *and another at* $\text{xxx}6$*.*

4.2 External-Memory Algorithms

The size of available memory affects the capacity of in-memory data structures to represent data cubes. In our experiments we restricted the size of available memory to less than 1 GiB. It is, of course, possible to associate a region of memory with a file, which can be bigger than RAM, and thus implement a data structure that uses more space than the available RAM. Since some of the data files we wish to process are very large—over 20 GiB—the execution speed of such an algorithm may still be very slow and careful attention must be paid to data access patterns [Coze 04]. The data transfer rate of external disks—in particular, flash memory Solid State Drives (SSD)—is getting closer to that of main memory [Lee 09]. However, flash media blocks must be erased before they can be reused so random write access is considerably more expensive than a read operation [Agra 08].
Because a memory-mapped data structure may require random writes, such an implementation on an SSD might not provide the most efficient solution. A simpler and more natural extension, which would enable larger cubes to be processed, is to find an efficient external memory implementation. The data cube can be stored in an external file whilst the important COUNTS (resp.SUMS) are maintained in memory.

Algorithm DD visits each dimension in sequence until it stabilises. Ideally, the stabilisation should occur after as few iterations as possible. Let I be the number of iterations through the input file till convergence; i.e. no more deletions are done. Value I is data dependent and (by Figure 4.4) is $\Theta(\sum_i n_i)$ in the worst case. In practice, I is not expected to be nearly so large, and working with large real data sets I did not exceed 56. Algorithms DD and QDD run in time O($Id|C|$), providing that we can iterate through and delete all cells in a slice, each in constant time per cell.

Experimentally, it appears that the relationship of I to k is bi-tonic, i.e. it is non-decreasing to $\kappa + 1$ and non-increasing thereafter. Is this always the case? Unfortunately, there is at least one cube for which this is not the case. Figure 4.2 illustrates such a cube. On the first iteration, processing columns first for the 2-carat diamond, a single cell is deleted. On

```
input: file inFile containing $d-$dimensional cube $C$, integers $k_1, k_2 \ldots k_d > 0$, a
        monotonic linear operation $\sigma$
// preprocessing scan computes $\sigma$ values for each slice
foreach dimension $i$ do
    Create hash table $\texttt{ht}_i$
    foreach attribute value $v$ in dimension $i$ do
        if $\sigma($ slice for value $v$ of dimension $i$ in $C) \geq k_i$ then
            $\texttt{ht}_i(v) = \sigma($ slice for value $v$ of dimension $i$ in $C)$
input: file inFile containing $d-$dimensional cube $C$, integers $k_1, k_2 \ldots k_d > 0$
output: the diamond data cube
stable ← false
while ¬stable do
    Create new output file outFile // iterate main loop

    stable ← true
    foreach row $r$ of inFile do
        $(v_1, v_2, \ldots, v_d) \leftarrow r$
        notDeleted ← true
        for $i \in \{1, \ldots, d\}$ do
            if $v_i \notin \operatorname{dom} \texttt{ht}_i$ then
                for $j \in \{1, \ldots, i-1, i+1, \ldots, d\}$ do
                    if $\texttt{v}_j \in \operatorname{dom} \texttt{ht}_j$ then
                        $\texttt{ht}_j(v_j) = \texttt{ht}_j(v_j) - \sigma(\{r\})$
                        if $\texttt{ht}_j(v_j) < k_i$ then
                            remove $v_j$ from $\operatorname{dom} \texttt{ht}_j$
                        end
                    end
                end
                stable ← false
                notDeleted ← false
                break
                // only delete this cell once
            end
        end
        if notDeleted then
            write $r$ to outFile
        end
    end
end
if ¬stable then
    inFile ← outFile // prepare for another iteration

end
return outFile
```

Algorithm SIDD: Diamond dicing for relationally stored cubes. Each iteration, less data is processed.

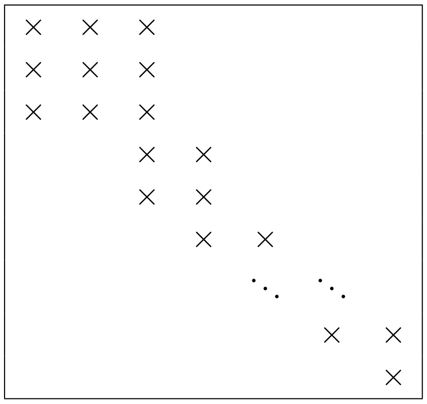


Figure 4.2: The 2-carat diamond requires more iterations to converge than 3-carat diamond.

subsequent iterations at most two cells are deleted until convergence. The 3-carat diamond converges after a single iteration. The value of k relative to κ does, however, influence I. Typically, when k is far from κ—either less or greater—fewer iterations are required to converge. However, when k exceeds κ by a very small amount, say 1, then typically a great many more iterations are required to converge to the empty cube.

4.2.1 A Hash-Table-Based Algorithm

Algorithm DD checks the σ-value for each attribute on every iteration. Calculating this value directly, from a data cube too large to store in main memory, would entail many expensive disk accesses.

The first external-memory solution, Algorithm Shrinking-Input Diamond Dice (SIDD), employs d hash tables that map attributes to their σ-values. A preprocessing step iterates once over the input file creating the d hash tables. When $\sigma = \text{COUNT}$, the σ-values for each dimension form a histogram, which might be precomputed in a DBMS.

If the cardinality of any of the dimensions is such that hash tables cannot be stored in main memory, then a file-based set of maps could be constructed. However, given a d-dimensional cube, there are only $\sum_{i=1}^{d} n_i$ slices and so the memory usage is $O(\sum_{i=1}^{d} n_i)$: for the experiments conducted, main memory hash tables suffice.

Algorithm SIDD reads and writes the files sequentially from and to disk and does not require potentially expensive random access, making it a candidate for a data-parallel implementation. In the data-parallel model, multiple processors execute the same commands on a partition of the data. In this case, the cube could be partitioned with each processor having access to a shared set of hash tables.

Algorithm SIDD runs in time O($Id|C|$); each attribute value is deleted at most once. We have I iterations of the outermost loop. At each iteration, the body of the j loop is executed at most $|C|$ times. Therefore, we examine O($|C|$) records each of which has d fields. We assume an amortized bound of O(1) for each hash table look-up or modification, and expect an implementation using the Hashtable class from the Java API to have such a bound on hash table operations. Often, the input file decreases substantially in the first few iterations and those cubes are processed faster than the overall bound suggests. The more carats we seek, the faster the file decreases initially.

4.2.2 Normalisation

A major cost of executing any external-memory algorithm is the time spent accessing data on disk. It is prudent, therefore, to reduce the number of I/O operations as much as possible: one way this can be achieved is to store the data cube as normalised binary integers using bit compaction [Ng 97]—mapping strings to small integers starting at zero. Recall, of course, that there are a finite number of values that a 32-bit integer can take. One could use 64-bit integers, thereby increasing the potential size of each dimension in the data cube. However, the overall file size may not be decreased and unless the dimension sizes warrant it, a 32-bit integer is preferred as it consumes 50% less memory.

A file storing binary data may be a fraction of the size of a file storing textual data. For example, given a comma-separated file where each three-dimensional record can be represented in 20 bytes, the same data when normalised requires only 12 bytes. This represents a space-saving of up to 40% and an associated reduction in the number of disk accesses.

With this in mind, a variation on Algorithm SIDD was implemented, one that takes a binary file of packed integers as its input. Consequently, the internal data structure used to maintain the list of SUMS for each attribute value is a set of ragged arrays. By normalising the data the files sizes are reduced and expensive string manipulation eliminated, thus giving an improvement in performance.

The speed of convergence of Algorithm SIDD and indeed the size of an eventual diamond may depend on the data-distribution skew. Cell allocation in data cubes is very skewed and frequently follows Zipfian distributions [Nade 03]. Suppose the number of allocated cells $C_{\texttt{dim},i}$ in a given slice i follows a Zipfian distribution: $P(C_{\texttt{dim},i} = j) \propto j^{-s}$ for $s > 1$. The parameter s is indicative of the skew. We then have that $P(C_{\texttt{dim},i} < k_i) = \sum_{j=1}^{k_i-1} j^{-s} / \sum_{j=1}^{\infty} j^{-s} = P_{k_i,s}$. The expected number of slices marked for deletion after one pass over all dimensions, prior to any slice deletion, is thus $\sum_{i=1}^{d} n_i P_{k_i,s}$.This quantity grows fast to $\sum_{i=1}^{d} n_i$ (all slices marked for deletion) as s grows. (See Figure 4.5.) For SUM-based diamonds, we not only have the skew of the cell allocation, but also the skew of the measures to accelerate convergence. In other words, we expect Algorithm SIDD to converge quickly over real data sets, but more slowly over synthetic cubes generated using uniform distributions.

4.2.3 A Sorting-Based Algorithm

The related work of Kumar et al. [Kuma 99] models the Web as a large directed graph. They suggest that large bipartite graphs found within the Web characterise emerging communities. In their model an emerging community has an (i, j) core which comprises sets of i web pages each linking to a set of j pages ("fans") and sets of j pages all of which have i links to them ("centres"). Their first step is to find these cores—bicliques—and then use them to discover the rest of the web community—bipartite graph. They discuss two different algorithms, both employing a sorting step, with which to prune away nodes that cannot be part of a core. A variation of their first algorithm has been implemented

cell	productID	salesPersonID	month
i	1	a	Jan
ii	1	b	Jan
iii	2	c	Feb
iv	2	d	Mar
v	3	a	Feb
vi	3	b	Feb
vii	3	d	Apr
viii	4	c	Apr
ix	4	e	Jan

Figure 4.3: Cube C.

and extended to d dimensions. It seeks a bipartite graph directly and its pseudocode is given in Algorithm Iterative-Pruning Diamond Dice (IPDD). Algorithm IPDD runs in time $O(I|C|d^2)$. We have I iterations of the outermost loop. At each iteration we process d files of O($|C|$) records and each record has d items. We assume an amortized bound of O(1) for each hash table look-up or insertion.

The second algorithm discussed by Kumar et al., one they refer to as the inclusion-exclusion algorithm, is unsuitable for use as a diamond builder since they require that fans are related to *exactly* i centres and that centres are related to *exactly* j fans. This does not conform to the definition of a diamond cube which allows for attributes to be related to k *or more* attributes in all other dimensions.

4.2.4 An SQL-Based Algorithm

A data cube can be represented in a relational database by a fact table. (See Figure 4.3.) In this example the dimensions are: productID {1,2,3,4}, salesPersonID {a,b,c,d,e} and month {Jan,Feb,Mar,Apr}. The cell numbers are noted in this example, but they do not form part of the cube. Formulating a diamond cube query in SQL is challenging since examples can be constructed where only one attribute ever has $\sigma(\text{slice}_{\texttt{dim,i}}) < k$, but its removal exposes another attribute, and so on. See Figure 4.4 and consider computing the

```
input: $d$ files each sorted on one of the dimensions, containing a $d-$dimensional
       cube $C$, integers $k_1, k_2 \ldots k_d > 0$, empty hash sets $s_1, s_2, \ldots, s_d$
output: the diamond data cube
stable ← false
while ¬stable do
    stable ← true
    for i ∈ $\{1, \ldots, d\}$ do
        file ← sortedfile_i
        r ← current row
        (attribute_1, attribute_2, ..., attribute_d, val) ← $r$
        while ¬EOF do
            f_i ← attribute_i
            sum ← 0
            repeat
                // traverse slice $f_i$ in dimension $i$
                deleted ← false
                for j ∈ $\{1, \ldots, d\}$ do
                    if attribute_j ∈ dom s_j then
                        // this attribute was deleted previously
                        deleted ← true
                    end
                end
                if ¬deleted then
                    // only consider the undeleted attributes
                    sum ← sum + val
                end
                r ← next row
                (attribute_1, attribute_2, ..., attribute_d, val) ← $r$
                field_i ← attribute_i
            until field_i ≠ f_i or EOF
            if sum < k and f_i ∉ dom s_i then
                s_i add f_i
                stable ← false
            end
        end
    end
end
// All deleted attributes are now recorded in the hash
   sets.The diamond is extracted with a simple file
   scan and check
```
Algorithm IPDD: Iterative pruning algorithm due to Kumar [Kuma 99]. Requires d copies of the data sorted in each of d dimensions.

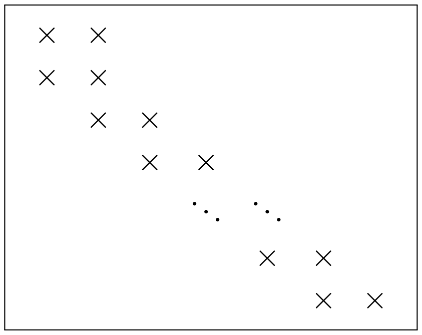


Figure 4.4: An $n \times n$ cube with $2n$ allocated cells (each indicated by a $\times$) and a 2-carat diamond in the upper left: it is a difficult case for several algorithms.

2-carat diamond from the cube in Figure 4.3. Every dimension must be examined three times to determine that cells i, ii, v and vi comprise the 2-carat diamond.

Using nested queries and joins, we could essentially simulate a fixed number of iterations of the outer loop in Algorithm DD. Unfortunately, we cannot bound the number of iterations for that loop by a constant, leaving our pure-SQL approach as merely approximating the diamond. In Figure 4.4, we see an example where the number of iterations $I = \Omega(n)$ and stopping after $o(n)$ iterations results in a poor approximation with $\Theta(n)$ allocated cells and attribute values—whereas the true 2-diamond has 4 attribute values and 4 allocated cells.

Algorithm SQLDD runs in time O($Id|C|$), assuming hash indices permitting O(1)-time checks on any temporary table *temp*$_i$. In the event that a B-tree index is used, the running time of Algorithm SQLDD would increase to $O(Id|C| \log_b(|n_d|))$ where b is the branching factor of the tree. Often, the relation $\mathcal{R}$ decreases substantially in the first few iterations, and those cubes are processed faster than this bound suggests.

We express the essential calculation in core SQL, as Algorithm SQLDD, but this calculation is repeated by an "outermost loop" outside of SQL[1]. Algorithm SQLDD can be executed against a copy of the fact table, which becomes smaller as the algorithm progresses. An alternate implementation, one that avoids the expensive DELETE operation,

[1]SQL-2003 supports looping and recursion [Terr 08]; thus the entire algorithm could be implemented in SQL, for those DBMSs that actually implement enough of SQL-2003.

```
INPUT: a $d-$dimensional data cube $C$ and $k > 0$
OUTPUT: the diamond $A$
initialise $\mathcal{R}$ to $C$, the fact table
repeat {major iteration}
   execute the fragment of SQL pseudocode shown below
until no records were deleted from $\mathcal{R}$
return $\mathcal{R}$ as $A$
```

```
CREATE TABLE $temp_1$ AS
   (SELECT $dim_1$ FROM $\mathcal{R}$
    GROUP BY $dim_1$ HAVING $\sigma(measure) < k$);

...
CREATE TABLE $temp_d$ AS
   (SELECT $dim_d$ FROM $\mathcal{R}$
    GROUP BY $dim_d$ HAVING $\sigma(measure) < k$);
DELETE FROM $\mathcal{R}$
  WHERE $dim_1$ IN (SELECT * FROM $temp_1$) OR ...
        $dim_d$ IN (SELECT * FROM $temp_d$);
```

Algorithm SQLDD: Variation where the inner two loops in Algorithm DD are (almost) computed in SQL (Indices on the temporary relations are assumed.) This process can be repeated until $\mathcal{R}$ stabilises.

is to supplement the fact table with a boolean-valued column χ that indicates whether this fact is in the diamond cube. If an index is created for each column, excluding χ, the diamond cube computation can be sped up. In this new context, deletion of a slice is simply of the form "UPDATE facttable SET χ = 0 WHERE DIM = i". Recreating a smaller fact table after the first few passes may be used to further speed up the query since we expect the fraction of slices marked for deletion to be high initially. To determine the exact computational complexity of these SQL queries, we need to specify what type of index is used. Ideally, the indexes should have the performance of a hash table, i.e. with constant time look-up. When $k_1 = k_2 = \ldots = k_d$, we can replace χ with an integer-valued column that indicates whether the cell belongs to the χ_i-carat cube since each cell belongs to a single smallest diamond cube. See Table 4.1 as an example. It is impractical to maintain this kind of information for every possible combination of $k_1, k_2, \ldots, k_d$.

Four different versions of Algorithm SQLDD were implemented and tested using MySQL

Table 4.1: Fact table with a column indicating the maximum number of carats for the corresponding cell in Table 2.2b.

productID	salesPersonID	carat
1	a	2
1	b	2
2	c	1
2	d	1
3	b	2
3	d	1
4	c	1
4	e	1
3	a	2

5.0. The only indexing option available for our large cubes was a B-tree, so for each version B-tree indexes were built on all attribute values, except for column χ. (Code is given in Appendix B.1.)

- The first (SQL1) deleted slices from a copy of the data cube whenever the COUNT of an attribute value fell below the k threshold.
- The second (SQL2) did not delete any information from the data cube, but instead updated a boolean valued-attribute.
- The third version (SQL3) updated a boolean-valued attribute and also rebuilt the data cube when the number of cells marked for deletion was at least 50% of the original.
- The fourth version (SQL4) updated a boolean-valued attribute and rebuilt the data cube whenever 75% of the remaining cells were marked for deletion.

The results are given in Chapter 5

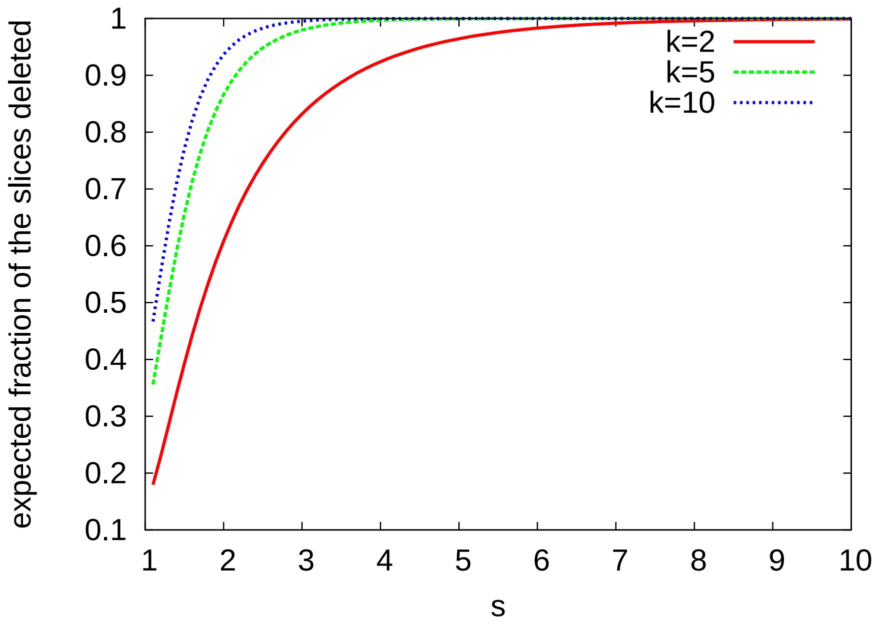


Figure 4.5: Expected fraction of slices marked for deletion after one pass under a Zipfian distribution for various values of the skew parameter s.

4.3 Finding the Largest Number of Carats, $\kappa(C)$

The determination of $\kappa(C)$, the largest value of k for which C has a non-trivial diamond, is a special case of the computation of the diamond-cube lattice. (See Proposition 3.2.) Identifying $\kappa(C)$ may help guide analysis. Once κ is determined we can remove the κ-diamond, then seek the κ-1 diamond, then seek the $\kappa - 2$ diamond and so on, until we have a set of dense clusters, which can be used for further statistical analysis as suggested by Barbará [Barb 01]. We assume an integer value for $\kappa(C)$. In the case of SUM-diamonds, where κ may not be an integer, we compute $\lfloor \kappa(C) \rfloor$. Three approaches to finding κ were identified:

1. Set the parameter k to 1 + the lower bound, provided by Proposition 3.6[2] for COUNT-diamonds or Proposition 3.7 for SUM-diamonds, and check whether there is a diamond with k carats. Repeat, incrementing k, until an empty cube results. At each step, Proposition 3.2 says the computation can start from the cube from the previous

[2]or Theorem 6.2

iteration, rather than from C. If the lower bound is very far from κ this results in a long, slow computation and, therefore, may not be the best approach.

2. Observe that $\kappa(C)$ is in a finite interval. For COUNT-diamonds, we have a lower bound from Proposition 3.6 and an upper bound $\prod_{i=1}^{d-1} n_i$ or $|C|$. (If this upper bound is unreasonably large, the computation could start with the lower bound and repeatedly double it.) For SUM-diamonds, we have a lower bound from Proposition 3.7 and an upper bound from Proposition 3.8. Execute the diamond-dicing algorithm and set k to a value determined by a binary search over its valid range. Every time a new midpoint k is tested, the computation can begin from the largest nonempty k-diamond that has been seen so far. (See Proposition 3.2 and note that Algorithm SIDD does not destroy its input.)

   The second approach is better. Let us compare one iteration of the first approach (which begins with a k-carat diamond and seeks a $k+1$-carat diamond) and a comparable iteration of the second approach (which begins with a k-carat diamond and seeks a $(k+k_{\text{upper}})/2$-carat diamond). Both will end up making at least one scan, and probably several more, through the k-carat diamond. Binary search, when it makes an unsuccessful attempt while narrowing in on $\kappa(C)$, may result in fewer scans since, experimentally, we observed that processing a k-carat diamond for values of k much larger than $\kappa(C)$ typically requires fewer scans. Furthermore, a binary search makes fewer attempts overall unless $\kappa(C)$ lies within $\log(\text{upper} - \text{lower})$ of the lower bound. Binary search is recommended, given that it will find $\kappa(C)$ in O($\log \kappa(C)$) iterations.

3. Our third approach is one adapted from a numerical technique. Although function dependent, it has been shown to exhibit a convergence rate of $\log(\log(n))$ over some continuous and monotonically increasing functions [Arme 85]. It is not normally used for searching memory-resident data since the calculations are more costly and

complex than the bit-shift required for binary search. However, since the major time cost of searching for κ occurs during diamond-building, any method that could reduce the number of diamonds generated is likely to be a good choice. The Pegasus method, first introduced by Dowell and Jarratt [Dowe 72], is a modified Regula-Falsi method for finding a simple root of a continuous function and has been adapted to the discrete case [Stra 82]. The new guess for κ, x_3, is generated by equation 4.1, where x_2 is the most recent guess, x_1 is the guess before that and f_2 and f_1 are the file sizes of the x_1 and x_2 diamonds.

$$x_3 = x_2 - (x_2 - x_1) \times f_2/(f_2 - f_1) \tag{4.1}$$

When the most recent guess for κ falls on the same side of the root as the previous guess, the value for f_1 is modified using equation 4.2.

$$f_1 = (f_2/(f_2 + f_3)) \times f_1 \tag{4.2}$$

The Pegasus method assumes a single zero on the interval to be searched, but we want to find the point at which the function **first** becomes zero (the k-value of the first empty diamond). Applying Equation 4.1 and Equation 4.2 will not modify the value of x_3: we have already found a zero—the empty cube at the upper end of the search interval. Therefore, we must adjust f_1 or f_2 to force a change of value for x_3. The choices experimented with were values close to zero, $-f_1$, and much smaller than $-f_1$. The results are discussed in Chapter 5.

Chapter 5

Experiments

Experiments were designed to assess whether

- our theoretical bounds on $\kappa(C)$ hold.
- diamonds can be computed efficiently, i.e. within a few minutes and using less than 1 GiB of memory even for very large data sets.
- the size of the κ-diamond is affected by the number of dimensions.
- $\kappa(C)$ is robust against modest amounts of missing data.

We implemented the three different strategies for finding $\kappa(C)$ as described in Section 4.3. The lower bound estabished in Proposition 3.6 was used as an initial guess for $\kappa(C)$ and in every case it returned a non-empty diamond.

A comparison of the algorithms presented in Chapter 4 illustrates that the customised Java code computes diamonds more than fifteen times faster than either the sorting-based approach of Algorithm IPDD, or the SQL-based approach of Algorithm SQLDD on cubes of any significant size. Since it is difficult to express the diamond dice query in SQL, and our Java implementation was faster, we recommend the addition of the diamond dice as primitive to a future version of SQL.

We discovered that the κ-diamond captured a large fraction of some of our cubes. Therefore, we designed experiments to determine the relationship (if any) between the number of dimensions and the size of the κ-diamond. The results indicate that the distribution of the data has more effect on the size of a diamond than the number of dimensions.
This chapter concludes with some experiments designed to test the robustness of $\kappa(C)$. They were carried out on real-data sets and a synthetically generated cube whose attributes follow a Zipfian distribution. From these experiments, we find that $\kappa(C)$ is robust against modest amounts of missing data.

5.1 Hardware and Software

All experiments were carried out on an Apple Mac Pro with two double-core Intel Xeon processors (2.66 GHz) and 2 GiB of RAM. Experiments used a 500 GiB SATA Hitachi disk (model HDP725050GLA360 [Hita 09, Dell 09]), with average seek time (to read) of 14 ms, average rotational latency of 4.2 ms, and capability for sustained transfers at 300 MiB/s. This disk also has an on-board cache size of 16 MiB, and is formatted for the Mac OS Extended filesystem (journaled). The software runs on Mac OS 10.4.

5.2 Implementations

An implementation of Algorithm DD was validated on several small cubes for which the solution could be found by hand. It was then applied to a couple of cubes extracted from TWEED [Enge 07] a small real data set. These cubes became our unit test data and each algorithm implementation was validated against them. The code is archived at my personal website [Webb 09].
Algorithms SIDD, IPDD and SQLDD were implemented in Java using SDK version 1.5.0. At run-time the *-server* flag was set; it increased the memory usage to 192 MB [1] and turned

[1]as reported by the jstat utility

Table 5.1: Cube K2: the number of duplicates was recorded and used as the measure for the SUM-diamond.

aaron	aaron	abihu	aaron
aaron	aaron	abihu	eleazar
aaron	aaron	abihu	eleazar
aaron	aaron	abihu	eleazar
aaron	aaron	abihu	ithamar
aaron	aaron	abihu	ithamar
aaron	aaron	abihu	ithamar
aaron	aaron	accept	befor
aaron	aaron	acept	lord

(a) Raw data

aaron	aaron	abihu	aaron	1
aaron	aaron	abihu	eleazar	3
aaron	aaron	abihu	ithamar	3
aaron	aaron	accept	befor	1
aaron	aaron	acept	lord	1

(b) Cube K2 data

on compiler optimisations, such as method in-lining, which improve the performance of long-running software. Each of our algorithms was implemented in less than 275 lines of code: Kumar, Algorithm IPDD, used the fewest and DBCount—the string implementation of Algorithm SIDD—used the most.

5.3 Data Used in Experiments

A varied selection of freely-available real-data sets together with some systematically generated synthetic data sets were used in the experiments. Each data set had a particular characteristic: a few dimensions or many, dimensions with high or low cardinality or a mix of the two, small or large number of cells. They were chosen to illustrate that diamond dicing is tractable under varied conditions and on many different types of data.

Brief descriptions of how each data cube was extracted from the raw data follow: the complete details are given in Appendix A. In every case duplicates were eliminated from the resulting cube by aggregating the measure. Table 5.1 shows the "before" and "after" view of a subset of one of our cubes, K2.

5.3.1 Real Data

Six of the seven real-data sets were downloaded from the following sources.

1. TWEED: http://folk.uib.no/sspje/tweed.htm [Enge 07]
2. Netflix: http://www.netflixprize.com [Netf 07]
3. Census Income: http://kdd.ics.uci.edu/ [Hett 00]
4. Census 1881:http://www.prdh.umontreal.ca/census/en/main.aspx
5. Weather: http://cdiac.ornl.gov/ftp/ndp026b/ [Hahn 04]
6. Forest: http://kdd.ics.uci.edu/ [Hett 00]

The seventh data set was generated from the King James version of the Bible available at Project Gutenberg [Proj 09].

TWEED

The TWEED [Enge 07] data set contains over 11 000 records of individual events related to internal terrorism in 18 countries in Western Europe between 1950 and 2004. Of the 52 attributes in the TWEED data, 37 were regarded as measures since they recorded the number of people killed and injured in each of the terrorist attacks. For instance, a record might indicate that 5 people were killed and 10 people injured, of whom 2 were police, 11 were civilians and 2 were other militants. For our purposes these measures were aggregated and thus the measure associated with this record in our data cube would be 15. Cube TW1 retained dimensions Country, Year, Action and Target with cardinalities of $16 \times 53 \times 11 \times 11$. For cubes TW2 and TW3 all fifteen dimensions not deemed measures were retained. Cardinalities of the dimensions ranged from 3 to 284. Table 5.2 gives statistics of the TWEED data and full details can be found in Table A.2.

Table 5.2: Statistics of TWEED, Netflix and census data.

cube	dimensions	$\|C\|$	$\sum_{i=1}^{d} n_i$	measure
TW1	4	1 957	88	count
TW2	15	4 963	674	count
TW3	15	4 963	674	sum killed
NF1	3	100 478 158	500 137	count
NF2	3	100 478 158	500 137	sum rating
NF3	4	20 000 000	473 753	count
C1	27	135 753	5 607	count
C2	27	135 753	504	sum stocks
C3	27	135 753	504	sum wages
C4	7	4 277 807	343 442	count

Netflix

At approximately 2 GiB, the Netflix [Netf 07] data set is the largest openly-available movie-rating database. It comprises 4 dimensions: MovieID $\times$ UserID $\times$ Date $\times$ Rating. The data can be downloaded as a set of 17 770 text files in a tar archive. Each file contains between 4 and 232 945 ratings for a single movie, attributed to distinct reviewers. The files were amalgamated into a single comma-separated file by executing a purpose-built Java application. Two three-dimensional cubes NF1 and NF2, both with about 10^8 allocated cells, were extracted using dimensions MovieID, UserID and Date. The measure, Rating, together with the aggregator, SUM, was used for NF2, whereas the COUNT aggregator was used for NF1. A third cube—NF3—was extracted from the Netflix data when we discovered that the SQL algorithm was taking a very long time (over 24 hours) to run. NF3 has four dimensions and it comprises the first 20 million records of the data. The aggregator COUNT was used for NF3.

Census Income

The third real-data set, Census-Income, comes from the UCI KDD Archive [Hett 00]. The cardinalities of the dimensions ranged from 2 to 91 and there were 135 753 records. The original 41 dimensions were rolled-up to 27 and three cubes were extracted. The COUNT

Table 5.3: Statistics of forest, weather and King James Bible data.

cube	dimensions	$\|C\|$	$\sum_{i=1}^{d} n_i$	measure
F1	11	580 950	15 923	count
W1	12	123 245 878	50 320	count
W2	12	123 245 878	50 320	sum total cloud cover
W3	12	1 002 751	18 597	count
K1	4	363 412 308	33 588	count
K2	4	363 412 308	33 588	sum 4-gram instances
K3	10	986 145 104	1 558	count
K4	10	986 145 104	1 558	sum 10-gram instances
K5	4	24 000 000	8 270	count
K6	4	32 000 000	9 141	count
K7	4	40 000 000	9 561	count
K8	4	48 000 000	9 599	count

aggregator was used for C1. Two fields, income from stocks (C2) and hourly wage (C3) were the measures for the aggregator SUM. The query used to generate cube C2 and details of the original data are given in Appendix A.1 and Appendix B. This data set has been widely used [Ben 06b, Kase 08, Lemi 10, Aoui 07].

Census 1881

The Census 1881 data set came from a publicly available SPSS file *1881_sept2008_SPSS.rar* that was converted to a flat file [Lemi 09]. In the process the special values "ditto" and "do" were replaced by the repeated value and all commas within values were deleted. Cube C4 has seven dimensions ranging in cardinality from 138 to 152 882. It has over 4 million records.

King James Bible

KJV-4grams [Kase 08] is a data set motivated by applications of data warehousing to literature. It is a large list (with duplicates) of 4-tuples of words obtained from the verses in the King James Bible [Proj 09], after stemming with the Porter algorithm [Port 97] and removal of stemmed words with three or fewer letters. A verse is defined as a sequence of text with *digit:digit* at the beginning of a line and paragraphs are delimited by empty lines. Occurrence of row w_1, w_2, w_3, w_4 indicates that the first paragraph of a verse contains words w_1 through w_4, in this order. This data is a scaled-up version of word co-occurrence cubes used to study analogies in natural language [Turn 05, Kase 06]. K1 was extracted from KJV-4grams. Four subcubes of K1 were also processed: K5: first 24 000 000 rows; K6: first 32 000 000 rows; K7: first 40 000 000 rows and K8: first 48 000 000 rows.

KJV-10grams has similar properties to KJV-4grams, except that there are 10 words in each row and the process of creating KJV-10grams was terminated when 1 billion records had been generated—at the end of Genesis 3:15. Cube K3 was extracted from KJV-10grams. Duplicate records were removed using the post-processing step as described in Appendix A.4. A count of each unique sequence was kept during post-processing and became the measure for cubes K2 and K4. The statistics for all eight cubes are given in Table 5.3.

Forest Cover

As with Census Income, the Forest Cover data came from the UCI KDD Archive [Hett 00]. The first 11 dimensions in the data set were extracted using a Java application and the resulting comma separated file was sorted with its duplicate records removed. On inspection, there were a few corrupted records containing parentheses, ampersands or periods, which were handled as described in Appendix A.3. Cardinalities of the dimensions of cube F1 range from 2 to 5 786. The cube has 580 950 cells.

Weather

Surface synoptic weather reports for the entire globe for the period December 1st, 1981 to November 30th, 1991 were downloaded from The Carbon Dioxide Information Analysis Center (CDIAC) [Hahn 04]. Details of the extraction process and a full description of the data are given in Appendix A.6 and Table A.3. The cardinalities of the dimensions of

Table 5.4: Statistics of the synthetic data cubes.

Cube	dimensions	$\lvert C\rvert$	$\sum_i n_i$
U1	3	999 987	10 773
U2	4	1 000 000	14 364
U3	10	1 000 000	35 910
S1	3	939 153	10 505
S2	4	999 647	14 296
S3	10	1 000 000	35 616
SS1	3	997 737	74 276
SS2	4	999 995	99 525
SS3	10	1 000 000	248 703

cubes W1 and W2 range from 2 to 12 155.

As with the Netflix data set, aggregators COUNT and SUM were used as measures for cubes W1 and W2 respectively. Statistics for the weather cubes are given in Table 5.3. One month of this data—land observations for September 1983—is often used by other researchers [Feng 04, Wang 02, Xin 03] and was also used in this work for W3.

5.3.2 Synthetic Data

To investigate the effect that data distribution might have on the size and shape of diamonds, nine cubes of 1 000 000 cells with varying dimensionality and distribution were constructed. The first set comprised three cubes whose values in each dimension were independently generated from a uniform distribution on the range 0 to 3591. This range was chosen to be comparable with the $\sum_i n_i$ of cubes S1, S2 and S3. For the three-dimensional cube (U1), 13 duplicate cells were discarded, resulting in a cube of 999 987 cells. The 4-D (U2) and 10-D (U3) cubes had no duplicate cells.

As stated in Section 4.2, real data has often been observed to follow a Zipfian distribution so the remaining data cubes were generated with a power distribution. Data were generated using this transformation method: Let r be a random number uniformly distributed in the range $0 < r < 1$ then

$$x = x_{\min}(1 - r)^{-1/(\alpha-1)}$$

is a random power-law-distributed real number in the range $x_{min} \leq x < \infty$ with exponent α [Newm 05]. Cubes SS1, SS2 and SS3 were generated with an exponent of 2.0 in order to simulate the 80–20 relationship. The remaining cubes (S1, S2, S3) were generated with an exponent of 3.5 to vary the skew. Indeed, Newman [Newm 05] observed that the number of copies of best selling books sold between 1895 and 1965 followed power distribution with exponent 3.5. Statistics of the synthetic data cubes are given in Table 5.4.

5.4 Preprocessing Step

All of the data cubes used in the experiments were stored in comma-separated (CSV) flat files. Since we do not expect our algorithms to be sensitive to the row order, we did not reorder the rows prior to processing. As discussed in Chapter 4, the basic algorithm for building a diamond requires a COUNT (resp. SUM) of each attribute value in each iteration. Although the COUNT aggregator is referenced in the following discussion, it applies equally to the SUM aggregator.

The algorithms used in our experiments required different preprocessing of the cubes and the general preprocessing method is illustrated in Figure 5.1. For Algorithm SIDD an in-memory data structure was used to maintain counts of the attribute values. Algorithm IPDD used d sorted files and Algorithm SQLDD referenced the cube stored in a relational database. Consequently, the preprocessor wrote different kinds of data to supplementary files depending on which algorithm was to be used.

- Algorithm SIDD

  The preprocessor creates d hash tables mapping attribute values to their counts and all attribute values are treated as character strings. The mappings are converted to strings and stored in a flat file. Each (key, value) pair is delimited by a tab and stored

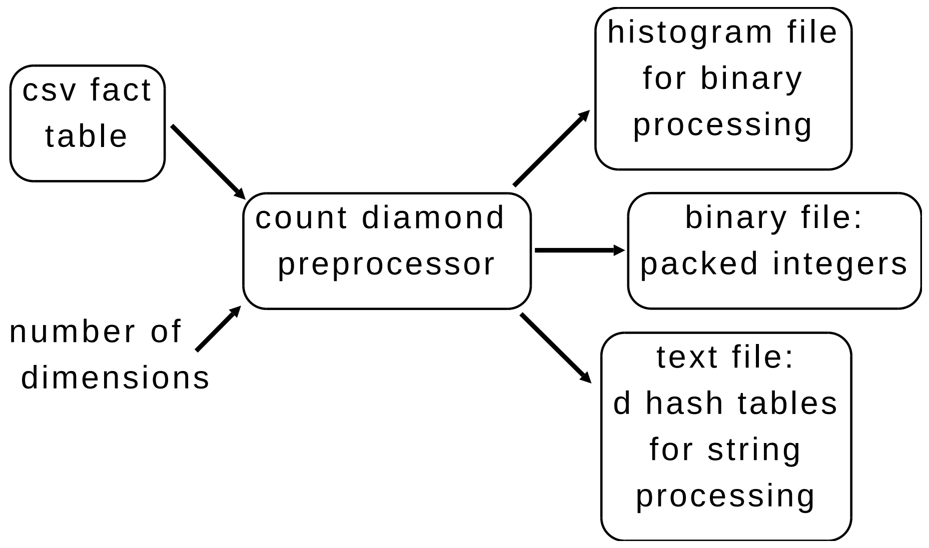

csv fact table
number of dimensions
count diamond preprocessor
histogram file for binary processing
binary file: packed integers
text file: d hash tables for string processing

Figure 5.1: Preprocessing stage.

on a separate line. A comma-separated file of normalised integers is also generated, which is used as input to the SQL-based Algorithm SQLDD.

For the binary variation of Algorithm SIDD, the preprocessor creates two binary files: a normalised file of packed integers, and one storing d ragged arrays mapping the normalised attribute values to their counts (a histogram).

- Algorithm IPDD

  Kumar's approach was to use two sorted files, one per dimension, in his Web-trawling algorithm. This work generalises his algorithm to d dimensions, so the preprocessor for Algorithm IPDD creates d files each sorted on a different dimension. Two sets of files are output: one of normalised binary data and the other, character data that are not normalised.

- Algorithm SQLDD

  The preprocessor for SQL1 creates a table in a local database and loads the comma-separated file into the table. It also makes a copy of the table as this algorithm deletes cells from the original cube. The preprocessor for SQL2/SQL3/SQL4 creates a table in a local database and appends a boolean-valued column to indicate whether

Table 5.5: Wall-clock times (in seconds) for preprocessing each real-world data set. DB-Count and DBSum implement Algorithm SIDD for both string and binary data. A '—' indicates that this algorithm was not applied to the corresponding data cube.

Cube	DBCount/DBSum (SIDD)	Kumar (IPDD)	SQL1 (SQLDD)	SQL2/SQL3 (SQLDD)
C1	6.8×10^0	5.0×10^1	7.0×10^{-1}	5.4×10^0
C2	1.3×10^1	—	—	—
C3	1.3×10^1	—	—	—
F1	1.3×10^1	4.7×10^1	1.5×10^0	4.1×10^1
K1	3.5×10^3	—	—	5.1×10^4
K2	3.2×10^3	—	—	—
K3	2.0×10^4	—	—	—
K4	1.8×10^4	—	—	—
K5	2.0×10^2	—	—	1.1×10^3
K6	2.8×10^2	—	—	1.4×10^3
K7	3.4×10^2	—	—	1.8×10^3
K8	4.0×10^2	—	—	2.2×10^3
NF1	1.5×10^3	2.5×10^3	1.1×10^2	—
NF2	1.4×10^3	—	—	—
NF3	2.4×10^2	—	2.5×10^1	8.8×10^2
TW1	1.9×10^{-1}	2.0×10^1	4.0×10^{-1}	4.0×10^{-1}
TW2	3.3×10^{-1}	1.3×10^0	8.0×10^{-1}	4.0×10^{-1}
TW3	3.6×10^{-1}	—	—	—
W1	2.6×10^3	—	—	—
W2	2.9×10^3	—	—	—
W3	2.7×10^1	—	2.3×10^0	7.5×10^1

this cell remains in the diamond. It also loads the comma-separated data file and constructs indexes when appropriate.

The preprocessing of the cubes was timed separately from diamond building. Preprocessed data could be used many times, varying the value for k, without incurring additional preparation costs. Table 5.5 summarises the times needed to preprocess each cube in preparation for the algorithms that were run against it. For comparison, sorting the Netflix comma-separated data file—using the GNU sort utility—took 29 minutes. All preprocessors were written in less than 250 lines of Java code using SDK 1.5.0. The timings were taken under the same settings and on the same machine described in Section 5.2

Two algorithms did not fare as well in computing diamonds as the others. Kumar (Algorithm IPDD) did not work well with high dimensional cubes. It required a sorted copy of the cube for each dimension which incurred additional space and I/O costs. SQL1 was much slower than the other SQL variants: we only ran it against a few cubes. We did not implement a SUM-diamond-builder with Kumar or the SQL variants and chose a subset of the remaining cubes against which to run SQL2 and SQL3.

5.5 Finding $\kappa(C)$ for COUNT-based Diamonds

To test the theory of Chapter 3, and in particular Proposition 3.6, the κ-diamond was built for each of the data sets. Three strategies had been identified to find κ. All approaches used the initial guess (k) for κ as the value calculated using Proposition 3.6. When a non-empty diamond was built, and for every data set this was indeed the case, k was repeatedly doubled until an empty cube was returned and a tighter range for κ had been established. The first approach was a basic binary search, which used the newly discovered lower and upper bounds as the end points of the search space. Each time a non-empty diamond was returned, it was used as the input to the next iteration of the search. When the guess overshot κ and an empty diamond was returned, the most recent non-empty cube was used as the input.

The second approach also made use of the new lower bounds that had been established for κ. This time k was incremented by one as long as a non-empty diamond was returned. In this way, the previously generated diamond could be reused at every iteration. Despite not having to start with the original cube, except on the first iteration, it quickly became apparent that this approach was not practical. If left to continue it would have taken approximately 60 days to establish κ for K1 as each iteration took between 100 and 200 seconds and 34 918 iterations would have executed before returning an empty diamond.

The third approach was to use an adaptation of the Pegasus method that was discussed in

Chapter 4. Unfortunately, this approach failed to produce a better result than the simple binary search. A zero for the ‘function’ is already known (an empty diamond was produced at the upper boundary) and applying Equation 4.1 directly gives the known upper boundary as the next value for x_3. If f_2 is given an arbitrary value, say $-f_1$ then the search range is effectively halved producing a similar result to binary search. By choosing a value close to zero, or much smaller than $-f_1$, for f_2 several values close to the upper boundary (resp. the lower boundary) were searched, and more iterations were required to find κ than when using binary search. Table 5.6 illustrates the results from applying a modified Pegasus algorithm to cube C1.

Table 5.6: The modified Pegasus search was no more efficient than binary search.

search	iters
modified Pegasus ($-f_1$)	9
modified Pegasus (-1)	20
modified Pegasus ($2 * -f_1$)	13
binary	8

5.5.1 King James Bible

The modified binary search approach was used to establish that κ is 125 433 for cube K1. The initial k-value chosen was the lower bound calculated from the equation given in Proposition 3.6:

$$\frac{363\,412\,308}{8\,245 + 8\,386 + 8\,415 + 8\,503} - 3 \approx 10\,829$$

This did indeed return a non-empty diamond and the k-value was doubled repeatedly until an empty diamond resulted (173 264). An upper bound for κ had now been established. Since a non-empty diamond had been returned when using k = 86 632, a binary search was executed over the range 86 632 to 173 264.

A modified binary search as described for K1 was executed for K3, which established that

Table 5.7: The number of iterations it took to determine κ for COUNT-based diamonds.

cube	iters to convergence	estimated κ	κ
TW1	6	19	38
TW2	10	5	37
NF1	19	197	1 004
NF3	17	39	272
C1	8	282	672
F1	29	34	92
W1	13	2 447	4 492
W3	17	50	178
K1	8	10 829	125 433
K3	6	637 042	9 464 226
K5	6	2 900	6 603
K6	13	3 500	21 170
K7	11	4 182	25 091
K8	5	5 000	54 769

Table 5.8: The number of iterations it took to determine κ on SUM-based diamonds. The estimate for κ is the tight lower bound from Proposition 3.7.

cube	iters to convergence	estimated κ	κ
TW3	3	85	85
NF2	40	5	3 483
C2	5	1 853	3 600 675
C3	7	9 999	71 464
W2	19	32	20 103
K2	3	1 216 437	1 216 437
K4	5	2	9 610 236

Table 5.9: Values remaining in the K3 κ-diamond.

dimension 1	dimension 6	dimension 10
said	upon	seed
unto	life	shall
lord	thou	head
thou	shalt	thou
serpent	belli	shalt
hast	dust	bruis
because	will	heel
done	enmiti	
	between	

the actual value is 9 464 226. As shown in Table 5.10, the cardinalities of the κ-diamond are quite small (7–10) and the values for dimensions 1, 6 and 10 are given in Table 5.9. The values for κ(K1) can be found in Appendix E. The definition of a verse used to generate the King James Bible cubes gave rise to an interesting result—a single large paragraph from the book of Joshua chapter 15 gave rise to the κ(K1) diamond.

5.5.2 Other Cubes

Following the procedure outlined in Section 5.5.1, κ was established for the other data cubes: statistics are provided in Table 5.7. The estimate of κ comes from Proposition 3.6 and the iterations recorded is the number needed to find the κ-diamond. The estimates for κ varied between 4% and 50% of the actual value and there is no clear pattern to indicate why this might be. Two very different cubes both had estimates that were 50% of the actual value: TW1, a small cube of less than 2 000 cells and low dimensionality, and W1, a large cube of 123×10^6 cells with moderate dimensionality.

5.6 Finding $\kappa(C)$ for SUM-based Diamonds

From Proposition 3.8, we have that $\min_i(\max_j(\sigma(\text{slice}_j(D_i))))$ is an upper bound for $\kappa(C)$ of any sum-diamond and from Proposition 3.7 a lower bound is the maximum value

Table 5.10: Statistics for real-data cubes and their respective κ-diamonds.

	dimension sizes	cells	% captured by κ-diamond
NF1	$17\,766 \times 480\,189 \times 2\,182$	100 479 586	9%
κ-diamond	$3\,082 \times 6\,833 \times 1\,351$	8 654 370	
K1	$8\,246 \times 8\,387 \times 8\,416 \times 8\,504$	363 412 308	4%
κ-diamond	$66 \times 72 \times 72 \times 65$	14 441 597	
K3	$145 \times 165 \times 154 \times 152 \times 156$ $\times 158 \times 158 \times 157 \times 153 \times 160$	986 145 104	30%
κ-diamond	$8 \times 10 \times 10 \times 10 \times 10$ $\times 9 \times 10 \times 11 \times 9 \times 7$	293 468 482	
W1	$28\,767 \times 2 \times 4\,337 \times 6\,344 \times 8\,696 \times 101$ $\times 13 \times 14 \times 11 \times 10 \times 1\,799 \times 226$	123 245 878	47%
κ-diamond	$12\,155 \times 2 \times 3\,433 \times 4\,746 \times 5\,990$ $\times 13 \times 14 \times 11 \times 10 \times 1\,591 \times 162$	58 389 402	
C1	$91 \times 9 \times 52 \times 47 \times 17 \times 3 \times 7 \times 24 \times 15$ $\times 5 \times 10 \times 2 \times 3 \times 6 \times 8 \times 6 \times 6 \times 51$ $\times 38 \times 8 \times 10 \times 9 \times 10 \times 3 \times 4 \times 7 \times 53$	135 753	42%
κ-diamond	$48 \times 7 \times 25 \times 27 \times 12 \times 3 \times 5 \times 20 \times 13$ $\times 4 \times 6 \times 2 \times 3 \times 1 \times 4 \times 5 \times 1 \times 1$ $\times 7 \times 6 \times 2 \times 2 \times 2 \times 2 \times 2 \times 7 \times 9$	57 536	
F1	$1\,978 \times 361 \times 67 \times 553 \times 701$ $\times 5\,786 \times 207 \times 185 \times 256 \times 5\,827 \times 2$	580 950	30%
κ-diamond	$908 \times 360 \times 37 \times 175 \times 226 \times 998$ $\times 120 \times 92 \times 185 \times 2$	175 275	

Table 5.11: Comparison of string and binary algorithms to compute the κ-diamond. Times were averaged over ten runs and then normalised against the binary execution times. All times reported are in seconds.

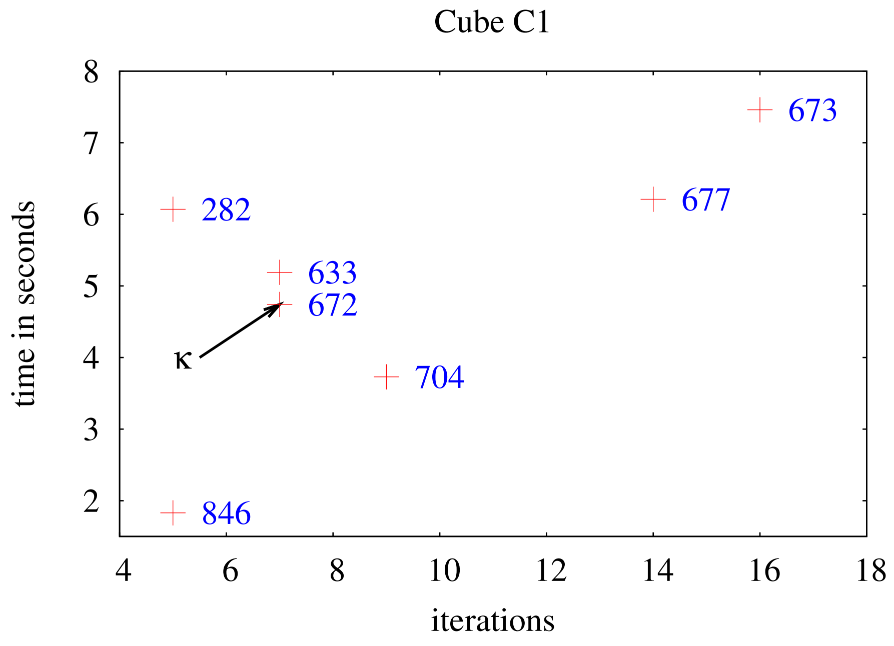
	TW3	NF2	NF3	W1	W2	W3	C2	C3	C4
binary	0.1	459	44	4 067	5 243	27	1.9	2.6	27
string	0.3	1 180	128	5 201	5 806	40.5	3.8	4.5	79
slowdown	3	2.6	2.9	1.3	1.1	1.5	2.0	1.7	3

	K1	K2	K3	K4	K5	K6	K7	K8
binary	661	4 627	11 902	11 277	150	91	120	65
string	1 83	6 851	19 027	17 358	282	222	284	185
slowdown	2.8	1.5	1.6	1.5	1.8	2.4	2.4	2.8

stored in any cell. Indeed, for cubes TW3 and K2 the lower bound was the κ value. For this reason, the approach to finding κ for the sum-diamonds varies slightly in that the first guess for k should be the lower bound + 1. If this returns a non-empty diamond, then a binary search over the range [lower bound +1–upper bound] is used to find κ. Statistics are given in Table 5.8. Although for two cubes the lower bound coincided with their $\kappa(C)$, in cube K4 there was no single cell (or slice) that dominated the data, since the maximum cell value was 2. In a cube with 10 dimensions and $\approx 9 \times 10^8$ cells, this resulted in the lower bound being six orders of magnitude from $\kappa(\mathsf{K4})$.

5.7 Comparison of Algorithm Speeds

Neither the initial file size nor the number of cells pruned from each k-diamond alone explains the time necessary to generate each diamond. As can be seen in Figure 5.2 more time is expended when k is close to κ. When k is much larger than κ we converge faster because more attribute values are pruned in early iterations, thus quickly reducing the file size. We also see the converse, a few iterations over large files may take more time than several iterations over small files, as evidenced by the time taken to process the five iterations when $k = 282$ on cube C1. In fact, the observed run-time was proportional to the total number of cells processed, as illustrated in Figure 5.3.



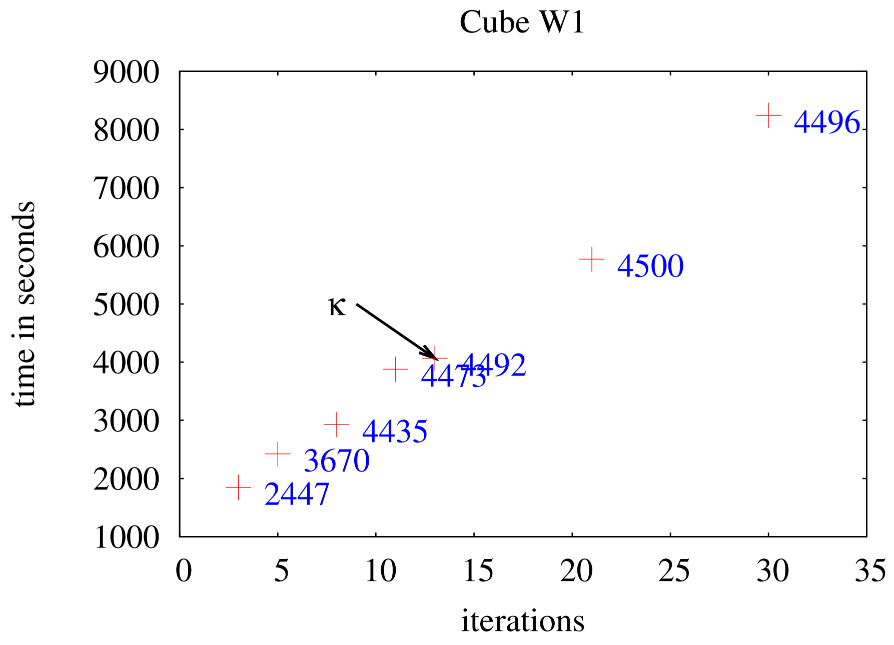
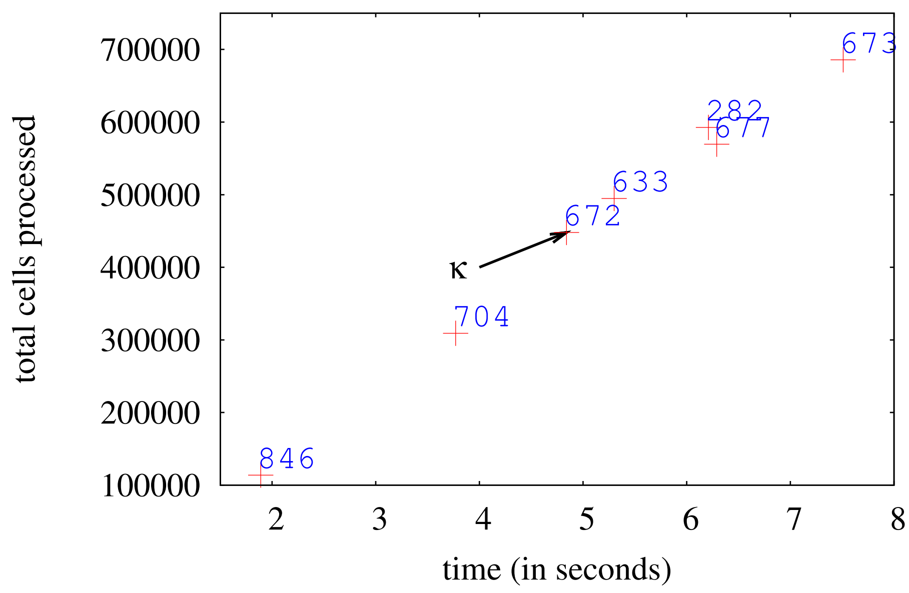


Figure 5.2: Times and iterations needed to generate diamonds with different k-values (labelled). In each case more time and iterations are required for k-values that slightly exceed κ.

Cube C1



Cube W1

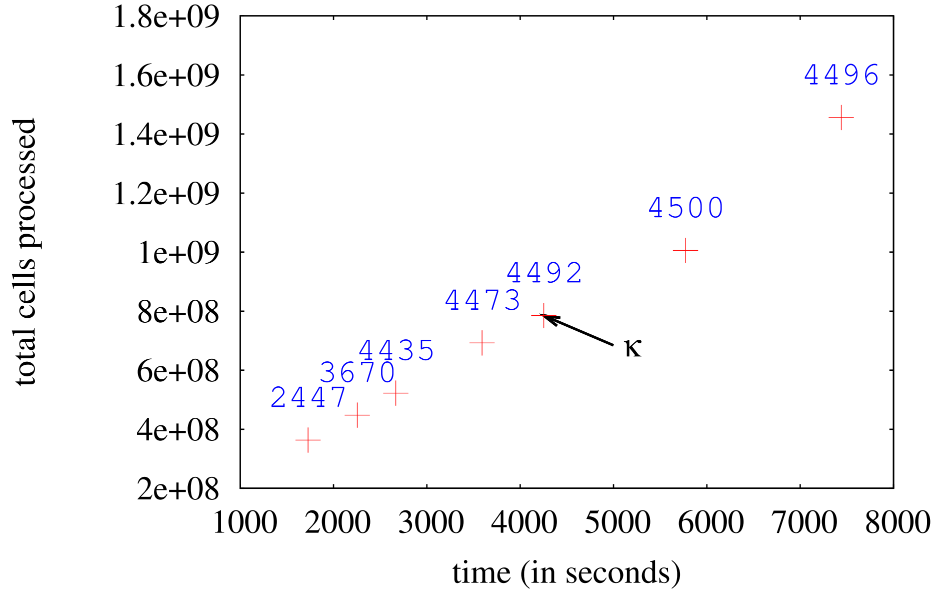


Figure 5.3: Run-time is proportional to the total number of cells processed.

Table 5.11 compares the speed of the string and binary versions of Algorithm SIDD run against the larger data sets. Times were averaged over ten runs and then normalised against the binary version, which was up to three times faster. File sizes of the binary data were smaller and working only with primitive data types meant that there was no expensive object manipulation necessary. In Table 5.12 we see that Kumar—Algorithm IPDD—was between 1.4 and 27 times slower than the binary implementation. As discussed in Section 4.2.4, we tested four alternative implementations of the SQL-based approach—Algorithm SQLDD. The first version (SQL1) deleted cells from a copy of the fact table. It had very poor performance—up to 175 times slower than our binary implementation for cube NF3. A second version (SQL2), one that updated a boolean-valued attribute when cells were pruned—instead of deleting the row—had times comparable to the string version for the smaller cubes, but its performance still lagged behind our binary implementation on the larger cubes. A third version (SQL3) rebuilt the table when 50% of the cells were marked for deletion. It did have improved running times, compared to SQL2, over the large cubes, so we implemented a modified version that rebuilt the table whenever 75% of the remaining cells were marked for deletion. This variation showed an improved performance of 10% over SQL3 for cube K6 and 40% for cube K8. However, SQL4 was still more than 25 times slower than our binary implementation on the large data cubes. Preprocessing for each SQL version are given in Table 5.13. Generally, building indexes significantly increases preprocessing time. Processing times are given in Table 5.14. We generated cubes K5, K6, K7 and K8 to see the difference in conclusions that handling more and more data would give. We see in Table 5.15 that the trend in slowdowns is consistent. The fluctuation in binary times is explained by the total number of cells processed for each cube—more than 106×10^6 for cube K5, 79×10^6 for cube K6, 103×10^6 for cube K7 and 68×10^6 for cube K8. It is not clear why the difference in slowdown between K5 and K6 jumps an order of magnitude for SQL2: the query plans executed by MySQL were identical. Also, experiments that reduced the amount of RAM available when processing

Table 5.12: Relative slowdown of algorithms compared to the binary version. Times were averaged over ten runs and then normalised against the binary execution times. SQL processing for NF1 was forcibly terminated after 26 hours ($\otimes$).

	TW1	TW2	NF1	NF3	C1	F1	K1
binary time (secs)	0.07	0.2	255	44	4.8	24	661
string slowdown	1.4	2.0	3.5	2.9	2	1.4	2.8
Kumar slowdown	1.4	9.5	15	—	27	17	—
SQL2 slowdown	0.85	2.5	$\otimes$	100	2	2.3	152

Table 5.13: Comparison of preprocessing times for SQL implementations. Times are averaged over 10 runs. All times reported are in seconds.

	SQL1	SQL2/SQL3/SQL4
TW1	4.0×10^{-1}	4.0×10^{-1}
TW2	8.0×10^{-1}	4.0×10^{-1}
NF3	2.5×10^{1}	8.8×10^{2}
C1	7.0×10^{-1}	5.0×10^{0}
C4	2.0×10^{1}	1.9×10^{2}
F1	1.5×10^{0}	4.1×10^{1}
W3	2.3×10^{0}	7.5×10^{1}
K5	2.2×10^{1}	1.1×10^{3}
K6	—	1.4×10^{3}
K7	—	1.8×10^{3}
K8	—	2.2×10^{3}

K5, did not result in a measurable difference in time to execute. However, more than twice as many iterations were required for K6 to converge as K5; at the same time 30% more data was processed.

We did not run SQL4 against cubes whose diamond retained more than 75% of their cells. To test the effect of different database engines on our code, we also implemented SQL4 for Microsoft SQLServer and compared it with a MySQL version. These experiments were carried out on a different machine than the others, and the results are presented in Appendix B.1.1. We found that MySQL was nearly twice as fast as SQL Server for this particular test.

Table 5.14: Comparison of SQL implementations. Times are averaged over 10 runs. For Cube K5[†] fewer than 50% of cells were marked for deletion so there was no table rebuild. All times reported are in seconds.

	SQL1	SQL2	SQL3	SQL4
TW1	3×10^{-1}	6×10^{-2}	—	—
TW2	7.7×10^{1}	5.0×10^{1}	—	—
NF3	7.7×10^{3}	4.5×10^{3}	4.8×10^{3}	4.9×10^{3}
C1	1.5×10^{2}	1×10^{1}	1.1×10^{1}	—
C4	5.9×10^{2}	2.5×10^{2}	2.3×10^{2}	2.8×10^{2}
F1	1.5×10^{3}	5.4×10^{1}	4.8×10^{1}	—
W3	1.7×10^{2}	6.1×10^{1}	5.8×10^{1}	5.6×10^{1}
K5	6.1×10^{3}	4.8×10^{2}	[†]4.8×10^{2}	—
K6	—	3.4×10^{3}	2.7×10^{3}	2.4×10^{3}
K7	—	4.6×10^{3}	3.8×10^{3}	3.2×10^{3}
K8	—	6.2×10^{3}	5.7×10^{3}	3.4×10^{3}

Table 5.15: Relative slowdown of the SQL algorithms compared to the binary implementation.

	K5	K6	K7	K8
binary time (secs)	150	91	120	65
SQL2 slowdown	3.2	38	39	95
SQL3 slowdown	3.2	30	32	88
SQL4 slowdown	—	26	27	53

5.8 Diamond Size and Dimensionality

In our real-data experiments the size (in cells) of the κ-diamond of the high-dimensional cubes is large, e.g. the κ-diamond for K3 captures almost 30% of the data. How can we explain this? Is this property a function of the number of dimensions? To answer this question the κ-diamond was generated for each of the synthetic cubes. Estimated κ, its real value and the size in cells for each cube are given in Table 5.16. The κ-diamond captures 99% of the data in cubes U1, U2 and U3—dimensionality has no effect on diamond size for these uniformly distributed data sets. Likewise, dimensionality did not affect the size of the κ-diamond for the skewed data cubes as it captured between 23% and 26% of the data in cubes S1, S2 and S3 and between 12% and 16.5% in the other cubes. These

Table 5.16: High dimensionality does not affect diamond size.

Cube	dimensions	iters	Estimated	κ	size (cells)	% captured
U1	3	6	89	236	982 618	98
U2	4	6	66	234	975 163	98
U3	10	7	25	229	977 173	98
S1	3	9	90	1141	227 527	24
S2	4	14	67	803	231 737	23
S3	10	14	25	208	260 864	26
SS1	3	18	11	319	122 878	12
SS2	4	19	7	175	127 960	13
SS3	10	17	1	28	165 586	17

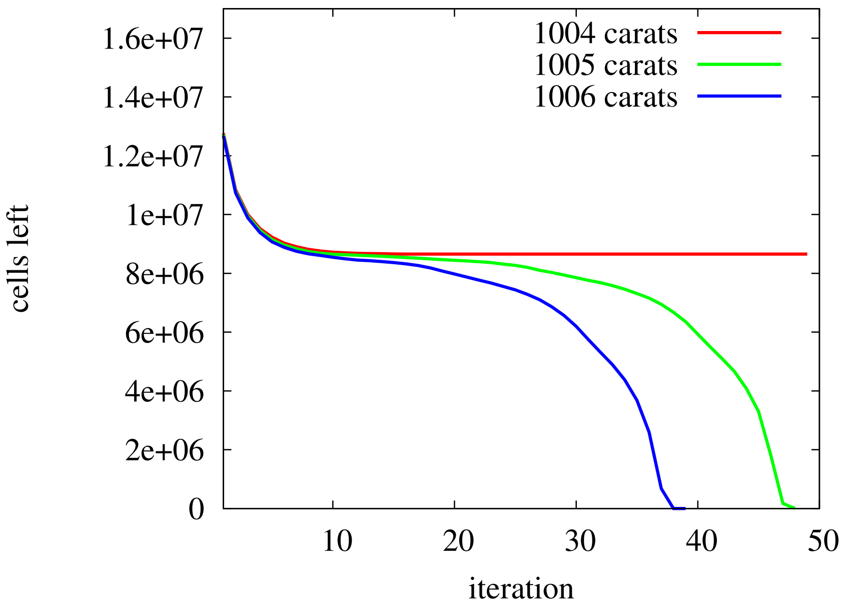
results indicate that the distribution of the data is more of a factor in determining how much of the data is captured by a diamond dice.

5.9 Iterations to Convergence

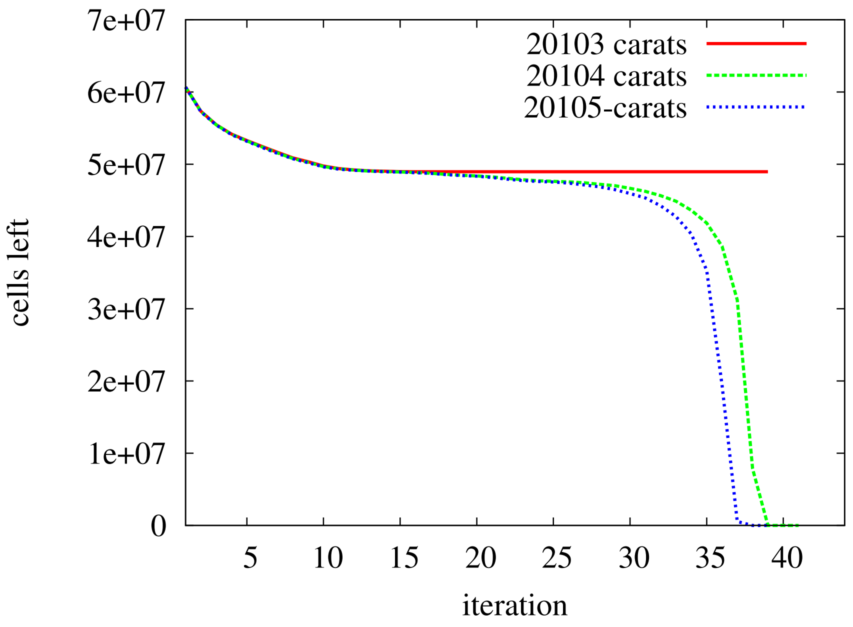
In Section 4.2 we observed that in the worst case it could take $\Theta(\sum_i n_i)$ iterations before the diamond cube stabilised. In practice this was not the case. (See Table 5.7, Table 5.8 and Figure 5.2.) All cubes converged to the κ-diamond in less than 1% of $\sum_i n_i$, with the exception of the small cubes TW1 and TW2 which took less than 7% $\sum_i n_i$. Algorithm SIDD required 19 iterations and 4 minutes[2] to compute the 1 004-carat κ-diamond for NF1 and it took 50 iterations and an average of 8 minutes to determine that there was no 1 005-carat diamond.

We measured the number of cells remaining in cube NF1 after each iteration of different k-diamonds to determine how quickly the diamond converges to an empty diamond when k exceeds κ. Figure 5.4a shows the number of cells present in the diamond after each iteration for 1 004–1 006 carats. The initial number of cells in each cube is not shown in the figure—84% of the cells were removed during the first iteration of NF1 and 46% were removed from W2—to include them would distort the plots. The curve for 1 006 reaches

[2]averaged over 10 runs



(a) Cube NF1, computing 1 004-, 1 005- and 1 006-carat diamonds.



(b) Cube W2, computing 20 103-, 20 104-, and 20 105-carat diamonds.

Figure 5.4: Cells remaining after each iteration of Algorithm SIDD.

zero first, followed by that for 1 005. Since $\kappa(\mathsf{NF1}) = 1\,004$, that curve stabilises at a nonzero value. We see a similar result for W2 in Figure 5.4b where $\kappa(\mathsf{W2})$ = 20 103. It takes longer to reach a critical point when k only slightly exceeds κ.

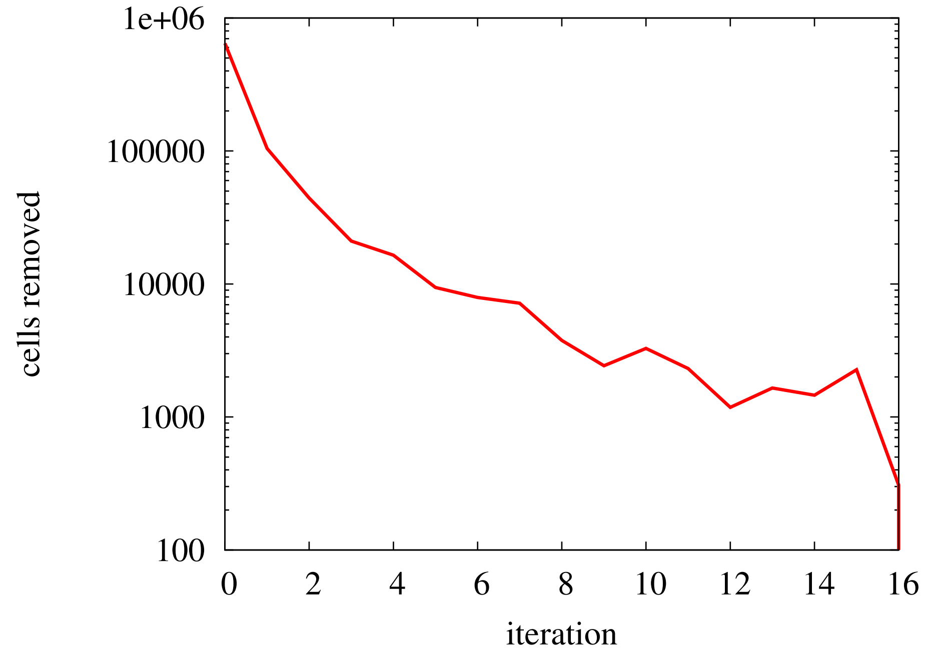
Intuitively, one should not be surprised that more iterations are required when $k \approx \kappa(C)$: attribute values that are almost in the diamond are especially sensitive to other attribute values that are also almost in the diamond.

The number of iterations required until convergence for all our synthetic cubes was also far smaller than the upper bound, e.g. cube S3: 35 616 (upper bound) and 14 (actual). We had expected to see the uniformly distributed data taking longer to converge than the skewed data (Section 4.2.2). This was not the case: in fact the opposite behaviour was observed. (See Table 5.16.) For cubes U1, U2 and U3 the diamond captured 98% of the cube: less than 23 000 cells were removed, suggesting that they started with a structure very like a diamond but for the skewed data cubes—S1, S2, S3, SS1, SS2 and SS3—the diamond was more "hidden". Figure 5.5 shows how many cells were removed each iteration from U1 and SS1. The figures are plotted on a log scale, since more than 73% of the cells were removed on the first iteration for both U1 and SS1.

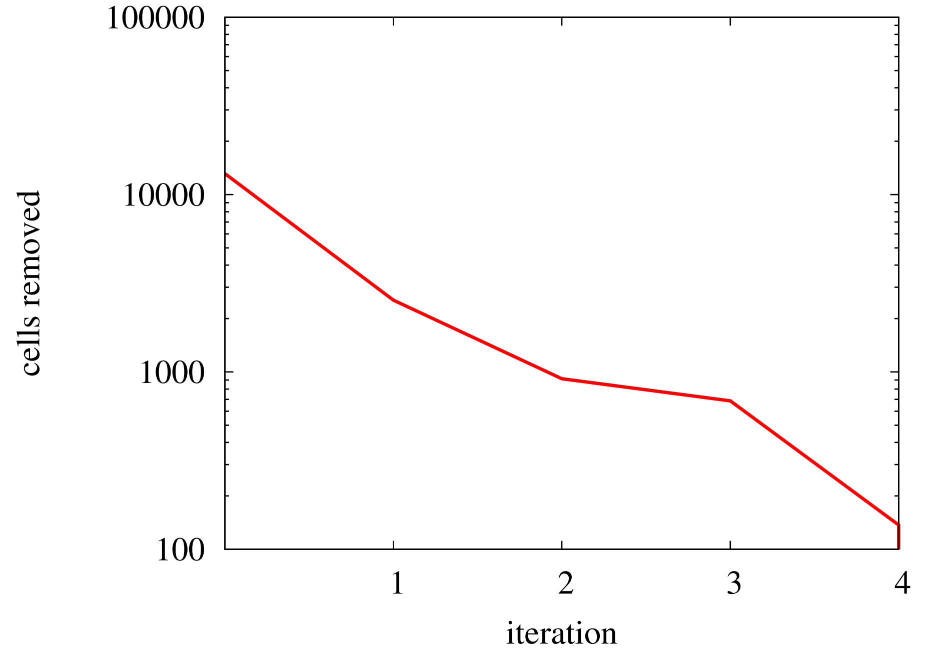
5.10 Robustness Against Randomly Missing Data

How robust is diamond dicing against random noise that models the data-warehouse problem [Thom 02] of missing data? In Section 3.6 we saw that, in the worst case, deallocation of a single cell could dramatically affect the diamond. How sensitive is κ to varying amounts of missing data in real data cubes?

We experimented with cubes TW1, F1, NF3 and SS2 and built new cubes, which are collectively labelled as TW1′, F1′, NF3′ and SS2′ in Table 5.17. Existing data points had an independent probability p_{missing} of being omitted from the data set. We determined κ for the modified cubes for 30 tests each with p_{missing} values between 1% and 5%. The



(a) Cube SS1



(b) Cube U1

Figure 5.5: Cells removed per iteration (log scale).

Table 5.17: Robustness of $\kappa(\mathsf{TW1})$ and $\kappa(\mathsf{SS2})$ under various amount of randomly missing data: for each probability, 30 trials were made. Each column is a histogram of the observed values.

$\kappa(\mathsf{SS2}')$	Prob. of cell's deallocation				
	1%	2%	3%	4%	5%
175					
174	1				
173	29				
172		6			
171		24			
170			12		
169			18		
168				22	
167				8	7
166					21
165					2
164					

(a) $\kappa(\mathsf{SS2})$ is 175 with no missing data

$\kappa(\mathsf{NF3}')$	Prob. of cell's deallocation				
	1%	2%	3%	4%	5%
270	28				
269	2				
268		3			
267		27			
265			8		
264			22		
263				3	
262				12	
261				15	
260					4
259					23
258					3

(b) $\kappa(\mathsf{NF3})$ is 272 with no missing data

$\kappa(\mathsf{TW1}')$	Prob. of cell's deallocation				
	1%	2%	3%	4%	5%
38	19	12	3	2	
37	10	17	17	10	4
36	1	1	10	16	18
35				2	7
34					1

(c) $\kappa(\mathsf{TW1})$ is 38 with no missing data

$\kappa(\mathsf{F1}')$	Prob. of cell's deallocation				
	1%	2%	3%	4%	5%
91	30				
90		30			
89			30	1	
88				29	1
87					29

(d) $\kappa(\mathsf{F1})$ is 92 with no missing data

results are presented in Table 5.17. In each case, $\kappa(C)$ of the modified cube was less than the original κ-value. For example, of the 30 runs for cube NF3 with 5% missing data, 23 times $\kappa(\mathsf{NF3}')$ was 259, 4 times it was 260 and for the remaining 3 it was 258: so $\kappa(\mathsf{NF3}')$ of each modified cube was within 6% of the true value (272). Although noise degraded the result for all cubes, the worst observed was for TW1 where $\kappa(\mathsf{TW1}')$ was only once more than 8% different than $\kappa(\mathsf{TW1})$. These results suggest that diamond dicing in general, and κ in particular, is robust against modest amounts of missing data.

Chapter 6

Related Problems

Finding dense subcubes in large data is a difficult problem [Lee 06, Barb 01, Ben 06a]. We investigate the role that diamond cubes might play in the solution of three NP-hard problems, each of which seeks a dense subcube, and show that the diamond—while perhaps not providing an exact solution—is a good starting point. The first problem, finding the Largest Perfect Subcube, assumes use of the aggregator COUNT whilst for the remaining problems—Densest Cube with Limited Dimensions and Heaviest Cube with Limited Dimensions—SUM is assumed.

6.1 Largest Perfect Cube

A *perfect* cube contains no empty cells and the perfect cube of size $(n_1 \times n_2 \times \cdots \times n_d)$ is a $k_1, k_2, \ldots, k_d$-diamond where each $k_i = \prod_{j=1, j \neq i}^{d} n_i$. In Proposition 6.1, we show that finding the largest perfect diamond is NP-hard. A motivation for this problem is found in Formal Concept Analysis. (See Section 2.4.3.) For example, consider the two-dimensional cube of Figure 4.1. The set of countries and actions remaining in a perfect diamond can be thought of as a concept—the largest set of countries (objects) where a variety of terrorist activities (attributes) occurred.

A **subcube** S, of a cube C, is obtained by applying the dice operator to C: specifying attribute values in one or more dimensions to retain in the subcube.

Proposition 6.1. *Finding a perfect subcube with largest volume is NP-hard, even in two dimensions.*

Proof. A two-dimensional cube is essentially an unweighted bipartite graph. Thus, a perfect subcube corresponds directly to a *biclique*—a clique in a bipartite graph. Finding a biclique with the largest number of edges has been shown NP-hard by Peeters [Peet 00], and this problem is equivalent to finding a perfect subcube of maximum volume. □

Finding a diamond might be part of a sensible heuristic to solve this problem. We show in the next lemma that if a large perfect subcube exists within a data cube, then there is also a nontrivial large diamond.

Lemma 6.1. *For* COUNT*-based carats, a perfect subcube of size $s_1 \times s_2 \times \cdots \times s_d$ is contained in the $(\prod_{i=1}^{d} s_i)/\max_i(s_i)$-carat diamond and in the $k_1, k_2, \ldots, k_d$-carat diamond, where $k_i = (\prod_{j=1}^{d} s_j)/s_i$.*

Proof. Clearly, the perfect subcube of size $s_1 \times s_2 \times \cdots \times s_d$ is a $k_1, k_2, \ldots, k_d$-carat subcube. Thus, it must be included in the $k_1, k_2, \ldots, k_d$ -carat diamond. □

This helps in two ways: if there is a nontrivial diamond of the specified size, $s_1 \times s_2 \times \cdots \times s_d$ we can search for the perfect subcube within it; however, if there is only an empty diamond of the specified size, there is no perfect subcube.

6.2 Heaviest Cube with Limited Dimensions

In the OLAP context, given a cube, a user may ask to "find a subcube with 10 attribute values per dimension." Meanwhile, he may want the resulting subcube to have maximal average—he is, perhaps, looking for the 10 attributes from each dimension that, in combination, give the greatest profit. We call this problem the HEAVIEST CUBE WITH LIMITED

DIMENSIONS (HCLD), which we formalise as: pick $\min(n_i, p_i)$ attribute values for dimension D_i, for all i's, so that the resulting subcube has maximal average, where n_i is the size of dimension i and p_i is the required number of attribute values for dimension i and unallocated cells are counted as zeros. This problem does not restrict the number of attribute values (p_i) to be the same for each dimension. We have that the HCLD must intersect with diamonds.

Proposition 6.2. *A cube without any non-empty $k_1, k_2, \ldots, k_d$-carat subcube has sum less than $\sum_{i=1}^{d}(n_i - 1)k_i + \max(k_1, k_2, \ldots, k_d)$ where the cube has size $n_1 \times n_2 \times \cdots \times n_d$.*

Proof. Suppose that a cube of dimension $n_1 \times n_2 \times \cdots \times n_d$ does not contain a non-empty $k_1, k_2, \ldots, k_d$-sum-carat cube. Such a cube must contain at least one slice with sum less than k_i, remove it. The remainder must also not contain a $k_1, k_2, \ldots, k_d$-sum-carat cube, remove another slice and so on. This process may go on at most $\sum_{i=1}^{d}(n_i - 1)$ times, at which point there is only one cell left. The maximum value this remaining cell can take is $\max(k_1, k_2, \ldots, k_d)$. Hence, the sum of the cube is less than $\sum_{i=1}^{d}(n_i - 1)k_i + \max(k_1, k_2, \ldots, k_d)$. $\square$

Corollary 6.1. *Any solution to the HCLD problem having size $s_1 \times s_2 \times \cdots \times s_d$ and average greater than*

$$\frac{\sum_{i=1}^{d}(s_i - 1)k_i + \max(k_1, k_2, \ldots, k_d)}{\prod_{i=1}^{d} s_i},$$

must intersect with the $k_1, k_2, \ldots, k_d$-SUM-carat diamond.

Proof. The denominator is the volume of the HCLD solution. From Proposition 6.2 we have that a cube without any $k_1, k_2, \ldots, k_d$-sum-carat cube has sum less than $\sum_{i=1}^{d}(s_i - 1)k_i + \max(k_1, k_2, \ldots, k_d)$. Therefore, a solution to the HCLD problem having average greater than

$$\frac{\sum_{i=1}^{d}(s_i - 1)k_i + \max(k_1, k_2, \ldots, k_d)}{\prod_{i=1}^{d} s_i}$$

must intersect with the $k_1, k_2, \ldots, k_d$-SUM-carat diamond. □

6.3 Densest Cube with Limited Dimensions

A special case of the HCLD problem, where all measures have the value 1, is the DENSEST CUBE WITH LIMITED DIMENSIONS. We formalise it as: pick $\min(n_i, p_i)$ attribute values for dimension D_i, for all i's, so that the resulting subcube is maximally dense, where n_i is the size of dimension i and p_i is the required number of attribute values for dimension i. Intuitively, a densest cube should at least contain a diamond. We proceed to show that a sufficiently dense cube always contains a diamond with a large number of carats. From Proposition 3.1 we have a cube that does not contain a $k_1, k_2, \ldots, k_d$-carat subcube has size at most $1+\sum_{i=1}^{d}(k_i-1)(n_i-1)$ and density at most $(1+\sum_{i=1}^{d}(k_i-1)(n_i-1))/\prod_{i=1}^{d} n_i$. We also have the following corollary to Proposition 3.1:

Corollary 6.2. *Any solution of the DCLD problem having density above*

$$\frac{1 + (k-1)\sum_{i=1}^{d}(\min(n_i, p_i) - 1)}{\prod_{i=1}^{d} \min(n_i, p_i)}$$

must share a non-empty k-carat cube in common with the k-carat diamond.

Proof. The denominator is the volume of the DCLD solution and from Corollary 3.1, we have that a cube having more than $1 + (k-1)\sum_{i=1}^{d}(n_i - 1)$ allocated cells, must contain a k-carat cube. We also have from the union property of diamonds, Proposition 3.1 that any cube with k carats is a subcube of the k-diamond. Thus, the intersection between the DCLD solution and the k-carat diamond cube contains a k-carat cube. □

1	1	1	1	1					
1	1	1	1	1					
1	1								
1	1								
1	1								
					1	1	1		
						1	1	1	
							1	1	1
					1			1	1
					1	1			1

Figure 6.1: Example showing that a diamond (bottom-right quadrant) may not have optimal density.

When $n_i \geq p$ for all i, then the density threshold of Corollary 6.2 is $(1+(k-1)d(p-1))/p^d$: this value goes to zero exponentially as the number of dimensions increases. Thus, as the number of dimensions increases, a DCLD solution is guaranteed to intersect with the k-carat diamond.

We might hope that when the dimensions of the diamond coincide with the required dimensions of the densest cube, we would have a solution to the DCLD problem. Alas, this is not true. Consider the 2-D cube in Figure 6.1. The bottom-right quadrant forms the largest 3-carat subcube. In the bottom-right quadrant, there are 15 allocated cells whereas in the upper-left quadrant there are 16 allocated cells. This proves the next result.

Lemma 6.2. *Given a cube with size $n_1 \times n_2 \times \cdots \times n_d$ and required dimension sizes $p_1, p_2, \ldots, p_d$, even if a* COUNT*-based diamond has exactly $\min(n_i, p_i)$ attribute values for dimension D_i, for all i's, it may still not be a solution to the DCLD problem.*

We are interested in large data sets; the next proposition shows that solving DCLD and HCLD is difficult.

Proposition 6.3. *The DCLD and HCLD problems are NP-hard.*

Proof. The EXACT BALANCED PRIME NODE CARDINALITY DECISION PROBLEM (EBP-NCD) is NP-complete [Dawa 01]—for a given bipartite graph $G = (V_1, V_2, E)$ and a prime number p, does there exist a biclique U_1 and U_2 in G such that $|U_1| = p$ and $|U_2| = p$?

Given an EBPNCD instance, construct a 2-D cube where each value of the first dimension corresponds to a vertex of V_1, and each value of the second dimension corresponds to a vertex of V_2. Fill the cell corresponding to $v_1, v_2 \in V_1 \times V_2$ with a measure value if and only if v_1 is connected to v_2. The solution of the DCLD problem applied to this cube with a limit of p will be a biclique if such a biclique exists. $\square$

6.3.1 A Related Graph Problem

A graph problem that is closely related to both DCLD and HCLD is the DENSE SUBGRAPH problem [Suzu 05,Geor 07]. The NP-hard bipartite DENSE SUBGRAPH problem is defined as follows: for a bipartite graph ($G = (U, V, E)$) where each edge e has a non-negative weight $w(e)$, given m_1 and m_2, find a subset X of U of size m_1 and a subset Y of V of size m_2 such that the sum of the weights of the subgraph (X, Y, E) is maximised. The unweighted dense subgraph problem has $w(e) = 1$ for each edge.

The DENSE SUBGRAPH problem is two-dimensional. The DCLD and HCLD problems are not restricted to two dimensions. Finding the densest subgraph can be solved in polynomial time [Gall 89] and the problem becomes NP-hard only when you specify how large the graph should be. The HCLD and DCLD problems require us to specify the number of attribute values that should be present in any solution and the DENSE SUBGRAPH problem also requires us to specify the size of the solution. The two-dimensional versions of DCLD and HCLD are equivalent to the unweighted and weighted DENSE SUBGRAPH problems, respectively.

6.4 Effectiveness of DCLD Heuristic

The DCLD problem is NP-complete and determining the quality of a heuristic over even moderately-sized data cubes poses difficulties. Therefore, two different approaches were taken. In the first, an exact solution was computed for two small, synthetically-generated three-dimensional cubes and compared with the solutions from

- a Monte Carlo Heuristic (MCH)—Algorithm 6.1
- the diamond-based heuristic, DCLD From Diamond (DCLDFD)—Algorithm 6.2
- a hybrid, Monte Carlo From Diamond (MCFD), which only differs from Algorithm 6.1 in that it takes a diamond cube as input instead of the whole data cube.

```
input: $d$-dimensional cube $C$, integers $p_1, p_2, \ldots, p_d, n$
output: Cube with size $p_1 \times p_2 \times \cdots \times p_d$
// assume $p_i \leq n$
$\Delta \leftarrow C$
foreach dimension $i$ do
    Sort slices of dimension $i$ of $\Delta$ by their COUNT values
    Retain only the top $p_i$ slices and discard the remainder from $\Delta$
end
for $t \leftarrow 1$ to 1000 do
    bestAlternative $\leftarrow$ COUNT($\Delta$)
    bestCube $\leftarrow \Delta$
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $d$ do
            // Randomly choose an element to swap in each
               dimension
            chosen $\leftarrow$ random value $v$ of dimension $j$ that has been retained in $\Delta$
            unchosen $\leftarrow$ random value $w$ from dimension $j$ in $C$, but where $w$ is
            not in $\Delta$
            Form $\Delta'$ by temporarily adding slice unchosen and removing slice
            chosen from $\Delta$
            if COUNT($\Delta'$) $>$ bestAlternative then
                bestCube $\leftarrow \Delta'$
                bestAlternative $\leftarrow$ COUNT($\Delta'$)
            end
        end
    end
end
return bestCube
```

Algorithm 6.1: Monte Carlo DCLD heuristic (MCH).

Second, the heuristics were applied to cubes TW1 and F1. To compute an exact solution for either of these cubes would be prohibitively time consuming.

```
input: $d$-dimensional cube $C$, integers $p_1, p_2, \ldots, p_d$
output: Cube with size $p_1 \times p_2 \times \ldots \times p_d$
// Use binary search to find $k$
Find max $k$ where the $k$-carat diamond $\Delta$ has shape $p'_1 \times p'_2 \times \cdots \times p'_d$, where
$\forall i p'_i \geq p_i$
for $i \leftarrow 1$ to $d$ do
    Sort slices of dimension $i$ of $\Delta$ by their $\sigma$ values
    Retain only the top $p_i$ slices and discard the remainder from $\Delta$
end
return $\Delta$
```
Algorithm 6.2: DCLD heuristic that starts from a diamond (DCLDFD).

The first synthetic three-dimensional cube DCLD1 has 92 allocated cells, with attribute values in the range 0 to 11 for each dimension. It was generated by simulating a roll of a loaded 12-sided die d (=3) times, to get the coordinates of an allocated cell. The probability of getting a 0 is $\frac{10}{3}\%$, as is the probability of getting a 1 or of getting a 2. For 3, or for any larger number, the probability of getting that number is 10%. The values were generated using Algorithm C.1. In all, 100 cells were generated and duplicate cells were removed. (See Appendix C.1.)

The exact solution to the DCLD($p = 8$) problem was found for cube DCLD1 by enumerating all possible cubes of the chosen size and keeping cubes with the maximum density. The solution, which took over 24 hours to process, is a unique 60-cell cube with density 0.1172. The κ-diamond was established as the diamond that most closely satisfied the DCLD constraint, with 9, 8 and 9 values for dimensions 1, 2 and 3 respectively. For DCLD1 the κ-diamond contained the exact solution. DCLDFD captured 59 cells. A refinement to the algorithm that allows for replacing attribute values when there are equally likely candidates, did find the exact solution. The three bottom ranked attribute values for dimension 1 had cardinality 5, so were equally likely candidates; likewise, for dimension 3, the bottom 2 ranked attribute values. By choosing each of these attribute values in turn (6 combinations) and checking the resulting cube's density, the exact solution was found. The exact solution to DCLD($p = 8$) was found at a cost of generating 6 cubes in contrast to the 121 287 375 cubes generated by the brute force method.

Table 6.1: Comparison of DCLD algorithms.

cube	algorithm	cells
DCLD1	exact solution	60
	DCLDFD (0 swaps) – Algorithm 6.2	59
	DCLDFD (6 swaps)	60
	MCFD (1 000 iters)	59
DCLD2	exact solution	59
	DCLDFD (0 swaps)	59
	MCFD (1 000 iters)	59
TW1	DCLDFD (0 swaps)	179
	MCFD (1 000 iters)	179
F1	DCLDFD (0 swaps)	151
	DCLDFD (3 swaps)	153
	MCFD (10 iters)	151

A second three-dimensional cube, DCLD2, was generated using the same method as for DCLD1. The die weights: the probability of getting a 0 was $\frac{50}{3}\%$, as was the probability of getting a 1 or of getting a 2. For 3, or for any larger number, the probability of getting that number was 5.5%. Again 100 cells were generated, and this time six duplicates were removed before processing. (See Appendix C.2.) Fourteen exact solutions to the DCLD($p = 8$) were generated using the brute force method, emphasising that the DCLD problem need not have a unique solution. Eleven of the fourteen solutions were contained within the κ-diamond, illustrating also that the diamond need not coincide with a DCLD solution.

MCH gave poor solutions for cubes DCLD1 and DCLD2, capturing less than 80% of the exact solution after 1 000 iterations. It was not considered again as a potential heuristic for this problem.

It is infeasible to compute an exact solution to the DCLD problem, for all but the smallest values of p, for cubes with non-trivial dimension cardinalities. However, applying the heuristics to two real data cubes, TW1 and F1, we find that using the diamond as a starting point gives a better solution than one generated using this Monte Carlo method. Results for DCLD(p=90) on F1 and DCLD(p=5) on TW1 are given in Table 6.1.

MCFD, which uses the κ-diamond as its starting point, does not improve the result of DCLDFD. Furthermore, it is more expensive to run. An execution of DCLDFD requires a single pass over the data cube, unless there are equally like candidates in the diamond, whereas MCFD executes a pass over the data for each iteration. DCLDFD finds an acceptable solution with little work. On `F1` the time difference was significant: 17 seconds for DCLDFD with 3 swaps and 43 seconds for MCFD. Running MCFD for 1 000 iterations on this cube would take a very long time—over an hour—and there is no guarantee that the effort would result in an improved solution.

In conclusion, we have shown that the diamond cube is a good approximation for a solution to the DCLD problem. The results of our experiments also show that diamonds provide a speedier solution compared with these Monte Carlo algorithms, whilst giving as good or better accuracy.

Chapter 7

Conclusion

This dissertation establishes properties of the diamond and defines union and intersection operations over diamonds. Algorithms to extract diamonds have been designed, implemented and tested on real as well as artificially constructed data with predefined distributions. We have also investigated the role diamonds might play in the solution of three NP-hard problems that seek dense subcubes.

Properties of Diamonds We presented a formal analysis of the diamond cube. We have shown that, for the parameter k associated with each dimension in every data cube, there is only one $k_1, k_2, \ldots k_d$-diamond. By varying the k_i's we get a collection of diamonds for a cube and these diamonds form a lattice. We established upper and lower bounds on the parameter k for both COUNT and SUM-based diamond cubes.

Experiments We have designed, implemented and tested algorithms to compute diamonds on real and synthetic data sets. Experimentally, the algorithms bear out our theoretical results. An unexpected experimental result is that the number of iterations required to process the diamonds with k slightly greater than $\kappa(C)$ is often twice that required to process the κ-diamond, which also resulted in an increase in running time.

When executed against large data sets, relational database or OLAP operators did not

provide a timely solution to the multidimensional queries presented in this work. The binary version of the Java code was more than 25 times faster than our most efficient SQL implementation. Further, the algorithm and the Java implementation are simple, requiring less than 275 lines of code.
Our experimental results suggest that diamond dicing and the statistic $\kappa(C)$ are robust against missing data.

Related Problems We have proved a relationship between diamonds and the solutions to three NP-Complete problems:

- **Largest Perfect Cube.** If a large perfect subcube of size $s_1 \times s_2 \times \ldots, s_d$ exists, it is contained in the $k_1, k_2, \ldots, k_d$-carat diamond, where $k_i = (\prod_{j=1}^{d} s_j)/s_i$.
- **Heaviest Cube with Limited Dimensions** Any solution to the HCLD problem having size $s_1 \times s_2 \times \cdots \times s_d$ and average greater than

  $$\frac{\sum_{i=1}^{d}(s_i - 1)k_i + \max(k_1, k_2, \ldots, k_d)}{\prod_{i=1}^{d} s_i},$$

  intersects with the $k_1, k_2, \ldots, k_d$-SUM-carat diamond.
- **Densest Cube with Limited Dimensions** Any solution of the DCLD problem having density above

  $$\frac{1 + (k-1)\sum_{i=1}^{d}(\min(n_i, p_i) - 1)}{\prod_{i=1}^{d} \min(n_i, p_i)}$$

  must share a non-empty k-carat cube in common with the k-carat diamond.

Our experiments with the diamond-based heuristic to solve the DCLD problem show that the diamond cube is a good approximation for the solution.

7.1 Future Research Directions

In the course of diamond cube development and evaluation, several potential avenues for future work have presented themselves. We have identified three broad areas: faster computation, generalisations and applications of diamonds, together with a couple of variations: fractional diamonds and the $k-$plex. A brief overview of these ideas follows.

7.1.1 Faster Computation

Although it is faster to compute a diamond cube using our implementation than using the standard relational DBMS operations or OLAP operators, the speed does not conform to the OLAP goal of near constant time query execution. Different approaches could be taken to improve execution speed: compress the data so that more of the cube can be retained in memory; use multiple processors in parallel; or, if an approximate solution is sufficient, we might process only a sample of the data.

Parallel Computation and Compression First, a parallel implementation [Chen 05] making use of more than one processor to compute the diamond cube, perhaps coupled with cube compression [Hami 07, Holl 07] may result in an overall speed-up. In a shared-memory implementation, the data cube could be partitioned across multiple processors with the values of $\sigma(\text{slice}_j(D_i))$ and a flag—indicating a modification—maintained in the shared memory area. Each processor could compute the diamond for its partition. Little communication between processors would be necessary to amalgamate the diamond when the algorithm terminates.

Typically, shared memory architectures are limited to a few processors. An alternative implementation on a "shared-nothing" architecture would allow more processors to execute in parallel. Updates to the diamond would need to be communicated and there would be the inevitable trade-off between time gained by increasing the number of processors, and the extra time required to communicate intermediate results between the processors. We could also exploit the lattice by precomputing some key elements using parallel processing and then refine the solution to the required diamond.

Approximate Diamonds Second, can we predict the size of a diamond of a given carat number while examining only a sample of the data cube? Sampling is one technique that has been used to provide approximate answers to long-running OLAP queries [Babc 03, Vitt 98] If we make assumptions about the distribution of the data can we get a better guess? Missaoui et al. [Miss 07] advocate using probabilistic models to query data cubes.

7.1.2 Generalisations of Diamonds

We have identified three new possible generalisations of diamonds: diamonds of higher order; diamonds with pinned attributes and diamonds allowing a limited number of negative measures.

Diamonds of higher order We consider only diamonds where each slice holds an attribute value constant from a single dimension. However, this concept could be generalised to slices of order δ where a slice is the cube obtained when a single attribute value is fixed in each of δ different dimensions. For example, a slice of order 0 is the entire cube, a slice of order 1 is the more traditional definition of a slice [OLAP] and so on. For a d-dimensional cube, a slice of order d is a single cell.

Diamonds are related to *iceberg cubes* and we could recover them by obtaining the slice of order d where $\sigma(x)$ returns the measure corresponding to cell x. The iceberg query, introduced by Fang et al. [Fang 98], performs an aggregate function over a specified dimension list and then eliminates aggregate values below some specified threshold. Given a relation $R(\mathrm{attr_1}, \ldots, \mathrm{attr_m}, \mathrm{rest})$, the typical iceberg query has the form of Query 7.1. Consider the Inventory relation of Figure 7.1. Suppose we are interested in the vehicles of the same make and model in our inventory, but only those cases where there is more than one available. An iceberg query and its result are given in Query 7.2.

```
SELECT attr$_1$,...,attr$_m$, count(rest)
FROM  R
GROUP BY attr$_1$,...,attr$_m$
HAVING count(rest) > threshold
```

Query 7.1: Typical iceberg query.

Inventory		
colour	make	model
red	Subaru	Forester
red	Toyota	Camry
red	Subaru	Outback
blue	Toyota	Corolla
blue	Mazda	Miata
blue	Subaru	Forester
silver	Mazda	626
silver	Mazda	Mazda 3
silver	Toyota	Corolla

Figure 7.1: Inventory relation.

This result could be achieved with a diamond dice. Requiring that all slices of order d have a sum of k would result in the iceberg that contains cells whose measures are at least k.

Pinned Attribute Values Perhaps an analyst is interested in particular attribute values in one or more dimensions and a straightforward enhancement to the diamond cube algorithm would allow an analyst to require that attribute values of interest in one or more dimensions not be pruned from the diamond. Perhaps the analyst is interested in the greatest profit achieved over a particular set of stores and products. He could "pin" the stores and products of interest and then execute a diamond dice over his chosen subset of the data

```
SELECT make, model, count(Colour)
FROM Inventory
GROUP BY make,  model
HAVING count(Colour) ≥ 2
```

Query result		
make	model	count(colour)
Subaru	Forester	2
Toyota	Corolla	2

Query 7.2: Iceberg query applied to the Inventory relation of Figure 7.1 and its result.

cube. To illustrate, we return to the example provided in Example 1.1: A company wants to close shops and terminate product lines simultaneously. The CEO wants the maximal set of products and shops such that each shop would have sales greater than \$400 000, and each product would bring revenues of at least \$100 000—assuming we terminate all other shops and product lines. Further, suppose that the CEO's nephews run certain stores that cannot be closed, and the CEO's nieces have financial interests in some of the product lines. These stores and product lines must be kept; given that, how best to terminate unconstrained product lines and stores?

Solution to a Restricted kSCP Can kSCP be solved with the restriction that only a constant number of measures are negative? If the number of negative measures is limited, then each potential removal/reinstatement of attribute values might be checked in a reasonable time. There may still not be a unique solution to a restricted kSCP, but can we bound the number of negative measures such that the problem is tractable?

7.1.3 Applications

The diamond identifies an important subset of a data cube and may be useful to supplement analysis in different areas: as a preprocessor to prune less frequently used items before applying more traditional data mining techniques, as a way to decompose a data cube, or as a means of discovering important or closely associated words in a text.

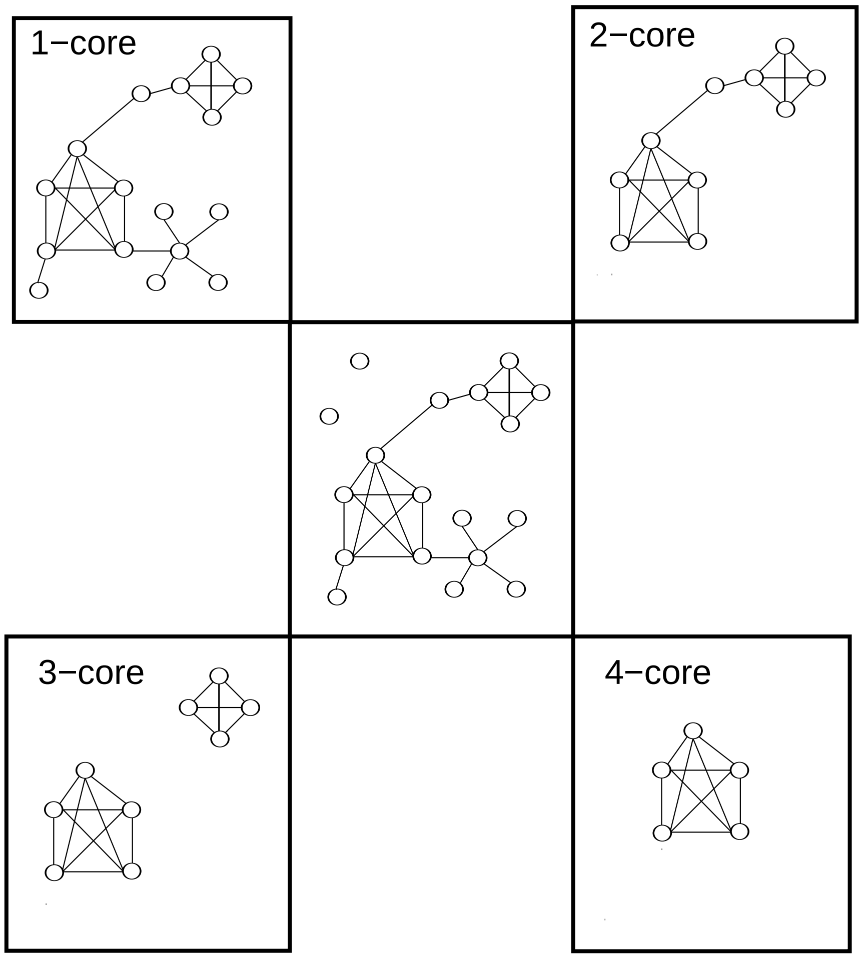
Diamonds and Association Rules Mining One difficulty experienced by data mining applications is that the search space is very large. Diamond dicing has potential as a preprocessing step for association rules mining. It would provide a means to prune away less important attributes. Association rules mining is well-studied [Agra 94, Lian 05] and it is incorporated into commercial data mining applications [SAS, IBM]. It is often used in conjunction with market basket data—a set of transactions where each transaction is a set of items bought by an individual customer. Association rules mining generates a set of rules with user-supplied parameters *support* and *confidence*. Given I the set of items occurring in the set of transactions, D, an association rule is an implication of the form $X \to Y$, where $X \subseteq I$, $Y \subseteq I$, and $X \cap Y = \phi$.

The rule $X \to Y$ holds in the transaction set D with confidence c if $c\%$ of transactions in D that contain X also contain Y. The rule $X \to Y$ has support s in the transaction set D if $s\%$ of transactions in D contain $X \cup Y$. One of the main disadvantages of association rules mining is the often vast number of rules that are generated. It is insufficient to increase the confidence and support thresholds to reduce the number of rules. For example, by increasing the support threshold, we might reduce the number of itemsets—sets of $X \cup Y$—from which the rules are generated but that, of itself, does not guarantee that the number of rules generated will be small enough for easy analysis [Hipp 00].

One option for presenting the most relevant rules to the analyst involves pruning the data, so that the rules are generated from the part of the cube of interest. The user is asked to define the subcube on which the rules mining should execute. With this smaller area to mine, the run time of the mining process is reduced [Ben 06b] and although the number of rules generated may not be fewer, they will be taken from the subcube of interest to the analyst. The diamond dice operator prunes less frequently used attributes from a data cube. It can also be used to define a subcube over which association rules are generated. One conjecture to investigate is that the sets of attributes remaining in the $k_1, k_2, \ldots, k_d$-diamond suggest a level of rule-support: each attribute is related to k_i combinations of attributes in other dimensions.

Diamond cores In network analysis, one topological property of interest [Zhou 07] is the k-core, removing nodes of degree less than k—in the same way that the diamond dice removes attribute values whose $\sigma(\text{slice})$ falls below k. The network is modelled as an undirected graph, unlike in Kumar et al.'s work [Kuma 99],where the network is modelled as a directed graph. In Zhou et al.'s approach, nodes are assigned to their c-shells, i.e. a node belongs to a c-shell if it has degree c and it is connected to other nodes

Figure 7.2: k-cores of a network [Zhou 07].



of degree c. (See Figure 7.2.) The k-core decomposition of a network disentangles its hierarchical structure by progressively focusing on its central cores. Similarly, a data cube could be decomposed by removing the largest core (κ-diamond); find the largest core in the remainder . . . ; continue until all attribute values have been assigned into one core or another. This concept has applications in social networking and text mining also [Feld 07].

Language Translation We saw in Section 2.5.1 that the diamond operator could be used as an alternative to a top-k query for selecting items to present in a tag cloud. Similarly, can the diamond help with language translation or other text analysis? Is there a data cube for which the κ-diamond represents an important subset of words that can be used to deduce a translation? For example, there is still much debate about how to decipher the symbols found on tablets from the Indus River civilisation, *circa* 3000 BC [Raoa 09]. Is there a way to build a cube such that a κ-diamond links words that are the most important/frequent? We have seen that the diamond dice identified a specific verse in the Bible when the κ-diamond was generated from a co-occurrence cube in Section 5.5.1. For another example, evidence-based medicine is an approach to medical research that can involve the systematic review of published articles using machine learning techniques [Kouz 09]. Is there a role for diamonds in processing the co-occurrence matrices used as one data representation scheme for this systematic review?

7.1.4 Other Ideas

Fractional Diamonds A potential extension to the diamond dice is a 'fractional' diamond, where the slice density meets a given constraint α. Current implementations of the diamond dice do not consider boolean dimensions any differently than those of higher cardinality and it may be beneficial to apply a fractional diamond dice to cubes with boolean or other low-cardinality dimensions. However, this might present challenges similar to the k-SUM Cube Problem: it is not obvious that there would be a unique solution.

k-plex In social network analysis, one way to relax the assumptions of a clique is to allow an actor to be a member of a *k-plex* if they have ties to all but k other members [Hann 05]. A similar concept could be applied to diamond dicing—a slice is permitted if it has no more than k *empty* cells. Is this some kind of dual to our diamond?

Glossary

allocated cell	the measure of an allocated cell is a value $v \in \mathbb{R}$	25
attribute values	the set of names that comprise the n_i elements of dimension i	23
carat	a cube has k_i carats over dimension D_i, if for every non-empty slice x along dimension D_i, $\sigma(x) \geq k_i, i \in \{1, 2, \ldots, d\}$.	28
cell	a cell of cube C is a 2-tuple $((x_1, x_2, \ldots, x_d) \in D_1 \times D_2 \times \ldots \times D_d, v)$ where v is a measure and $v = f(x_1, x_2, \ldots, x_d)$	25
compatible	if $f_A(x) \neq \perp$ and $f_B(x) \neq \perp$ then $f_A(x) = f_B(x)$	28
cube	a cube is the 2-tuple (D, f): the set of dimensions $\{D_1, D_2, \ldots, D_d\}$ together with a total function (f) which maps tuples in $D_1 \times D_2 \times \ldots \times D_d$ to $\mathbb{R} \cup \{\perp\}$, where $\perp$ represents undefined.	25

diamond	the diamond is $\bigcup_{A \in \mathcal{A}} A$, where $\mathcal{A}$ is the set of all cubes (of some starting cube C) having $k_1, k_2, \ldots, k_d$ carats.	29
dice	defines a subcube S by specifying attribute values $\widehat{D}_1, \widehat{D}_2, \ldots, \widehat{D}_d$ where each $\widehat{D}_i \subseteq D_i$. The cells in the slices corresponding to the unspecified attribute values are deallocated	25
dimension	the set of attribute values that define one axis of a multidimensional data structure	23
lattice	a partially ordered set in which any two elements have a unique supremum—least upper bound—and an infimum—greatest lower bound	30
maximal	a $k_1, k_2, \ldots, k_d$-carat cube is **maximal** when it is not contained in a larger cube having $k_1, k_2, \ldots k_d$ carats over dimensions $D_1, D_2 \ldots, D_d$	29
monotonically non-decreasing	An aggregator σ is monotonically non-decreasing if $S' \subset S$ implies $\sigma(S') \leq \sigma(S)$.	26
monotonically non-increasing	An aggregator σ is monotonically non-decreasing if $S' \subset S$ implies $\sigma(S') \geq \sigma(S)$.	26
perfect	a perfect cube contains only allocated cells	14, 34

slice	the cube obtained when a single attribute value is fixed in one dimension.	25
subcube	A subcube S, of a cube C, is obtained by applying the dice operator to C: specifying attribute values in one or more dimensions to retain in the subcube	89
unallocated cell	the measure of an unallocated cell is set to $\perp$	25

Appendices

Appendix A

Real Data Cubes

Details of the data cubes, and how they were extracted from the raw data, are given in the following sections.

A.1 Census

Census data were downloaded from http://kdd.ics.uci.edu/ [Hett 00] in comma-separated format. The 27 dimensions used in cubes C1 and C2 were extracted using the Unix awk utility and the commands used to extract cube C1 and to remove duplicates are given in Command A.1 and Command A.2. The shaded dimensions of Table A.1 are those used in our experiments. A comma-separated file that incorporated the measures, wage per hour and dividends from stock, was loaded into a MySQL database.

Command A.1 Awk command to extract cube C1

```
awk -F',' '{print $1","$2","$3","$4","$5","$7","$8",
"$9","$10","$11","$12","$13","$14", "$15", "$16","$20",
"$21","$22","$23","$24","$26","$27","$28","$29","$30",
"$31"," $40","}' census-income.data > census_count.csv
```

Table A.1: Census Income data: dimensions and cardinality of dimensions. Shaded dimensions and measures are those retained for cubes C1, C2 and C3. Dimension numbering maps to those described in the file *census-income.names* [Hett 00].

dim	dim cardinality	
d0	91	age
d1	9	class of worker
d2	52	detailed industry recode
d3	47	detailed occupation recode
d4	17	education
d6	3	enrol in edu inst last wk
d7	7	marital stat
d8	24	major industry code
d9	15	major occupation code
d10	5	race
d11	10	Hispanic origin
d12	2	sex
d13	3	member of a labour union
d14	6	reason for unemployment
d15	8	full or part time employment stat
d19	6	tax filer stat
d20	6	region of previous residence
d21	51	state of previous residence
d22	39	detailed household and family stat
d23	8	detailed household summary in household
d25	10	migration code-change in msa
d26	9	migration code-change in reg
d27	10	migration code-move within reg
d28	3	live in this house 1 year ago
d29	4	migration prev res in sunbelt
d30	7	num persons worked for employer
d39	53	weeks worked in year
	measures used for C2 and C3	
d5	1240	wage per hour
d18	1478	dividends from stocks
	unused dimensions	
d39	2	year
d16	132	capital gains
d17	113	capital losses
d24	99800	unknown
d30	5	family members under 18
d31	43	country of birth father
d32	43	country of birth mother
d33	43	country of birth self
d34	5	citizenship
d35	3	own business or self employed
d36	3	fill inc questionnaire for veteran's admin
d37	3	veterans benefits

Command A.2 Remove duplicate rows from census data.

```
sort census_count.csv | uniq > census_count_u.csv
```

A.2 TWEED

The TWEED data set was downloaded from http://folk.uib.no/sspje/tweed.htm [Enge 07] in SPSS format. It was transformed using SPSS to a comma-separated flat file, which was the required input format for all our implementations. Details of the dimensions used are given in Table A.2. The CSV file was loaded into a MySQL database.

Table A.2: Measures and dimensions of TWEED data. Shaded dimensions are those retained for TW1. All dimensions were retained for cubes TW2 and TW3 (with total people killed as the measure for TW3).

Dimension		Dimension cardinality
d1	Day	32
d2	Month	13
d3	Year	53
d4	Country	16
d5	Type of agent	3
d6	Acting group	287
d7	Regional context of the agent	34
d8	Type of action	11
d31	State institution	6
d32	Kind of action	4
d33	Type of action by state	7
d34	Group against which the state action is directed	182
d50	Group's attitude towards state	6
d51	Group's ideological character	9
d52	Target of action	11
Measure		
d49	total people killed	
people from the acting group	military	police
civil servants	politicians	business executives
trade union leaders	clergy	other militants
civilians		
total people injured		
acting group	military	police
civil servants	politicians	business
trade union leaders	clergy	other militants
civilians		
total people killed by state institution		
group members	other people	
total people injured by state institution		
group members	other people	
arrests	convictions	executions
total killed by non-state group at which the state directed an action		
people from state institution	others	
total injured by non-state group		
people from state institution	others	

A.3 Forest

Cube F1 was extracted from forestcover.dat, a file created by Liu [Liu 02] from Forest CoverType stored in the UCI KDD archive [Hett 00] The first eleven columns were copied into ForestCover.csv, which was then sorted and duplicates removed:

Command A.3 Sort and remove duplicates.

```
 sort -n -t ',' -u ForestCover.csv > fcover_unique.csv
```

We used the command `sed` to remove lines parentheses, ampersands and periods. The resulting file has 580 950 lines.

A.4 King James Bible

Unix utilities were used to extract a fact table from the 4-gram and 10-gram files (kj4d.csv and kj10d.csv) [Kase 08] generated by the kingjames script—kingjames.py—described in Section 5.3.1. First kingjames4d.csv was sorted and the unique records piped to kj4d.csv (Command A.4).

Command A.4 Unique records from kingjames4d.csv sorted and piped to kj4d.csv

```
sort kingjames4d.csv | uniq -c > kj4d.csv
```

The '-c' flag recorded a count for each instance of a line. This count was used as the measure in the sum-diamond experiments on cube K1. Further, it was necessary to move the count value to the final field of the fact table in preparation for loading into the MySQL database—all fact tables were assumed to have the format *attr0,attr1,...,attrd,measure*. All commas were replaced by spaces (Command A.5).

Command A.5 All commas replaced by spaces

```
sed -i 's/\,/\ /g' kj4d.csv
```

Then the fields containing data and the measure were written to a new file (Command A.6).

Command A.6 Data and measure fields written to kj4dWithMeasures.csv

```
awk -F ' ' '{print $2","$3","$4","$5","$1","}' kj4d.csv
> kj4dWithMeasures.csv
```

A similar approach was employed for the 10-dimensional version.

A.5 Netflix

The Netflix training set was downloaded from http://www.netflixprize.com [Netf 07]. It comprises 1 770 files in a zipped archive. Each file contains the dates and ratings given by distinct reviewers to one movie. The first line of the file indicates the movie number and the remaining data is in CSV format with three columns (reviewer,rating,date).
The files were unzipped and then concatenated, adding a column containing the file number and moving the rating to the final column. The format of the resulting CSV file was four columns (movie number, reviewer, date, rating).

A.6 Weather

The weather data set contains 124 million surface synoptic weather reports from land stations for the 10-year period from December 1981 through November 1991, except January 1982. Each report is 56 characters in length. Files with the format <MONTH><YEAR>L.DAT.Z were downloaded from http://cdiac.ornl.gov/ftp/ndp026b/ [Hahn 04]. In total 119 data files were used. The files were unzipped and amalgamated to a single file (Command A.7).

Command A.7 Unzip and concatenate the files

```
zcat *.Z > weather_large.DAT
```

```
82040100,1,8250,29767,71082,71,8,8,5,5,10,-1,800,900,0,0,8,9,4
82040100,1,8250,29767,71082,71,5,10,-1,8,9,4
```

Figure A.1: Sample input line from weather_large.csv and its corresponding output line.

A comma-separated file was created by executing a Java application, which retained the attributes listed in Table A.3 and excluded attributes 7, and 12–16. (The code is available at http://www.hazel-webb.com/archive.htm.) The resulting file was loaded into a MySQL database. Figure A.1 gives a sample input line and its corresponding output line.

Table A.3: Weather data: dimensions and cardinality of dimensions for cubes W1 and W2.

Dimension		Dimension cardinality
attr0	year,month,day,hour	28 767
attr1	sky brightness	2
attr2	latitude $\times$ 100	4 337
attr3	longitude $\times$ 100	6 344
attr4	station number	8696
attr5	present weather	101
attr6	low cloud type	13
attr8	high cloud type	14
attr9	change code	11
attr10	solar altitude	10
attr11	relative lunar illuminance	1 799
ignored	land/ocean indicator	226
Measure	total cloud cover	
Unused measures		
	lower cloud amount	
	lower cloud base height	
	middle cloud amount $\times$ 100	
	high cloud amount $\times$ 100	
	middle cloud amount(non overlapped)	
	high cloud amount (non overlapped)	

Appendix B

SQL Queries

Tables census_stocks, tweed and weather_large were created by loading the relevant CSV files into a MySQL database as described in Appendix A. Cubes C2, TW3 and W2 were extracted from the MySQL database using Listing B.1, Listing B.2 and Listing B.3, respectively. They were also exported in CSV so that they could be used by the other implementations.

Listing B.1: Extract C2 from the census income data

```
INSERT INTO temp(attr0,attr1,attr2, attr3, attr4,attr5,attr6,attr7,
    attr8, attr9,attr10,attr11,attr12, attr13,
    attr14,attr15,attr16,attr17, attr18, attr19,attr20,attr21,attr22,
    attr23, attr24,attr25,attr26, measure)
SELECT attr0,attr1,attr2, attr3, attr4,attr5,attr6,attr7, attr8,
    attr9,attr10,attr11,attr12, attr13, attr14,attr15,attr16,attr17,
    attr18, attr19,attr20,attr21,attr22, attr23, attr24,attr25,attr26,
    sum(measure)
FROM census_stocks
GROUP BY attr0,attr1,attr2, attr3, attr4,attr5,attr6,attr7, attr8,
    attr9,attr10,attr11,attr12, attr13, attr14,attr15,attr16,attr17,
    attr18, attr19,attr20,attr21,attr22, attr23, attr24,attr25,attr26;
```

Listing B.2: Extract TWEED cube

```
INSERT INTO tweed15( d1, d2, d3, d4, d5, d6, d7, d8, d31, d32, d33,
   d34, d50, d51, d52, d49 )
SELECT d1, d2, d3, d4, d5, d6, d7, d8, d31, d32, d33, d34, d50, d51,
   d52, sum( d9 )
FROM tweed
GROUP BY d1, d2, d3, d4, d5, d6, d7, d8,d31, d32, d33, d34, d50, d51,
   d52
```

Listing B.3: Extract weather cube

```
INSERT INTO weather12d(attr0, attr1, attr2, attr3, attr4, attr5, attr6,
    attr7, attr8, attr9, attr10, attr11, measure)
SELECT attr0, attr1, attr2, attr3, attr4, attr5, attr9, attr10, attr11,
    attr16, attr17, attr18, sum(attr6)
FROM weather_large
GROUP BY attr0, attr1, attr2, attr3, attr4, attr5, attr9, attr10,
   attr11, attr16, attr17, attr18;
```

B.1 Implementation of Algorithm SQLDD

Part of the Java code implementing Algorithm SQLDD is given in Listing B.4. It assumes that a MySQL database table with the column names “attr1, attr2, . . . attr$d$, measure, flag” is populated with the cube of interest.

Listing B.4: Implementation of Algorithm SQLDD

```
/*******************************************************************************
 * DBSQL4.java
 * implementation of SQL4 using jdbc

 * Sets a delete flag to true when cell is deleted from cube and rebuilds whenever
 * 75% of remaining cells are marked for deletion
 *
*******************************************************************************/
```

```
import java.sql.*;
import java.util.*;


public class DBSQL4 {

    private static Statement stmt;
    private static Statement countStmt;
    private static Statement deleteStmt;
    private  static String table;
    private static String table_base;
    private static int k; //carats
    private static int d; //number of dimensions
    private static int cells;
    private static int cellsDeleted;
    private static Connection con;
    private static final double FACTOR = .75;


    public static void main(String[] args) {

        // process command line arguments for table name, number of dims, number of
            carats - code omitted
        // connect to the database - code omitted
        // call process

    }

    /**
       process for a k-carat diamond.
       Sets delete flag to true as it processes.
    **/
    public void process() {

        boolean deletedSome = false;
        boolean moreToDelete = false;

        try {
            moreToDelete = true;
            //iterate until we have found the diamond
            while (moreToDelete) {
                moreToDelete = false;
                //loop through dims in single iteration
                for (int dim = 0; dim < d; dim++) {
                    if (countAndDelete("attr"+dim,dim,k))
                        moreToDelete = true;
                }
            }
        } catch (Exception e) {}
        //catch exceptions - code omitted
    }

    /**
       countAndDelete
       @param col dimension name
       @param dim dimension number
       @param k number of carats
       table table name is static
       queries table for the list of unique attributes and stores in table temp
       sets delete flag of attributes with count < k
    **/
    public static boolean countAndDelete(String col, int dim, int k) {
        String statement = "";
        ResultSet countsRS;
        boolean deletedSome = false;
        try {
            stmt  = con.createStatement();
            countStmt = con.createStatement();
            deleteStmt = con.createStatement();

            if (cells > 0 && cellsDeleted > 0 && cellsDeleted > cells*FACTOR) {
                if (table.charAt(table.length()-1) == '2')
                    table_base = table.substring(0,table.length()-1);
                else
                    table_base = table +"2";
                //rebuild the table and use table2 from now on to process
                deleteStmt.executeUpdate("DELETE FROM " + table_base);
                //clear out any old data then insert the next iteration
                deleteStmt.executeUpdate("INSERT into " + table_base +" SELECT * from
                     " + table +" WHERE flag = false");
                System.out.println("rebuilt table when " + cellsDeleted +" cells were
                     deleted");
                table = table_base;
                cells = cells - cellsDeleted;//rebuild the table as often as
                    necessary swapping between table and table2
                cellsDeleted = 0;//reset the counter
            }


            //ensure we have no leftover data in temp
            stmt.executeUpdate("DELETE FROM temp");

            //get attribute counts and store in temp
            int rows = countStmt.executeUpdate("INSERT INTO temp SELECT "+ col + " as
                 col, count(" + col + ") as cnt  FROM "+ table +" WHERE flag = false
                GROUP BY "+ col);
            if (rows > 0) {
                //delete attributes with count < k from table
                int del= deleteStmt.executeUpdate("UPDATE " +  table +",temp SET flag
                     = true WHERE temp.attr0 = "+table+"."+col+ " AND temp.attr1 < "
                    +k);
                if (del > 0) {
                    deletedSome= true;
                    cellsDeleted = cellsDeleted + del;
                    System.out.println("cellsDeleted and del " + cellsDeleted +" \t"
                        + del);
                }
            }
        } catch (Exception e) {
        }
        // catch Exceptions - code omitted
        return deletedSome;
    }

}
```

B.1.1 Algorithm SQLDD on Different Database Engines

We were interested in the effect of other database engines on our algorithms, so we implemented SQL4 on a different test machine. The SQL queries were run against MySQL version 5.0.41-community and SQL Server 2008 Enterprise edition version 10.0.1600.22

on a Dell 1750 with a 2.8 GHz Intel processor 80532K and 2 GiB RAM. It was running Windows Server 2003.

We used W3 and NF4, which comprised the first 500 000 cells of NF1. Results are given in Table B.1. Times in seconds are averaged over 10 runs. We see, for this particular test, that MySQL is almost twice as fast as SQL Server. Although we cannot make a general statement that all databases are too slow at computing diamonds, we do see that these two popular RDBMSs are both slower than the binary Java implementation.

cube	engine	Preprocessing	Processing
W3	SQL Server	189	192
	MySQL	27	103
	binary	51	25
NF4	SQL Server	31	57
	MySQL	7	28
	binary	12	3

Table B.1: Comparison of SQL Server and MySQL. Times (in seconds) are averaged over 10 runs.

Appendix C

Algorithm to Generate DCLD1

As discussed in Section 6.4 synthetic cubes were generated to test our DCLD heuristic. Algorithm C.1 was used to generate cube DCLD1. A similar approach with different weights was used to generate cube DCLD2. Figures C.1 and C.2 list the cells in cubes DCLD1 and DCLD2, respectively.

```
prob ← $0.1$
for i ∈ $1 \ldots 100$ do
    for j ∈ $1 \ldots d$ do
        // generate a uniform random number in the range
           0-119
        x ← ran(0 − 119)
        if x $< 120 *$ prob then
            // generate a uniform random number in the range
               0-2
            chosen$_j$ ← ran(0 − 2)
        else
            // 90% of the time we will generate a uniform
               random number in the range 3-11
            chosen$_j$ ← ran(3 − 11)
    // write chosen to cube DCLD1
```

Algorithm C.1: Generates DCLD1: values 0–2 have a 10% chance of being chosen in each dimension.

Figure C.1: Cells remaining in DCLD1 after the duplicates were removed.

```
0,0,10,   1,7,8,    1,9,9,    2,6,5,  2,8,5,    3,1,8,    3,2,8,    4,2,3,
4,3,8,    4,4,0,    4,7,3,    4,7,4,  4,7,9,    4,8,8,    4,9,9,    5,2,3,
5,4,8,    5,7,5,    5,8,7,    5,9,6,  6,2,6,    6,4,6,    6,5,5,    6,6,4,
6,7,5,    6,7,8,    6,8,3,    6,9,2,  7,5,3,    7,6,4,    7,6,7,    7,7,7,
7,8,8,    8,1,8,    8,2,3,    8,4,0,  8,4,6,    8,4,8,    8,5,3,    8,5,6,
8,5,9,    8,6,4,    8,6,5,    8,7,4,  8,8,3,    8,8,7,    8,8,9,    8,9,5,
9,1,2,    9,3,3,    9,6,9,    9,8,5,  10,6,7,   10,6,8,   10,6,9,   10,8,3,
1,11,3,   11,3,5,   11,4,3,   11,4,4, 11,4,9,   11,5,5,   2,11,8,   2,3,10,
3,11,0,   3,11,5,   3,11,9,   3,4,10, 3,5,11,   3,8,10,   4,10,5,   4,11,9,
4,3,10,   4,3,11,   4,6,10,   5,11,1, 5,11,3,   5,11,8,   6,11,9,   6,6,10,
6,6,11,   6,7,10,   7,11,6,   7,6,11, 7,8,10,   8,4,10,   9,10,4,   9,10,9,
9,2,11,   10,2,11,  11,9,11,  9,11,10,
```

Figure C.2: Cells remaining in DCLD2 after the duplicates were removed.

```
0,0,10,   0,0,2,   0,10,1,   0,1,10,   0,1,11,   0,1,3,    0,1,8,     0,2,3,
0,3,0,    0,5,6,   0,7,1,    0,9,0,    10,0,1,   1,0,1,    1,0,2,     1,0,6,
1,0,7,    1,10,5,  11,0,6,   1,1,1,    1,1,10,   11,1,0,   1,11,10,   11,11,4,
11,1,2,   1,11,6,  1,11,8,   11,3,2,   11,8,2,   11,9,0,   11,9,11,   1,2,1,
1,5,8,    1,6,1,   1,6,10,   1,7,1,    1,9,10,   1,9,9,    2,0,0,     2,0,5,
2,1,0,    2,10,0,  2,10,1,   2,10,2,   2,1,1,    2,1,11,   2,1,7,     2,2,1,
2,2,5,    2,2,8,   2,4,7,    2,5,4,    2,6,11,   2,6,7,    2,8,10,    2,9,2,
2,9,9,    3,0,2,   3,11,11,  3,11,3,   3,5,11,   3,9,8,    4,0,1,     4,1,1,
4,2,0,    4,5,1,   4,9,1,    5,0,3,    5,10,1,   5,1,2,    5,2,0,     5,4,8,
5,5,2,    5,6,11,  6,0,9,    6,10,0,   6,5,1,    7,0,6,    7,0,9,     7,1,1,
7,9,10,   8,0,0,   8,1,11,   8,2,2,    8,4,1,    8,4,4,    8,7,4,     8,9,8,
9,0,0,    9,1,1,   9,2,2,    9,2,7,    9,6,8,    9,7,8,
```

Appendix D

Java Code to Implement Algorithm SIDD

Part of the Java preprocessing and diamond building applications are given in Listing D.1 and Listing D.2. The preprocessor is common to both string and binary implementations. We list DBCount_binary.java and note that the string implementation only differs in the data type and the data structure used to keep track of counts.

Listing D.1: Preprocessor.

```
/***************************************************************************
 * DBCountPreprocessor.java
 * Preprocesses csv file  outputs are used by DBCount_binary or DBCount to
 * process for K.
 ***************************************************************************/
import java.util.*;
import java.io.*;

public class DBCountPreprocessor {

    private int d;
    //number of data dims
    private String file;
    //store input file name
    private Vector<Integer>[]data;
    //data structure ragged array of Vectors to store counts for nomalized attributes
    private Hashtable[] mappedCoords;
    //ds  maps attribute values to their counts
    private Hashtable[] mappedNorms;
    //ds helps to transform data to normalized integers
    private boolean norm = true;//default--preprocess for binary data
private boolean hash = true;//default--preprocess for string data
public static void main(String[] args) {

    // parse command line arguments for file name and number of dims -- code
        omitted

    // call createHashTables

    // call writeHashTables:  writes hashtables to file for future processing.
        Java does not guarantee order of elements returned from a HashTable, so
        it is necessary to run the (key,value) pairs through Enumerations in
        order to keep a persistent and accurate  record -- code omitted
    // call writeArrays: writes binary file of packed integers -- code omitted
    // call writeNormFile: writes normalized data to csv file and a file of
        packed integers for future binary processing -- code omitted
}


/**
   createHashTables
   @param fname String of the csv fact table to process.
   CSV file should have no headers or metadata.
   creates d hashtables of <key(attribute), value(count)> pairs
   creates d hash tables of <key(attribute>,value(normvalue)> pairs
   creates array of vectors that store counts for the normalized attributes

**/
public void createHashTables(String fname, int d) {
    BufferedReader raw;
    PrintWriter permanent_out;

    // initialize 'em happens regardless of output choice...
    for (int i = 0; i < d; i++) {
        mappedCoords[i]=new Hashtable<String,Integer>();
        mappedNorms[i] = new Hashtable<String,Integer>();
    }
    // map all of the attributes to their counts
    try {
        permanent_out = new PrintWriter(new BufferedWriter(new
            FileWriter(fname+"-string-to-norm")));
        raw = new BufferedReader(new InputStreamReader(new
            FileInputStream(fname)));
        String line = raw.readLine();
        Integer val = new Integer(1);
        Integer[] normVal = new Integer[d];
        //ds to store the normalized value for the next key in each dimension

        for (int i = 0; i < d; ++i)
            normVal[i] = new Integer(0);

        while (line!=null) {
            //ensure a comma at the end of the line so split will extract all the
                fields
            if (!(line.charAt(line.length()-1)==','))
                line = line + ",";
            String[] atts = line.split(",");
            for (int k = 0; k < d; ++k) {
                String key = atts[k];
                if (hash) {
                    if (!mappedCoords[k].containsKey(key)) {
                        //if it wasn't mapped yet

                        mappedCoords[k].put(key,val);
                        //add this key to mappedCoords hash
                    } else {
                        //increment count for this attribute
```

```
                            Integer intVal = (Integer)mappedCoords[k].get(key);
                            mappedCoords[k].put(key,intVal + val);
                            //increase count of this key in mappedCoords hash
                        }
                    }

                    if (norm) {
                        if (!mappedNorms[k].containsKey(key)) {
                            //if it wasn't mapped yet

                            mappedNorms[k].put(key,normVal[k]);
                            //add this key to mappedNorms hash and store the
                                relationshiop permanently
                            permanent_out.println("dim: "+ k + "\t"+ key
                                +"\t"+normVal[k].intValue());

                            data[k].add(val);
                            //add a place holder for counts in the appropriate vector

                            normVal[k] = (Integer)(normVal[k].intValue() +
                                val.intValue());
                            //increase the current norm value for the next key seen
                        } else {
                            //increment count for this attribute
                            Integer count =(Integer) mappedNorms[k].get(key);
                            int accumulator = data[k].get(count);
                            data[k].set(count, accumulator+1);
                            //increase count of this key in the appropriate vector
                                slot
                        }
                    }
                }

                line = raw.readLine();
            }
            raw.close();
            permanent_out.close();

        } catch (IOException ie) {
            ie.printStackTrace();
        }
    }

}
```

Listing D.2: Diamond builder for binary data.

```
/****************************************************************************
 * DBCount_binary.java
 * input:  1) preprocessed binary file (output of DBCountPreprocessor.java)
 *                 format of input file:
 *                 number of dims,  cardinalities for each dim, data - all integers
 *  2) k: number of carats
 ****************************************************************************/
import java.util.*;
import java.io.*;

public class DBCount_binary {

    private int d;     //number of data dims
    private int K;     //number of carats
    private int iters = 0;     //count number of times the file is processed until
        convergence
    private String file;     //to store input file name from command line
    int[][] data;     //ragged array of arrays storing counts of the attrvals
    private int FILE_NAME_LENGTH;

    public static void main(String[] args) {

        // parse command line arguments for file name and number of carats K -- code
            omitted
        // call loadArrays: read in counts for each attribute in each dimension --
            code omitted
        // call initArrays: sets count to zero for every attribute whose count is
            less than K -- code omitted

        // call process

        // call writeDiamond: writes cells remaining in diamond to csv file -- code
            omitted
        // call writeDiamondCounts: writes all attribute values and their current
            count to file -- code omitted
    }

    /* process
     * counts for dim i are stored in data(i)
     * a count of zero removes this attr from the diamond
     */
    public void process(String inFile, String outFile ) {

        File file = null;
        iters++;
        boolean reprocess = false;//assume this will be the last time through the
            input file

        try {
            FileInputStream file_input = new FileInputStream (new File(inFile));
            DataInputStream data_in    = new DataInputStream( new
                BufferedInputStream(file_input ));

            file = new File(outFile);
            DataOutputStream data_out = new DataOutputStream(new
                BufferedOutputStream(new FileOutputStream(file)));

            int[] attrs = new int[d];//somewhere to store each line of fact table
            boolean del = false;
            try {
                while (true) {
                    for (int i = 0; i < d; ++i) {
```

```
                    attrs[i] = data_in.readInt();
                    //read in a line of the fact table
                }

                del = false;
                //we haven't yet deleted anything
                for (int i =0; i < d; ++i) {
                    //get counts array for dim i
                    if (data[i][attrs[i]] == 0) {
                        del = true;
                        reprocess = true;//
                        //we deleted something so we will need to reprocess the
                            file
                        //this attr is deleted so update others in the row
                        for (int j = 0; j < d; ++j) {
                            if (j != i) {
                                if (data[j][attrs[j]] > 0) {
                                    data[j][attrs[j]]--;
                                    //decrement the count
                                    if (data[j][attrs[j]] < K)
                                        data[j][attrs[j]] = 0;
                                    //this should be deleted on next iter
                                }
                            }
                        }
                        if (del) break;
                        //only delete this row once
                    }
                }
                if (!del) {
                    for (int v: attrs) {
                        data_out.writeInt(v);
                        //write this row to new diamond file
                    }
                }
            }
        } catch (EOFException eof) {
            //do nothing this terminates the loop
        }

        data_in.close ();
        data_out.close();

        if (reprocess) {
            if (inFile.charAt(FILE_NAME_LENGTH)=='_')
                (new File(inFile)).delete();
            //and delete the old file except original data
            inFile = inFile.substring(0,FILE_NAME_LENGTH)+"_"+iters +"B";
            //force a new file to be output each time
            process(outFile,inFile);
            //call process again with newly minted data file
        }
    } catch  (IOException e) {
        System.out.println ( "IO Exception =: " + e );
    }
}

}
```

Appendix E

Words Remaining in $\kappa(\mathsf{K1})$ Diamond

The κ-diamond for $\mathsf{K1}$ was established as described in Section 5.5.1 and the words remaining in the κ-diamond are given in Tables E.1, E.2, E.3 and E.4. Words that occur in one dimension, but none of the others are shown in boldface.

Table E.1: Words remaining in dimension one of $\kappa(\mathsf{K1})$ diamond.

words remaining in dimension one			
adadah	adithaim	adullam	**amam**
ashnah	azekah	azem	baalah
bealoth	beersheba	bethpalet	bizjothjah
chesil	**children**	citi	**coast**
dimonah	**eder**	**edom**	eltolad
enam	engannim	eshtaol	gederah
hadattah	**hazargaddah**	hazarshu	**hazor**
heshmon	**hezron**	hormah	**ithnan**
jagur	jarmuth	**judah**	**kabzeel**
kedesh	**kerioth**	**kinah**	lebaoth
madmannah	**moladah**	nine	rimmon
sansannah	sharaim	**shema**	**shilhim**
socoh	**southward**	tappuah	**telem**
their	**toward**	**tribe**	twenti
uttermost	vallei	villag	**were**
which	with	zanoah	ziklag
ziph	zoreah		

Table E.2: Words remaining in dimension two of $\kappa(\mathsf{K1})$ diamond.

words remaining in dimension two			
achzib	adithaim	adullam	ashan
ashdod	ashnah	azekah	azem
baalah	beersheba	bethdagon	bethpalet
bizjothjah	bozkath	cabbon	chesil
citi	dilean	eglon	ekron
eltolad	enam	engannim	eshtaol
ether	even	**fourteen**	from
gederah	gederoth	**gederothaim**	hadashah
hazarshu	hormah	jarmuth	jiphtah
joktheel	keilah	kithlish	lachish
lahmam	lebaoth	libnah	madmannah
makkedah	mareshah	migdalgad	mizpeh
naamah	near	nezib	nine
rimmon	sansannah	sharaim	shilhim
sixteen	socoh	tappuah	that
their	town	twenti	unto
vallei	villag	which	with
zanoah	zenan	ziklag	zoreah

Table E.3: Words remaining in dimension three of $\kappa(\mathsf{K1})$ diamond.

words remaining in dimension three			
achzib	anab	anim	aphekah
arab	ashan	ashdod	ashnah
bethdagon	bethtappuah	border	bozkath
cabbon	citi	dannah	debir
dilean	dumah	eglon	egypt
ekron	eleven	eshean	eshtemoh
ether	even	from	gaza
gederoth	giloh	goshen	great
hadashah	hebron	holon	humtah
janum	jattir	jiphtah	joktheel
keilah	kirjatharba	kirjathsannah	kithlish
lachish	lahmam	libnah	makkedah
maon	mareshah	migdalgad	mizpeh
mountain	naamah	near	nezib
nine	river	shamir	sixteen
socoh	that	their	thereof
town	unto	villag	which
with	zanoah	zenan	zior

Table E.4: Words remaining in dimension four of $\kappa(\mathsf{K1})$ diamond.

words remaining in dimension four			
anab	anim	aphekah	arab
ashdod	**bethanoth**	**betharabah**	bethtappuah
bethzur	border	**cain**	**carmel**
citi	dannah	debir	dumah
egypt	eleven	**eltekon**	**engedi**
eshean	eshtemoh	gaza	**gedor**
gibeah	giloh	goshen	great
halhul	hebron	holon	humtah
janum	jattir	**jezreel**	**jokdeam**
juttah	kirjatharba	**kirjathba**	**kirjathjearim**
kirjathsannah	**maarath**	maon	**middin**
mountain	**nibshan**	nine	**rabbah**
river	**salt**	**secacah**	shamir
socoh	their	thereof	**timnah**
town	unto	villag	which
wilder	with	zanoah	zior
ziph			

Bibliography

[Agra 08] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. "Design Tradeoffs for SSD Performance". In: *USENIX Annual Technical Conference*, pp. 57–70, 2008.

[Agra 87] R. Agrawal. "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries". In: *ICDE'87*, pp. 580–590, 1987.

[Agra 94] R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules". In: J. B. Bocca, M. Jarke, and C. Zaniolo, Eds., *VLDB'94*, pp. 487–499, Morgan Kaufmann, 12–15 1994.

[Albe 06] A. Albert. "Diamond Dicing". Honours Thesis, University of New Brunswick Saint John, 2006.

[Ande 06] C. Anderson. *The Long Tail*. Hyperion, 2006.

[Aoui 07] K. Aouiche and D. Lemire. "A Comparison of Five Probabilistic View-size Estimation Techniques in OLAP". In: *DOLAP'07*, pp. 17–24, 2007.

[Aoui 09a] K. Aouiche, D. Lemire, and R. Godin. "Web 2.0 OLAP: From Data Cubes to Tag Clouds". *Lecture Notes in Business Information Processing*, Vol. 18, pp. 51–64, 2009.

[Aoui 09b] K. Aouiche, D. Lemire, and R. Godin. *Web Information Systems and Technologies*, Chap. Web 2.0 OLAP: From Data Cubes to Tag Clouds, pp. 51–64. Vol. 18 of *Lecture Notes in Business Information Processing*, Springer, 2009.

[Arme 85] A. C. Armenakis, L. E. Garey, and R. D. Gupta. "An Adaptation of a Root Finding Method to Searching Ordered Disk Files". *BIT*, Vol. 25, No. 4, pp. 562–568, 1985.

[Babc 03] B. Babcock, S. Chaudhuri, and G. Das. "Dynamic Sample Selection for Approximate Query Processing". In: *SIGMOD'03*, pp. 539–550, 2003.

[Barb 01] D. Barbará and X. Wu. "Finding Dense Clusters in Hyperspace: An Approach Based on Row Shuffling.". In: *Advances in Web-age Information Management*, pp. 305–316, 2001.

[Ben 06a] R. Ben Messaoud, O. Boussaid, and S. Loudcher Rabaséda. "Efficient Multi-dimensional Data Representations Based on Multiple Correspondence Analysis". In: *KDD'06*, pp. 662–667, 2006.

[Ben 06b] R. Ben Messaoud, S. Loudcher Rabaséda, O. Boussaid, and R. Missaoui. "Enhanced Mining of Association Rules from Data Cubes". In: *DOLAP '06*, pp. 11–18, 2006.

[Benn 07] J. Bennett and S. Lanning. "The Netflix Prize". In: *KDD Cup and Workshop 2007*, 2007.

[Borz 01] S. Börzsönyi, D. Kossmann, and K. Stocker. "The Skyline Operator". In: *ICDE '01*, pp. 421–430, IEEE Computer Society, 2001.

[Care 97] M. J. Carey and D. Kossmann. "On Saying "Enough already!" in SQL". In: *SIGMOD'97*, pp. 219–230, 1997.

[Cerf 09] L. Cerf, J. Besson, C. Robardet, and J.-F. Boulicaut. "Closed Patterns Meet N-ary Relations". *ACM Trans. Knowl. Discov. Data*, Vol. 3, No. 1, pp. 1–36, 2009.

[Chak 06] D. Chakrabarti and C. Faluoutsos. "Graph Mining: Laws, Generators, and Algorithms". *ACM Computing Surveys*, Vol. 38, pp. 1–69, 2006. Article 2.

[Cham 74] D. D. Chamberlin and R. F. Boyce. "SEQUEL: A Structured English Query Language". In: *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pp. 249–264, ACM, New York, NY, USA, 1974.

[Chen 05] Y. Chen, T. Eavis, F. Dehne, and A. Rau-Chaplin. "PnP: Parallel and External Memory Iceberg Cube Computation". In: *ICDE'05*, pp. 576–577, 2005.

[Codd 70] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". *Commun. ACM*, Vol. 13, No. 6, pp. 377–387, 1970.

[Corm 04] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. "Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-dimensional Data". In: *SIGMOD '04*, pp. 155–166, ACM Press, New York, NY, USA, 2004.

[Corm 05] G. Cormode and S. Muthukrishnan. "What's Hot and What's Not: Tracking Most Frequent Items Dynamically". *ACM Trans. Database Syst.*, Vol. 30, No. 1, pp. 249–278, 2005.

[Coze 04] O. Cozette, A. Guermouche, and G. Utard. "Adaptive Paging for a Multi-frontal Solver". In: *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pp. 267–276, ACM, New York, NY, USA, 2004.

[Dawa 01] M. Dawande, P. Keskinocak, J. M. Swaminathan, and S. Tayur. "On Bipartite and Multipartite Clique Problems". *Journal of Algorithms*, Vol. 41, No. 2, pp. 388–403, November 2001.

[Dell 09] Dell. *Specifications: Hitachi Deskstar P7K500 User's Guide*. 2009. https://support.dell.com/support/edocs/storage/P160227/specs.htm(Last checked 06–9-2010).

[Donj 99] D. Donjerkovic and R. Ramakrishnan. "Probabilistic Optimization of Top N Queries". In: *VLDB'99*, pp. 411–422, 1999.

[Dowe 72] M. Dowell and P. Jarratt. "The 'Pegasus' Method for Computing the Root of an Equation". *BIT Numerical Mathematics*, Vol. 12, No. 4, pp. 503–508, Dec. 1972.

[Ency 07] Encyclopædia Brittanica online. *diamond*. Encyclopædia Brittanica, Inc., 2007. Last checked 07-29-2009.

[Enge 07] J. O. Engene. "Five Decades of Terrorism in Europe: The TWEED Dataset". *Journal of Peace Research*, Vol. 44, No. 1, pp. 109–121, 2007.

[Fang 98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. "Computing Iceberg Queries Efficiently". In: *VLDB'98*, pp. 299–310, 1998.

[Feld 07] R. Feldman and J. Sanger. *The Text Mining Handbook*. Cambridge University Press, 2007.

[Feng 04] J. Feng, Q. Fang, and H. Ding. "PrefixCube: Prefix-sharing Condensed Data Cube". In: *DOLAP '04*, pp. 38–47, ACM Press, New York, NY, USA, 2004.

[Gall 89] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. "A Fast Parametric Maximum Flow Algorithm and Applications". *SIAM Journal on Computing*, Vol. 18, No. 1, pp. 30–55, 1989.

[Gant 00] V. Ganti, M. L. Lee, and R. Ramakrishnan. "ICICLES: Self-Tuning Samples for Approximate Query Answering". In: *VLDB'00*, pp. 176–187, 2000.

[Geor 07] G. F. Georgakopoulos and K. Politopoulos. "MAX-DENSITY Revisited: a Generalization and a More Efficient Algorithm". *The Computer Journal*, Vol. 50, No. 3, p. 348, 2007.

[Godi 95] R. Godin, R. Missaoui, and H. Alaoui. "Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices". *Computational Intelligence*, Vol. 11, pp. 246–267, 1995.

[Gray 96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total". In: *ICDE '96*, pp. 152–159, 1996.

[Hahn 04] C. Hahn, S. Warren, and J. London. “Edited Synoptic Cloud Reports from Ships and Land Stations over the Globe, 1982–1991”. http://cdiac.ornl.gov/ftp/ndp026b/ (Last checked 06-09-2010), Jan. 2004.

[Hale 01] A. Y. Halevy. “Answering queries using views: A survey”. *The VLDB Journal*, Vol. 10, No. 4, pp. 270–294, 2001.

[Hami 07] C. H. Hamilton and A. Rau-Chaplin. “Compact Hilbert Indices: Space-filling Curves for Domains with Unequal Side Lengths”. *Information Processing Letters*, Vol. 105, No. 5, pp. 155–163, 2007.

[Hann 05] R. Hanneman and M. Riddle. “Introduction to Social Networks”. http://www.faculty.ucr.edu/~hanneman/nettext/, 2005.

[Hett 00] S. Hettich and S. D. Bay. “The UCI KDD archive”. http://kdd.ics.uci.edu (Last checked 06-09-2010), 2000.

[Hipp 00] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. “Algorithms for Association Rule Mining — a General Survey and Comparison”. *SIGKDD Explor. Newsl.*, Vol. 2, No. 1, pp. 58–64, 2000.

[Hirs 05] J. E. Hirsch. “An Index to Quantify an Individual’s Scientific Research Output”. 2005. doi:10.1073/pnas.0507655102 (Last checked 07-29-2009).

[Hita 09] Hitachi Global Storage Technologies. *Deskstar P7K500*. 2009. http://www.hitachigst.com/tech/techlib.nsf/techdocs/30C3F554C477835B86257377006E61A0/$file/HGST_Deskstar_P7K500_DS_FINAL.pdf (Last checked 06-09-2010).

[Holl 07] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. “How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans”. In: *SIGMOD’07*, pp. 389–400, 2007.

[Holz 06] K. Holzapfel, S. Kosub, M. G. Maaß, and H. Täubig. “The Complexity of Detecting Fixed-Density Clusters”. 2006.

[IBM] IBM. “DB2 Intelligent Miner”. http://www-306.ibm.com/software/data/iminer/(Last checked 07-29-2009).

[Ilya 08] I. F. Ilyas, G. Beskales, and M. A. Soliman. “A Survey of Top-k Query Processing Techniques in Relational Database Systems”. *ACM Comput. Surv.*, Vol. 40, No. 4, pp. 1–58, 2008.

[ISO 08] ISO 9075-1:2008. *Information technology: database languages – SQL– Part 1 Framework*. ISO, Geneva, Switzerland, 3rd Ed., 2008.

[Kase 06] O. Kaser, S. Keith, and D. Lemire. “The LitOLAP Project: Data Warehousing with Literature”. In: *CaSTA’06*, 2006.

[Kase 07] O. Kaser and D. Lemire. “Tag-Cloud Drawing: Algorithms for Cloud Visualization”. In: *WWW 2007 – Tagging and Metadata for Social Information Organization*, 2007.

[Kase 08] O. Kaser, D. Lemire, and K. Aouiche. “Histogram-Aware Sorting for Enhanced Word-Aligned Compression in Bitmap Indexes”. In: *DOLAP ’08*, 2008.

[Klug 82] A. Klug. “Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions”. *Journal ACM*, Vol. 29, No. 3, pp. 699–717, 1982.

[Knut 97] D. E. Knuth. *Fundamental Algorithms*. Vol. 1 of *The Art of Computer Programming*, Addison-Wesley, 1997.

[Kouz 09] A. Kouznetsov, S. Matwin, D. Inkpen, A. H. Razavi, O. Frunza, M. Sehatkar, and L. Seaward. *Advances in Artificial Intelligence*, Chap. Classifying Biomedical Abstracts Using Committees of Classifiers and Collective Ranking Techniques, pp. 224–228. Vol. 5549/2009 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2009.

[Kuma 99] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. “Trawling the Web for Emerging Cyber-communities”. In: *WWW ’99*, pp. 1481–1493, Elsevier North-Holland, Inc., New York, NY, USA, 1999.

[Lee 06] S.-L. Lee. “An Effective Algorithm to Extract Dense Sub-cubes from a Large Sparse Cube.”. In: A. M. Tjoa and J. Trujillo, Eds., *DaWaK*, pp. 155–164, Springer, 2006.

[Lee 09] S.-W. Lee, B. Moon, and C. Park. “Advances in Flash Memory SSD Technology for Enterprise Database Applications”. In: *SIGMOD ’09*, pp. 863–870, ACM, New York, NY, USA, 2009.

[Lemi 08] D. Lemire and O. Kaser. “Hierarchical Bin Buffering: Online Local Moments for Dynamic External Memory Arrays”. *ACM Trans. Algorithms*, Vol. 4, No. 1, pp. 1–31, 2008.

[Lemi 09] D. Lemire and O. Kaser. “Reordering Columns for Smaller Indexes”. 2009. in preparation, available from http://arxiv.org/abs/0909.1346.

[Lemi 10] D. Lemire, O. Kaser, and K. Aouiche. “Sorting Improves Word-aligned Bitmap Indexes”. *Data & Knowledge Engineering*, Vol. 69, No. 1, pp. 3–28, 2010.

[Li 06] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang. “DADA: a Data Cube for Dominant Relationship Analysis”. In: *SIGMOD’06*, pp. 659–670, 2006.

[Lian 05] W. Lian, D. W. Cheung, and S. M. Yiu. “An Efficient Algorithm for Finding Dense Regions for Mining Quantitative Association Rules”. *Computers & Mathematics with Applications*, Vol. 50, No. 3-4, pp. 471–490, August 2005.

[Liu 02] R. X. Liu. *Simplifying Parallel Datacube Computation*. Master's thesis, University of New Brunswick, 2002.

[Loh 02a] Z. X. Loh, T. W. Ling, C. H. Ang, and S. Y. Lee. "Adaptive Method for Range Top-k Queries in OLAP Data Cubes". In: *DEXA'02*, pp. 648–657, 2002.

[Loh 02b] Z. X. Loh, T. W. Ling, C. H. Ang, and S. Y. Lee. "Analysis of Pre-computed Partition Top Method for Range Top-k Queries in OLAP Data Cubes". In: *CIKM'02*, pp. 60–67, 2002.

[Luo 94] Z.-Q. Luo and D. Parnas. "On the Computational Complexity of the Maximum Trade Problem". *Acta Mathematical Applicatae Sinica (English Series)*, Vol. 10, No. 4, pp. 434–440, 1994.

[Mani 05] A. Maniatis, P. Vassiliadis, S. Skiadopoulos, Y. Vassiliou, G. Mavrogonatos, and I. Michalarias. "A Presentation Model & Non-Traditional Visualization for OLAP". *International Journal of Data Warehousing and Mining*, Vol. 1, pp. 1–36, 2005.

[Miss 07] R. Missaoui, C. Goutte, A. K. Choupo, and A. Boujenoui. "A Probabilistic Model for Data Cube Compression and Query Approximation". In: *DOLAP*, pp. 33–40, 2007.

[Morf 07] K. Morfonios, S. Konakas, Y. Ioannidis, and N. Kotsis. "ROLAP Implementations of the Data Cube". *ACM Comput. Surv.*, Vol. 39, No. 4, p. 12, 2007.

[Mors 07] M. D. Morse, J. M. Patel, and H. V. Jagadish. "Efficient Skyline Computation over Low-Cardinality Domains". In: *VLDB'07*, pp. 267–278, 2007.

[Nade 03] T. P. E. Nadeau and T. J. E. Teorey. "A Pareto Model for OLAP View Size Estimation". *Information Systems Frontiers*, Vol. 5, No. 2, pp. 137–147, 2003.

[Netf 07] Netflix, Inc. "Nexflix Prize". http://www.netflixprize.com(Last checked 06-09-2010), 2007.

[Newm 05] M. Newman. "Power laws, Pareto Distributions and Zipf's Law". *Contemporary Physics*, Vol. 46, No. 5, pp. 323–351, 2005.

[Ng 97] W. Ng and C. Ravishankar. "Block-oriented Compression Techniques for Large Statistical Databases". *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 2, pp. 314–328, 1997.

[OLAP] OLAP Council, The. "OLAP and OLAP Server Definitions". http://www.olapcouncil.org/research/resrchly.htm (Last checked 07-29-2009).

[Peet 00] R. Peeters. "The Maximum-Edge Biclique Problem is NP-Complete". Research Memorandum 789, Faculty of Economics and Business Administration, Tilberg University, 2000.

[Pei 05] J. Pei, M. Cho, and D. Cheung. “Cross Table Cubing: Mining Iceberg Cubes from Data Warehouses”. In: *SDM’05*, pp. 461–465, 2005.

[Pens 05] R. G. Pensa and J. Boulicaut. “Fault Tolerant Formal Concept Analysis”. In: *AI*IA 2005*, pp. 212–233, Springer-Verlag, 2005.

[Poli 99] D. N. Politis, J. P. Romano, and M. Wolf. *Subsampling*. Springer, 1999.

[Port 97] M. F. Porter. “An Algorithm for Suffix Stripping”. In: *Readings in information retrieval*, pp. 313–316, Morgan Kaufmann, 1997.

[Proj 09] Project Gutenberg Literary Archive Foundation. “Project Gutenberg”. http://www.gutenberg.org/ (checked 06-09-2010), 2009.

[Ragh 03] S. Raghavan and H. Garcia-Molina. “Representing Web Graphs”. In: *ICDE’03*, pp. 1–10, ACM Press, 2003.

[Raoa 09] R. P. N. Raoa, N. Yadavb, M. N. Vahiab, H. Joglekard, and R. A. I. Mahadevanf. “A Markov Model of the Indus Script”. *Proceedings of the National Academy of Sciences of the United States of America*, 2009.

[Redd 01] P. K. Reddy and M. Kitsuregawa. “An Approach to Relate the Web Communities through Bipartite Graphs”. In: *WISE’01*, pp. 302–310, 2001.

[Rizz 06] S. Rizzi, A. Abelló, J. Lechtenbörger, and J. Trujillo. “Research in Data Warehouse Modeling and Design: Dead or Alive?”. In: *DOLAP ’06*, pp. 3–10, ACM, New York, NY, USA, 2006.

[Roos 09] P. Roosevelt. “Matryoshka Dolls”. 2009. http://www.encyclopedia.com/doc/1G2-3404100805.html (Last checked 07-29-2009).

[Rumm 70] R. Rummel. *Applied Factor Analysis*. Northwestern University Press, 1970.

[SAS] SAS. “Data Mining with SAS Enterprise Miner”. http://www.sas.com/technologies/analytics/datamining/miner/#section=3. (Last checked 07-29-2009).

[Stav 07] A. Stavrianou, P. Andritsos, and N. Nicoloyannis. “Overview and Semantic Issues of Text Mining”. *SIGMOD Record*, Vol. 36, No. 3, September 2007.

[Stra 82] A. C. Stratton. *Comparison of Searching Techniques*. Master’s thesis, University of New Brunswick, 1982.

[Suzu 05] A. Suzuki and T. Tokuyama. “Dense Subgraph Problem Revisited”. In: *Joint Workshop “New Horizons in Computing” and “Statistical Mechanical Approach to Probabilistic Information Processing”*, 2005.

[Tech 05] K. Techapichetvanich and A. Datta. “Interactive Visualization for OLAP”. In: *ICCSA ’05*, pp. 206–214, 2005.

[Terr 08] G. Terracina, N. Leone, V. Lio, and C. Panetta. “Experimenting with Recursive Queries in Database and Logic Programming Systems”. *Theory and Practice of Logic Programming*, Vol. 8, No. 2, pp. 129–165, 2008.

[Thom 02] E. Thomson. *OLAP Solutions: Building Multidimensional Information Systems*. Wiley, second Ed., 2002.

[Turn 05] P. D. Turney and M. L. Littman. “Corpus-Based Learning of Analogies and Semantic Relations”. *Machine Learning*, Vol. 60, No. 1–3, pp. 251–278, 2005.

[Vitt 98] J. S. Vitter, M. Wang, and B. Iyer. “Data Cube Approximation and Histograms via Wavelets”. In: *CIKM ’98*, pp. 96–104, New York, U.S.A., 1998.

[Wang 02] W. Wang, J. Feng, H. Lu, and J. X. Yu. “Condensed Cube: An Efficient Approach to Reducing Data Cube Size”. In: *ICDE ’02*, p. 155, IEEE Computer Society, Washington, DC, USA, 2002.

[Webb 07] H. Webb. “Properties and Applications of Diamond Cubes”. In: *ICSOFT 2007 – Doctoral Consortium*, 2007.

[Webb 09] H. Webb. “Code Archive”. http://www.hazel-webb.com/archive.htm, 2009. (Last checked 06-09-2010).

[Will 89] R. Wille. “Knowledge Acquisition by Methods of Formal Concept Analysis”. In: *Proceedings of the Conference on Data Analysis, Learning Symbolic and Numeric Knowledge*, pp. 365–380, Nova Science Publishers, Inc., Commack, NY, USA, 1989.

[Xin 03] D. Xin, J. Han, X. Li, and B. W. Wah. “Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration”. In: *VLDB’03*, pp. 476–487, 2003.

[Yang 05] K. Yang. “Information Retrieval on the Web”. *Annual Review of Information Science and Technology*, Vol. 39, pp. 33–81, 2005.

[Yiu 07] M. L. Yiu and N. Mamoulis. “Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data”. In: *VLDB’07*, pp. 483–494, 2007.

[Zhou 07] S. Zhou and G.-Q. Zhang. “Chinese Internet AS-level Topology”. *Communications, IET*, Vol. 1, No. 2, pp. 209 – 214, 2007.

CURRICULUM VITÆ

Hazel Jane Webb

Degrees:
Bachelor of Science in Data Analysis (Computer Science) First Division
University of New Brunswick Saint John 1998–2001

Master of Computer Science
University of New Brunswick 2001–2005
Thesis: An Efficient Data Structure for Almost Static Data Sets

Publications:
Hazel Webb " Properties and Applications of Diamond Cubes" *Proceedings of ICSOFT International Conference on Software and Data Technologies*, Barcelona Spain, July 2007.

Hazel Webb, Owen Kaser and Daniel Lemire "Pruning Attribute Values from Data Cubes with Diamond Dicing" *Proceedings of IDEAS08 Twelfth International Database Engineering and Applications Symposium*, Coimbra Portugal, September 2008.

Conference Presentations:
Hazel Webb and James Stewart "Parallelizing Existing Code for Shared and Distributed Memory Architectures Using OpenMP and MPI" *APICS Mathematics/Statistics and Computer Science Joint Conference*, St. Francis Xavier University, October 2001.

Hazel Webb " Interpolation B+ tree: an Efficient File Structure with which to Store and Search Large Almost Static Data Sets" *APICS Mathematics/Statistics and Computer Science Joint Conference*, UNB Saint John, October 2004.