Evaluating Multidimensional Queries by Diamond Dicing

Hazel Webb\textsuperscript{a}, Owen Kaser\textsuperscript{a}, Daniel Lemire\textsuperscript{b}

\textsuperscript{a}University of New Brunswick Saint John
\textsuperscript{b}Université du Québec à Montréal

Abstract

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. We have developed, implemented and tested algorithms to compute diamonds on both real and synthetic large data sets. We show that our custom implementation is more than twenty-five times faster, on a large data set, than popular database engines.

Keywords: OLAP, information retrieval, multidimensional queries

1. Introduction

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 \cite{1} 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 \cite{2,3}, including iceberg queries \cite{4,5,6} and top-k queries \cite{7,8}.

Such reduced representations are sometimes of critical importance to get good online performance in Business Intelligence (BI) applications \cite{4}. Even when performance is not an issue, browsing and visualising the data frequently benefit from reduced views \cite{10}.

Often, business analysts are interested in distinguishing elements that are most crucial to their business, such as the \( \text{k} \) products jointly responsible for 50\% of all sales, from the long tail \cite{11}—the lesser elements. The computation of icebergs, top-k elements, or heavy-hitters has received much attention \cite{12,13,14}. This type of query can be generalised so that interactions between dimensions are allowed. For example, 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...
Table 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.

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

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. We propose 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 [15], a graph-trawling approach used to analyse social networks.

To illustrate the proposal in the BI context, consider the following example. Table 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, 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.

For ease of visualisation, the example above is provided in two dimensions, but the diamond cube operator is defined over $d$ dimensions. For example, a three-dimensional query applied in the same BI context might constrain products, stores and manufacturers.

The proposed operation is a diamond dice [16, 17]. It produces a diamond, as formally defined in Section 2.

Other approaches that seek important attribute values, e.g. the Skyline operator [18], Dominant Relationship Analysis [20], and Top-$k$ dominating queries [21], require dimension attribute values to be ordered, e.g. distance between a hotel and a conference.

---

1This article is an expanded version of a conference paper [17].
venue, so that data points can be ordered. The proposed operator requires no such arrangement.

2. Properties of Diamond Cubes

In this section a formal model of the diamond cube is presented. We show that diamonds are nested, 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.

2.1. Formal Model

Researchers and developers have yet to agree on a single multidimensional model for OLAP. Our simplified formal model incorporates several widely-accepted definitions for the terms illustrated in Figure 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 attributes that defines one axis of a multidimensional data structure. Each dimension $D_i$ has a cardinality $n_i$, the number of distinct attribute 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 1a the dimensions are time, product and location. Values are typically organized in hierarchies, e.g., the cities might be aggregated per country.

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 $v = f(x_1, x_2, \ldots, x_d)$ is called a measure. 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 paper, 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

\[
f'(x_1, x_2, x_3, \ldots, x_d) = f(x_1, x_2, v_3, x_3, \ldots, x_d)
\]

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 \( \hat{D}_1, \hat{D}_2, \ldots, \hat{D}_d \) where each \( \hat{D}_i \subseteq D_i \). The cells in the slices corresponding to the unspecified attribute values are deallocated—deallocation of a cell sets its value to \( \bot \). 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, \( \sigma \), that assigns a real number to a slice of a cube. We choose aggregators such that they are commutative. For example, SUM is an aggregator: \( \text{SUM(slice}_i) = v_1 + v_2 + \cdots + v_m \) where \( m \) is the number of allocated cells in slice \( i \) and the \( v_i \)'s are the measures.

A slice \( S' \) is a subset of slice \( S \) if every allocated cell in \( S' \) is also an allocated cell in \( S \). An aggregator \( \sigma \) is monotonically non-decreasing if \( S' \subseteq S \) implies \( \sigma(S') \leq \sigma(S) \). Similarly, \( \sigma \) is monotonically non-increasing if \( S' \subseteq S \) 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 \( \sigma \) is distributive 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, the SUM aggregator is distributive: \( \text{SUM}(a_0, \ldots, a_k, a_{k+1}, \ldots, a_{n-1}) = \text{SUM}(a_0, \ldots, a_k) + \text{SUM}(a_{k+1}, \ldots, a_{n-1}) \).

An aggregator \( \sigma \) is algebraic if there is an intermediate tuple-valued distributive function \( G \) from which \( \sigma \) can be computed. An algebraic example is AVERAGE: given the tuple (COUNT, SUM), one can compute AVERAGE by a ratio. In other words, if \( \sigma \) 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,

\[
\text{COUNT, SUM}(a_0, a_1) = F((1, a_0)(1, a_1)) = (1 + 1, a_0 + a_1)
\]
<table>
<thead>
<tr>
<th>movie</th>
<th>reviewer</th>
<th>date</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1488844</td>
<td>2005-09-06</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>822109</td>
<td>2005-05-13</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>885013</td>
<td>2005-10-19</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>30878</td>
<td>2005-12-26</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>823519</td>
<td>2004-05-03</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>893988</td>
<td>2005-11-17</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>124105</td>
<td>2004-08-05</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1248029</td>
<td>2004-04-22</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 2: Part of the Netflix [25] fact table (cube). Attributes (dimensions) are movie, reviewer and date. Each row is a fact (allocated cell). The measure is rating.

and generally,

\[
\text{COUNT, SUM}(a_0, \ldots, a_k, a_{k+1}, \ldots, a_{n-1}) = F((k + 1, a_0 + a_1 + \ldots + a_k),
(n - 1 - k, a_{k+1} + a_{k+2} + \ldots + a_{n-1}))
= (n, a_0 + a_1 + \ldots + a_{n-1})
\]

An algebraic aggregator \( \sigma \) is linear [24] if the corresponding intermediate query \( G \) 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 \( \alpha \). \text{SUM} and \text{COUNT} and \text{AVERAGE} are linear functions; \text{MAX}, \text{MIN}, \text{ROOT-MEAN-SQUARE} and \text{MEDIAN} are not linear. Linearity and monotonicity are independent properties: \text{MAX} is monotone but not linear, whereas \text{AVERAGE} is linear but not monotone.

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

### 2.2. Diamond Cubes are Unique

Diamonds are defined over data cubes where the measures all have the same sign. 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.
**Definition 2.1.** Let $\sigma$ 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 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.

Cubes $A$ and $B$ are compatible if $f_A(x) \neq \perp$ and $f_B(x) \neq \perp$ $\implies f_A(x) = f_B(x)$.

**Definition 2.2.** 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

$$f_{A \cup B}(x) = \begin{cases} \perp & \text{if } f_A(x) \text{ and } f_B(x) \text{ are } \perp, \\ f_A(x) & \text{if only } f_B(x) \text{ is } \perp, \\ f_B(x) & \text{if only } f_A(x) \text{ is } \perp, \\ f_A(x) & \text{if } f_A(x) \neq \perp \text{ and } f_B(x) \neq \perp. \end{cases}$$

The union of two cubes is illustrated in Figure 3.

**Proposition 2.1.** If the aggregator $\sigma$ 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, for $i = \{1, 2, \ldots, d\}$. Similarly, if the aggregator $\sigma$ 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 from the monotonicity of the aggregator. $\square$

**Definition 2.3.** 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

$$f_{A \cap B}(x) = \begin{cases} \bot & \text{if } f_A(x) \text{ and } f_B(x) \text{ are } \bot, \\ \bot & \text{if only } f_B(x) \text{ is } \bot, \\ \bot & \text{if only } f_A(x) \text{ is } \bot, \\ f_A(x) & \text{if } f_A(x) \neq \bot \text{ and } f_B(x) \neq \bot \end{cases}$$

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 $\sigma$ 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 the 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. 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.

Diamonds are defined over data cubes where measures all have the same sign. If one allows negative and positive measures in the same cube, there may no longer be a unique solution to the problem ‘find the $k_1, k_2 \ldots k_d$-carat cube’. Moreover, the problem that we call the $k$-SUM Cube Problem ($k$-SCP), is NP-Complete. Formally, 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 $s_1, s_2, \ldots, s_d$. An NP-completeness proof follows trivially by reduction from the Maximum Trade Problem (MTP) [26].

2.3. Diamond Cubes are Nested

The following propositions show that diamonds are nested. 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 $i$.

**Proposition 2.2.** 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 2.1, $A \cup B$ has at least $k$ carats, and because $A$ is maximal, $A \cup B = A$; thus, $B$ is contained in $A$. \qed

---

\[When \ \sigma \ \text{is not monotonically non-decreasing (resp. non-increasing), there may not be a unique diamond.} \]
2.4. 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 \( \sigma \) is COUNT are given in the following propositions. Without loss of generality we assume that \( n_1 \leq n_2 \leq \ldots \leq n_{d-1} \leq n_d \).

**Proposition 2.3.** Given the sizes of dimensions of a COUNT-based diamond cube, \( n_i \), an upper bound for the number of carats \( k \) of a non-empty subcube is \( \prod_{i=1}^{d-1} n_i \). An upper bound on the number of carats \( k_i \) for dimension \( i \) is \( \prod_{j=1, j \neq i} n_i \).

**Proof.** Consider a perfect diamond, i.e. all cells are allocated. We can extract a slice from this diamond by holding a value in some dimension, \( D_x \), constant. The number of allocated cells in this slice will be the product of cardinalities of the remaining dimensions. If we choose \( D_x \) to be the dimension of largest cardinality, then the number of allocated cells in this slice is the carat \( (k) \) count for this diamond. Since all cells are allocated, there is no non-empty diamond with more than \( k \) carats. The rest of the result follows similarly.

An alternate upper bound on the number of carats in any dimension is \(|C|\), the number of allocated cells in the cube. For sparse cubes, this bound may be more useful than that from Proposition 2.3.

Intuitively, a cube with many 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 2.4.** For \( d > 1 \) and \( \sigma = \text{COUNT} \), the size \(|C|\) of a \( d \)-dimensional non-empty cube \( C \) of \( k \) carats satisfies \(|C| \geq kn_d \geq k^{d/(d-1)} \); more generally, a non-empty \( k_1, k_2, \ldots, k_d \)-carat cube has size satisfying

\[
|C| \geq \max k_i n_i \geq (\prod_{i=1}^{d} k_i)^{1/(d-1)}, i \in \{1, 2, \ldots, d\}.
\]

**Proof.** Pick dimension \( D_d \); the cube has \( n_d \) slices along this dimension, each with \( k \) allocated cells, proving that \(|C| \geq kn_d \).

We have that \( kn_d \geq k(\sum n_i)/d \)—since the average number of allocated cells per slice must be less than or equal to \( \max_{i=1,2,\ldots,d} n_i \)—so that \(|C|\) is at least \( k(\sum n_i)/d \). If we prove that \( \sum n_i \geq dk^{1/(d-1)} \) then we will have that \( k(\sum n_i)/d \geq k^{d/(d-1)} \).

This result can be shown using Lagrange multipliers. Consider the problem of minimising \( \sum n_i \) given the constraints \( \prod_{i=1,2,\ldots,j-1,j+1,\ldots,d} n_i \geq k \) for \( j = 1, 2, \ldots, d \). These constraints are necessary since all slices must contain at least \( k \) cells. The corresponding Lagrangian is

\[
L = \sum n_i + \sum \lambda_j (\prod_{i=1,2,\ldots,j-1,j+1,\ldots,d} n_i - k).
\]

By inspection, the derivatives of \( L \) with respect to \( n_1, n_2, \ldots, n_d \) are zero and all constraints are satisfied when \( n_1 = n_2 = \ldots = n_d = k^{1/(d-1)} \). For these values, \( \sum n_i = dk^{1/(d-1)} \) and this must be a minimum, proving the result. The more general result follows similarly, by proving that the minimum of \( \sum n_i k_i \) is reached when \( n_i = (\prod_{j=1,\ldots,d} k_i)^{1/(d-1)} / k_i \) for all \( i \)’s.

\[\Box\]
We calculate the volume of a cube $C$ as $\prod_{i=1}^{d} n_i$ and its density is the ratio of allocated cells, $|C|$, to the volume ($|C|/\prod_{i=1}^{d} n_i$). 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 2.1.** For count-based carats, if a cube does not contain a non-empty $k$-carat subcube, then it has at most $1 + (k - 1) \sum_{i=1}^{d} (n_i - 1)$ allocated cells. Hence, it has density at most $(1 + (k - 1) \sum_{i=1}^{d} (n_i - 1))/\prod_{i=1}^{d} n_i$. More generally, a cube that does not contain a non-empty $k_1, k_2, \ldots, k_d$-carat subcube has size at most $1 + \sum_{i=1}^{d} (k_i - 1)(n_i - 1)$ and density at most $(1 + \sum_{i=1}^{d} (k_i - 1)(n_i - 1))/\prod_{i=1}^{d} n_i$.

**Proof.** Suppose that a cube of dimension at most $n_1 \times n_2 \times \ldots \times n_d$ contains no $k$-carat diamond. Then one 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 one slice containing at most $k - 1$ allocated cells. Remove it. This iterative process can continue at most $\sum_{i} (n_i - 1)$ times before 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. □

Corollary 2.1 follows from Theorem 2.1.

**Corollary 2.1.** A cube of size greater than $1 + (k - 1) \sum_{i=1}^{d} (n_i - 1)$ allocated cells, that is, having density greater than

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

must contain a non-empty $k$-carat subcube. If a cube contains more than $1 + \sum_{i=1}^{d} (k_i - 1)(n_i - 1)$ allocated cells, it must contain a non-empty $k_1, k_2, \ldots, k_d$-carat subcube.

**Proof.** From Theorem 2.1 we have that a cube cannot have more than $1 + \sum_{i=1}^{d} (k_i - 1)(n_i - 1)$ allocated cells unless it also contains a non-empty $k$-diamond. Therefore, if a cube has more than $1 + \sum_{i=1}^{d} (k_i - 1)(n_i - 1)$ allocated cells, it must contain a nonempty $k$-diamond. □

2.4.1. Maximum number of carats

Given a cube $C$ and $\sigma$, $\kappa$ 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 2.5.** For count-based carats, we have $\kappa(C) \geq |C|/\sum_{i} (n_i - 1) - 3$.

**Proof.** Solving for $k$ in Corollary 2.1 we have a lower bound on the maximal number of carats: $\kappa(C) \geq |C|/\sum_{i} (n_i - 1) - 3$. □
2.5. 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 2.6 and 2.7 give tight lower and upper bounds respectively for $\kappa$.

**Proposition 2.6.** Given a non-empty cube $C$ and the aggregator SUM, a tight lower bound on $\kappa$ is the value of the maximum cell. 

**Proof.** The $\kappa$-diamond, by definition, is non-empty, so it follows that when the $\kappa$-diamond comprises a single cell, then $\kappa$ takes the value of the maximum cell in $C$. When the $\kappa$-diamond contains more than a single cell, $m$ is still a lower bound: either $\kappa$ 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 2.7.** A tight upper bound for $\kappa$ is

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

**Proof.** Let $X = \{\text{slice}_j(D_i) \mid \text{SUM}(\text{slice}_j(D_i)) = \max_{k} (\text{SUM}(\text{slice}_k(D_i)))\}$ then there is one slice $x$ whose SUM is smaller than or equal to all other slices in $X$. Suppose $\kappa(C)$ is greater than SUM($x$) then it follows that all slices in this $\kappa$-diamond must have SUM greater than 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, $\min_i (\max_j (\text{SUM}(\text{slice}_j(D_i))))$ is an upper bound for $\kappa$.

To show that $\min_i (\max_j (\text{SUM}(\text{slice}_j(D_i))))$ is also a tight upper bound we only need to consider a perfect cube where all measures are identical.

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

**Lemma 2.1.** If the diamond of size $s_1 \times s_2 \times \cdots \times s_d$ has $k$-carats, then its sum is at least $k \max(s_i)$ for $i \in \{1, 2, \ldots, d\}$.

**Proof.** We know that the sum is at least as large as the sum of $\max(s_i)$, and each $s_i$ is at least as large as $k$, from which the result follows.

3. 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. Several different approaches
were taken to develop algorithms for computing diamonds. Although there is no publicly available implementation of Kumar et al.'s algorithm [15], we have developed one that resembles the description provided in their paper. We refer to it as Kumar for the remainder of this paper. It has been extended to address more than two dimensions. We also implemented an algorithm using SQL. An intuitive solution, that loops through the cube checking and updating the COUNT or SUM for all attribute values in each dimension until it stabilizes, provides the basis for our most efficient algorithm.

The basic algorithm for computing diamonds is given in Algorithm 1. Its 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 ($\sigma$ is SUM), or insufficient allocated cells ($\sigma$ is COUNT). When the algorithm terminates, only attribute values that meet the condition in every slice remain: a diamond.

```
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 ∈ {1, ..., d} do
        for i in all attribute values of dimension dim do
            $C_{\text{dim}, i} ← \sigma$(slice i on dimension dim)
            if $C_{\text{dim}, i} < k_{\text{dim}}$ then
                delete attribute value i
                stable ← false
            end
        end
    end
return cube without deleted attribute values;
```

**Algorithm 1:** Algorithm to compute the diamond of any given cube by deleting slices eagerly.

Algorithms based on this approach always terminate, though they might sometimes return 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 3.1.** Algorithm 1 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 1 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 1 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 \( k_{\dim} \). 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 \( k_{\dim} \) carats. Therefore, we have a contradiction and must conclude that \( A' \) does not exist and that \( A \) is maximal.

### 3.1. 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. We are interested in processing large data. Therefore, we seek an efficient external memory implementation where the data cube can be stored in an external file whilst the important counts (resp. sums) are maintained in memory.

Algorithm 1 checks the \( \sigma \)-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. Even with the \texttt{COUNTS} maintained in main memory, it is prudent 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 [27]—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.

Algorithm 2 employs \( d \) hash tables that map attributes to their \( \sigma \)-values. A preprocessing step iterates once over the input file creating the \( d \) hash tables.

### 3.2. An SQL-Based Algorithm

A data cube can be represented in a relational database by a fact table. (See Figure 4a.) In this example the dimensions are: productID \( \{1,2,3,4\} \), salesPersonID \( \{a,b,c,d,e\} \) and month \( \{\text{Jan,Feb,Mar,Apr}\} \). 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 \( \sigma(\text{slicedi},i) < k \), but its removal exposes another attribute, and so on. See Figure 4b and consider computing the 2-carat diamond from the cube in Figure 4a. 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 1. 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 4b we see an example where the number of
**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

```plaintext
do for each dimension $i$
    Create hash table $ht_i$
    do for each attribute value $v$ in dimension $i$
        if $\sigma(\text{slice for value } v \text{ of dimension } i \text{ in } C) \geq k_i$ then
            $ht_i(v) = \sigma(\text{slice for value } v \text{ of dimension } i \text{ in } C)$
    end
end
```

**input**: file `inFile` containing $d$-dimensional cube $C$, integers $k_1, k_2 \ldots k_d > 0$

**output**: the diamond data cube

```plaintext
stable ← false
while ¬stable do
    Create new output file `outFile` // iterate main loop
    stable ← true
    do for each row $r$ of `inFile`
        $(v_1, v_2, \ldots, v_d) \leftarrow r$
        notDeleted ← true
        do for each $i \in \{1, \ldots, d\}$
            if $v_i \notin \text{dom } ht_i$ then
                do for each $j \in \{1, \ldots, i-1, i+1, \ldots, d\}$
                    if $v_j \in \text{dom } ht_j$ then
                        $ht_j(v_j) = ht_j(v_j) - \sigma(\{r\})$
                        if $ht_j(v_j) < k_i$ then
                            remove $v_j$ from dom $ht_j$
                        end
                    end
                end
                stable ← false
                notDeleted ← false
                break
            end
        end
        if notDeleted then
            write $r$ to `outFile`
        end
    end
    if ¬stable then
        `inFile` ← `outFile` // prepare for another iteration
    end
end
```

**Algorithm 2**: Diamond dicing for relationally stored cubes. Each iteration, less data is processed.
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.

We express the essential calculation in core SQL, as Algorithm 3, but this calculation is repeated by an “outermost loop” outside of SQL. Algorithm 3 can be executed against a copy of the fact table, which becomes smaller as the algorithm progresses. The fastest variation of this algorithm did not delete slices immediately, but instead updated boolean values to indicate the slices were not included in the solution. The data cube was rebuilt when 75% of the remaining cells were marked for deletion.

3.3. Complexity Analysis

Algorithm 1 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 4b) 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. 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. Unfortunately, there are some cubes for which this is not the case. Figure 5 illustrates such a cube. On the first iteration, processing columns first for the 2-carat diamond, a single cell is deleted. On 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 $\kappa$ does, however, influence $I$. Typically, when $k$ is far from $\kappa$—either less or greater—fewer iterations are required to converge. However,

---

3SQL-2003 supports looping and recursion [28]; 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 $\text{temp}_1$ AS
\begin{verbatim}
  (SELECT $\text{dim}_1$ FROM $\mathcal{R}$
   GROUP BY $\text{dim}_1$ HAVING $\sigma$(measure) < $k$);
\end{verbatim}

... 

CREATE TABLE $\text{temp}_d$ AS
\begin{verbatim}
  (SELECT $\text{dim}_d$ FROM $\mathcal{R}$
   GROUP BY $\text{dim}_d$ HAVING $\sigma$(measure) < $k$);
\end{verbatim}

DELETE FROM $\mathcal{R}$
WHERE $\text{dim}_1$ IN (SELECT * FROM $\text{temp}_1$) OR ...
$\text{dim}_d$ IN (SELECT * FROM $\text{temp}_d$);

Algorithm 3: Variation where the inner two loops in Algorithm 1 are (almost) computed in SQL. This process can be repeated until $\mathcal{R}$ stabilises.

when $k$ exceeds $\kappa$ by a very small amount, say 1, then typically a great many more iterations are required to converge to the empty cube.

Algorithm 2 runs in time $O(Id(|C|))$; each attribute value is deleted at most once. Often, the input file decreases substantially in the first few iterations and those cubes are processed faster than this bound suggests. The more carats we seek, the faster the file decreases initially.

4. Experiments

Experiments were conducted on both real and synthetic data sets to show that 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. Some of the properties of diamonds, including their density (count-based diamonds) and the range of values the carats may take, were reviewed experimentally.

4.1. 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 [29][30]), 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.
Figure 5: The 2-carat diamond requires more iterations to converge than 3-carat diamond.

Table 2: Statistics of TWEED, Netflix, census and weather data.

| cube | dimensions | $|C|$ | $\sum_{i=1}^d n_i$ | measure       |
|------|------------|-----|-------------------|---------------|
| TW1  | 4          | 1957| 88                | count         |
| TW2  | 15         | 4963| 674               | count         |
| TW3  | 15         | 4963| 674               | sum killed    |
| NF1  | 3          | 100478158| 500137 | count         |
| NF2  | 3          | 100478158| 500137 | sum rating    |
| NF3  | 4          | 200000000| 473753 | count         |
| C1   | 27         | 135753| 5607   | count         |
| C2   | 27         | 135753| 504    | sum stocks    |
| W1   | 12         | 123245878| 50320  | count         |
| W2   | 12         | 123245878| 50320  | sum total cloud cover |

The algorithms were implemented in Java, using SDK version 1.5.0, and the code is archived at a public website [31]. The SQL experiments were conducted on MySQL version 5.0 with mysql-connector-java-5.1.7.

4.2. 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.

4.2.1. Real Data

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

1. TWEED: [http://folk.uib.no/sspje/tweed.htm](http://folk.uib.no/sspje/tweed.htm)
Table 3: Statistics of the King James Bible data.

| cube  | dimensions | $|C|$  | $\sum_{i=1}^{d} n_i$ | measure                        |
|-------|------------|------|----------------------|--------------------------------|
| B1    | 4          | 54,601,077 | 31,634               | count                          |
| B2    | 4          | 24,000,000 | 27,042               | count                          |
| B3    | 4          | 32,000,000 | 29,078               | count                          |
| B4    | 4          | 40,000,000 | 30,417               | count                          |
| B5    | 4          | 54,601,077 | 31,634               | sum 4-gram instances          |
| B6    | 10         | 365,231,367| 6,335                | count                          |

Details of how the cubes were extracted are available at a public website [31] and their statistics are given in Table 2. A brief description of how data cubes were extracted from the King James Bible data follows.

The fifth data set was generated from the King James version of the Bible available at Project Gutenberg [36]. KJV-4grams [37] 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 [36], after stemming with the Porter algorithm [38] and removal of stemmed words with three or fewer letters. Occurrence of row $w_1, w_2, w_3, w_4$ indicates 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 [39, 40].

B1 was extracted from KJV-4grams. Duplicate records were removed and a count of each unique sequence was kept, which became the measure for cube B5. Four subcubes of B1 were also processed: B2: first 24,000,000 rows; B3: first 32,000,000 rows; and B4: first 40,000,000 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 500 million records had been generated—at the end of Genesis 19:30. Cube B6 was extracted from KJV-10grams. The statistics for all six cubes are given in Table 3.

4.2.2. Synthetic Data

To investigate the effect that data distribution might have on the size and shape of diamonds, nine cubes of varying dimensionality and distribution were constructed. We chose 1,000,000 cells with replacement from each of three different distributions:

- uniform—cubes $U1$, $U2$, and $U3$
- power law with exponent 3.5 to model the 65-35 skewed distribution—cubes $S1$, $S2$, $S3$
- power law with exponent 2.0 to model the 80-20 skewed distribution—cubes $SS1$, $SS2$, $SS3$

Details of the cubes generated are given in Table 4.
4.3. Preprocessing Step

To determine if row ordering would have an effect on our binary implementation of Algorithm 2, we chose two cubes—C1 and B2—and shuffled the rows using the Unix utility `shuf`. We compared preprocessing and processing times for each of six cubes, averaged over ten runs. Extracting cubes from the data sets included a sorting step so that duplicates could be easily removed. We found that preprocessing the sorted cube was up to 25% faster than preprocessing the shuffled cube. However, execution times for Algorithm 2 were within 3% for each cube. Therefore, we did not reorder the rows prior to processing.

The algorithms used in our experiments required different preprocessing of the cubes and the general preprocessing method is illustrated in Figure 6. For Algorithm 2, an in-memory data structure was used to maintain counts of the attribute values. The Kumar algorithm used $d$ sorted files and Algorithm 3 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.

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
Table 5: Wall-clock times (in seconds) for preprocessing real-world data sets. DB-Count and DBSum implement Algorithm 2 for both string and binary data. A ‘—’ indicates that this algorithm was not applied to the corresponding data cube.

<table>
<thead>
<tr>
<th>Cube</th>
<th>DBCount/DBSum (Alg. 2)</th>
<th>Kumar (Alg. 3)</th>
<th>SQL (Alg. 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>5.0 × 10^2</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>B2</td>
<td>2.0 × 10^2</td>
<td>—</td>
<td>1.1 × 10^3</td>
</tr>
<tr>
<td>B3</td>
<td>2.8 × 10^2</td>
<td>—</td>
<td>1.4 × 10^3</td>
</tr>
<tr>
<td>B4</td>
<td>3.5 × 10^2</td>
<td>—</td>
<td>1.8 × 10^3</td>
</tr>
<tr>
<td>C1</td>
<td>8.4 × 10^9</td>
<td>5.0 × 10^1</td>
<td>5.4 × 10^3</td>
</tr>
<tr>
<td>NF1</td>
<td>1.5 × 10^3</td>
<td>2.5 × 10^3</td>
<td>3.6 × 10^3</td>
</tr>
<tr>
<td>NF3</td>
<td>2.4 × 10^2</td>
<td>—</td>
<td>8.8 × 10^2</td>
</tr>
<tr>
<td>TW1</td>
<td>1.9 × 10^{-1}</td>
<td>2.0 × 10^1</td>
<td>4.0 × 10^{-1}</td>
</tr>
</tbody>
</table>

additional preparation costs. Table 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. The Kumar algorithm did not work well with high dimensional cubes. It requires a sorted copy of the cube for each dimension which incurs additional space and I/O costs. We only ran it against a few cubes.

4.4. Finding κ(C) for COUNT-based Diamonds

To test the theory of Proposition 2.5, the κ-diamond was built for each of the data sets. The initial guess (κ) for κ was the value calculated using Proposition 2.5. When a non-empty diamond was built, and for every data set this was indeed the case, κ was repeatedly doubled until an empty cube was returned and a tighter range for κ had been established. Next a simple binary search, which used the newly discovered lower and upper bounds as the end points of the search space, was executed. 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.

Statistics are provided in Table 6a. The estimate of κ comes from Proposition 2.5 and the number of 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.

4.5. Finding κ(C) for SUM-based Diamonds

From Proposition 2.7 we have that \( \min_i \max_j (\sigma_j(D_i)) \) is an upper bound for κ(C) of any sum-diamond and from Proposition 2.6 a lower bound is the maximum value stored in any cell. Indeed, for cube TW3 the lower bound is the κ value. For this reason, the approach to finding κ for the sum-diamonds varies slightly in that the first
Table 6: Iterations to convergence for sum and count diamonds

(a) The number of iterations it took to determine $\kappa$ for COUNT-based diamonds.

<table>
<thead>
<tr>
<th>cube</th>
<th>iters</th>
<th>value of $\kappa$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TW1</td>
<td>6</td>
<td>19</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>NF1</td>
<td>19</td>
<td>197</td>
<td>1 004</td>
<td></td>
</tr>
<tr>
<td>NF3</td>
<td>17</td>
<td>39</td>
<td>272</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>8</td>
<td>282</td>
<td>672</td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>13</td>
<td>2 447</td>
<td>4 492</td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>12</td>
<td>1 723</td>
<td>14 383</td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>16</td>
<td>884</td>
<td>7 094</td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>19</td>
<td>1 098</td>
<td>8 676</td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>12</td>
<td>1 347</td>
<td>10 513</td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td>5</td>
<td>57 668</td>
<td>112 232 566</td>
<td></td>
</tr>
</tbody>
</table>

(b) The number of iterations it took to determine $\kappa$ on SUM-based diamonds. The estimate for $\kappa$ is the tight lower bound from Proposition 2.6.

<table>
<thead>
<tr>
<th>cube</th>
<th>iters</th>
<th>value of $\kappa$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TW3</td>
<td>3</td>
<td>85</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>NF2</td>
<td>40</td>
<td>5</td>
<td>3 483</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>5</td>
<td>1 853</td>
<td>3 600 675</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>19</td>
<td>32</td>
<td>20 103</td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>4</td>
<td>729</td>
<td>25 632</td>
<td></td>
</tr>
</tbody>
</table>

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 $\kappa$. Statistics are given in Table 6b.

4.6. 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 7 more time is expended when $k$ is close to $\kappa$.

When $k$ is much larger than $\kappa$ 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 8.

Table 7 compares the speeds of Algorithm 3 (SQL4) and the binary version of Algorithm 2 (BIN) run against some of the data sets. Times were averaged over five runs and then normalised against the binary version. We see that the SQL4 processed the small cube TW1 faster than BIN. However, our binary implementation effects greater speed-up as the cube size increases and the cube density decreases. In fact, we forcibly terminated processing NF1 after 19 hours.

4.7. Diamond Size and Dimensionality

The size (in cells) of the $\kappa$-diamond of the high-dimensional cubes is large, e.g. the $\kappa$-diamond for B6 captures 30% of the data. How can we explain this? Is this property a function of the number of dimensions? To answer this question the $\kappa$-diamond was
generated for each of the synthetic cubes. Estimated $\kappa$, its real value and the size in cells for each cube are given in Table 8. The $\kappa$-diamond captures 99% of the data in cubes $U_1$, $U_2$ and $U_3$—dimensionality has no effect on diamond size for these uniformly distributed data sets. Likewise, dimensionality did not affect the size of the $\kappa$-diamond for the skewed data cubes as it captured between 23% and 26% of the data in cubes $S_1$, $S_2$ and $S_3$ and between 12% and 16.5% in the other cubes. These 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.

4.8. Iterations to Convergence

In Section 3.3, 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 6a, Table 6b and Figure 7.) All cubes converged to the $\kappa$-diamond in less than 1% of $\sum_i n_i$, with the exception of the small cube $TW_1$, which took less than 7% $\sum_i n_i$. Al-
Table 7: Relative slowdown of the SQL algorithm compared to the binary implementation. Times were averaged over five runs and then normalised against the binary execution times. SQL processing for NF1 was forcibly terminated after 19 hours (\(\times\)).

<table>
<thead>
<tr>
<th>TW1</th>
<th>C1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>NF3</th>
<th>NF1</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary time (secs)</td>
<td>0.07</td>
<td>4.8</td>
<td>79</td>
<td>107</td>
<td>113</td>
<td>44</td>
</tr>
<tr>
<td>SQL / binary</td>
<td>0.85</td>
<td>2.3</td>
<td>29</td>
<td>41</td>
<td>62</td>
<td>111</td>
</tr>
</tbody>
</table>

Table 8: High dimensionality does not affect diamond size.

<table>
<thead>
<tr>
<th>Cube</th>
<th>dimensions</th>
<th>iters</th>
<th>value of (\kappa)</th>
<th>size (cells)</th>
<th>% captured</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>est.</td>
<td>actual</td>
<td></td>
</tr>
<tr>
<td>U1</td>
<td>3</td>
<td>6</td>
<td>89</td>
<td>236</td>
<td>982 618</td>
</tr>
<tr>
<td>U2</td>
<td>4</td>
<td>6</td>
<td>66</td>
<td>234</td>
<td>975 163</td>
</tr>
<tr>
<td>U3</td>
<td>10</td>
<td>7</td>
<td>25</td>
<td>229</td>
<td>977 173</td>
</tr>
<tr>
<td>S1</td>
<td>3</td>
<td>9</td>
<td>90</td>
<td>1141</td>
<td>227 527</td>
</tr>
<tr>
<td>S2</td>
<td>4</td>
<td>14</td>
<td>67</td>
<td>803</td>
<td>231 737</td>
</tr>
<tr>
<td>S3</td>
<td>10</td>
<td>14</td>
<td>25</td>
<td>208</td>
<td>260 864</td>
</tr>
<tr>
<td>SS1</td>
<td>3</td>
<td>18</td>
<td>11</td>
<td>319</td>
<td>122 878</td>
</tr>
<tr>
<td>SS2</td>
<td>4</td>
<td>19</td>
<td>7</td>
<td>175</td>
<td>127 960</td>
</tr>
<tr>
<td>SS3</td>
<td>10</td>
<td>17</td>
<td>1</td>
<td>28</td>
<td>165 586</td>
</tr>
</tbody>
</table>

Figure 9: Cells remaining after each iteration of Algorithm 2 for \(k = 1004, 1005 \) and \(1006\) on cube NF1.
Algorithm 2 required 19 iterations and 4 minutes\textsuperscript{4} to compute the 1 004-carat $\kappa$-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 $\kappa$. Figure 9 shows the number of cells present in the diamond after each iteration for 1 004–1 006 carats. The curve for 1 006 reaches zero first, followed by that for 1 005. Since $\kappa(NF1) = 1004$, that curve stabilises at a nonzero value. It takes longer to reach a critical point when $k$ only slightly exceeds $\kappa$.

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. This was not the case: in fact the opposite behaviour was observed. (See Table 8.) 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”.

5. Conclusion

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. We established upper and lower bounds on the parameter $k$ for both COUNT and SUM-based diamond cubes.

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 $\kappa$-diamond, which also resulted in an increase in running time.

We have shown that computing diamonds for large data sets is feasible. Our binary implementation fared better on large, sparse data cubes than other approaches and these results confirm that this algorithm is scalable. Further, the algorithm and the Java implementation are simple, requiring less than 275 lines of code.

5.1. Future Research Directions

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

\textsuperscript{4}averaged over 10 runs
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. These are some of the ideas to be explored in future work.

Acknowledgements

This work was supported by NSERC grants 155967 and 261437 as well as by a grant from the Québec–New Brunswick University Co-operation in Advanced Education and Research Program.

References


[40] O. Kaser, S. Keith, D. Lemire, The LitOLAP project: Data warehousing with literature, in: CaSTA’06.