Towards the Building of a Dense-Region-Based OLAP System

David W. Cheung Bo Zhou ‡ Ben Kao Hu Kan † Sau Dan Lee

Department of Computer Science, The University of Hong Kong, Hong Kong.

email: {dcheung, bzhou, kao, hukan, sdlee}@csis.hku.hk.

† Department of Automation, Tsinghua University, Beijing, China.

‡ Department of Computer Science and Engineering, Zhejiang University, Hangzhou, China.

Abstract

On-line Analytical Processing (OLAP) has become a very useful tool in decision support systems built on data warehouses. ROLAP (Relational OLAP) and MOLAP (Multidimensional OLAP) are two popular approaches for building OLAP systems. These two approaches have very different performance characteristics: MOLAP has good query performance but bad space efficiency, while ROLAP can be built on mature RDBMS technology but it needs sizable indices to support it. Many data warehouses contain many small clustered multidimensional data (*dense regions*), with sparse points scattered around in the rest of the space. For these databases, we propose that the dense regions be located and separated from the sparse points. The dense regions can subsequently be represented by small MOLAPs, while the sparse points are put in a ROLAP table. Thus the MOLAP and ROLAP approaches can be integrated in one structure to build a high performance and space efficient dense-region-based data cube. In this paper we define the dense region location problem as an optimization problem and develop a chunk scanning algorithm to compute dense regions. We prove a lower bound on the accuracy of the dense regions computed. Also, we analyze the sensitivity of the accuracy on user inputs. Finally, extensive experiments are performed to study the efficiency and accuracy of the proposed algorithm.

Keywords: Data Cube, OLAP, Dense Region, Data Warehouse, Multidimensional Data Base

1 Introduction

On-Line Analytical Processing (OLAP) has emerged recently as an important decision support technology [5, 7, 8, 11, 13]. It supports queries and data analysis on aggregated databases built in data warehouses. It is a system for collecting, managing, processing and presenting multidimensional data for analysis and management purposes.

Currently, there are two dominant approaches to implement data cubes: Relational OLAP (ROLAP) and Multidimensional OLAP (MOLAP) [2, 14, 17]. ROLAP stores aggregates in relation tables in traditional RDBMS; MOLAP, on the other hand, stores the aggregates in multidimensional arrays. In [18], the advantages and shortcomings of ROLAP and MOLAP are compared. Due to the direct access nature of arrays, MOLAP is more efficient in processing queries [10]. On the other hand, ROLAP is more space efficient for a large database if its aggregates have a very sparse distribution in the data cube. ROLAP, however, requires extra cost to build indices on the tables to support queries. Relying on indices handicaps the ROLAP approach in query processing. In short, MOLAP is more desirable for query performance, but is very space-inefficient if the data cube is sparse. In many real applications, unfortunately, sparse points are not uncommon.

The challenge here is how we could integrate the two approaches into a data structure for representing a data cube that is both query- and storage-friendly. Our observation is that in practice, even if a data cube is sparse overall, it is very often that it contains a number of small but *dense* clusters of data points. MOLAP fits in perfectly for the representation of these individual dense regions, supporting fast data retrieval. The *left-over* points, which usually represent a small percentage of the whole data cube, are distributed *sparsely* over the cube space. These sparse points would best be represented using ROLAP for space efficiency. Because of its small size, the ROLAP-represented sparse data can be accessed efficiently by indices on a relational table. Our approach to an efficient data cube representation is based on the above observation. In this paper we show how dense regions in a data cube are identified, how sparse points are represented, how dense regions and sparse points are indexed, and how queries are processed with our efficient data structure.

We remark that it has been recognized widely that the data cubes in many business applications exhibit the dense-regions-in-sparse-cube property. In other words, the cubes are sparse but not uniformly so. Data points usually occur in "clumps", i.e., they are not distributed evenly throughout the multidimensional space, but are mostly clustered in some rectangular regions. For instance, a supplier might be selling to stores that are mostly located in a particular city. Hence, there are few records with other city names in the supplier's customer table. With this type of distributions, most data points are gathered together to form some *dense regions*, while the remaining small percentage of data points are distributed sparsely in the cube space.

We define a *dense region* in a data cube as a rectangular-shaped region that contains more than a threshold percentage of the data points. To simplify our discussion, we first define some terms. We assume that each dimension (corresponding to an attribute) of a data cube is discrete, that it covers only a finite set of values.* The length of a dimension is defined as the number of distinct values covered by the dimension. We consider the whole data cube space to be partitioned into equal-sized *cells*. A cell is a *small rectangular* sub-cube. The *length* of an edge of a cell on the i-th dimension is the number of distinct values covered by the edge. The *volume* of a cell is the number of possible distinct tuples that can be stored in the cell. A *region* consists of a number of connected cells. †

*If the attribute is continuous, e.g., a person's height, we quantize its range into discrete buckets.

†Region as defined here is not necessary rectangular. However, our focus is on rectangular regions that can

The volume of a region is the sum of the volume of its cells. The density of a region is equal to the number of data points it contains divided by its volume. Now, given a density threshold ρ_{min}, a region S is dense if and only if:

- S's density $\geq \rho_{min}$;
- S is rectangular, i.e., all its edges are parallel to the corresponding dimension axes of the data cube;
- S does not overlap with any other dense regions.

If dense regions can be identified, we can combine the advantages of both the MOLAP and the ROLAP approaches to build a *dense-region-based OLAP system* in the following way. First, we store each dense region in a multidimensional array. Second, we store in a relation table all the sparse points that are not included in any dense region. Third, we build a R-tree-like index [9] to manage the access of both the dense regions and the sparse points in a unified manner. Figure 1 shows such a structure. The leaf nodes in the tree are enhanced to store two types of data: rectangular dense regions and sparse points. For the dense regions, we only store their boundaries. The data points inside each dense region are stored as a multi-dimensional array outside the R-tree. The sparse points are stored in a relational table and pointers to them from the leaf nodes are used to link up the R-tree with the sparse points. The motiviation of integrating MOLAP and ROLAP into a more space efficient and faster access strucutre for data cube has been proposed and discussed in [15]. Many different R-tree-like structures can be designed for this purpose. This approach needs an efficient technique of finding the dense regions and using them to build the access structure. Our contribution is on the development of a fast algorithm for this purpose.

As we have argued, in many practical systems, the number of sparse points is relatively small compared with the population of the whole cube. Hence, it is possible that the table and the R-tree index be held in main memory entirely for efficient query processing. For example, to answer a range query, the R-tree is searched first to locate the dense regions and the sparse points that are covered by the query. Only the related dense regions (small MOLAP arrays) are then retrieved from the database. Also, since the dense regions themselves follow the natural distributions of the data in the data cube, most routine range queries can be answered by retrieving a small number of the dense regions. A fast query response time and a small I/O cost thus ensue.

Our dense-region-based data structure has clear advantages over either the MOLAP or the ROLAP approaches. In a MOLAP system, the data cube is usually partitioned into many equal-sized chunks. Compression techniques such as "chunk-offset compression" or LZW compression [12] are used to compress the sparse chunks for storage efficiency [17]. Consequently, processing

be represented by multidimensional arrays. Therefore, in the following, when we define dense regions, we will restrict them to rectangular regions.

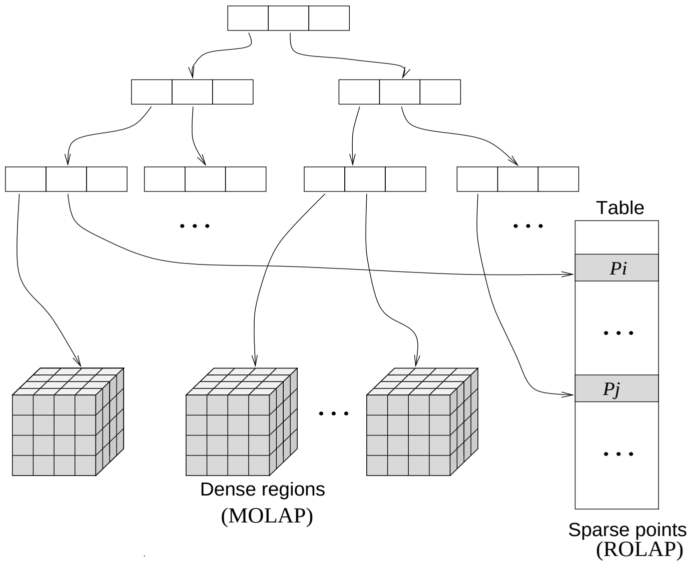

Figure 1: Dense regions and sparse points indexed by a R-tree like structure

a query involves retrieving and decompressing many chunks. On the other hand, if compression is not applied, MOLAP suffers from a very low storage efficiency and a high I/O cost.

Comparing with the ROLAP approach, our dense-region-based OLAP system inherits the merit of fast query performance from MOLAP, because most of its data are stored in multi-dimensional arrays. Our approach is also more space efficient because only the measure attributes are stored in the dense arrays, not all the attributes as in ROLAP. Moreover, ROLAP requires building "fat" indices which could consume a lot of disk space, sometimes even more than the tables themselves would take [8]. We have performed an extensive analysis on the performance gains that a dense-region-based OLAP system can achieve over the traditional ROLAP and MOLAP approaches. Due to space limitation, readers are referred to [4] for the details of the study. In the study, we shows that the dense-region-based OLAP approach is superior than both ROLAP and MOLAP in both query performance and storage efficiency. Also, we show that our approach is scalable and can thus handle very large data warehouses. Again, in this paper, our focus is on an efficient and effective algorithm for locating dense regions from a data cube — a crucial first step of our dense-region-based approach.

R-tree Access Structure

Before we move on to describe our algorithm for locating dense regions, let us point out that the R-tree access structure can be applied to data cube in two ways.

In the first way, we can assume that all aggregates in a data cubes have been computed. For a particular aggregates such as the aggregates on attributes A, B, C, D, we have a four dimensional cube containing all the aggregated values over these four attributes. An index on the dense regions in this four dimensional cube can be used to access the aggregated values for queries. At the end, we will have as many access index as the number of possible aggregates.

In the second way, we can store the measure values of all the raw data in a multidimensional array containing all the attributes. This is the base cube upon which we can build an index for the dense regions it contains. The base cube is called the *cube frame*. In the first approach, we have materialized all the aggregates; in the second one, we have materialized only the base cube. Again, our focus is to determine the dense regions in a multi-dimensional cube after data points (aggregations) have been computed.

Application

In order to establish our motivation of computing dense regions, we have investigated some real database to confirm our observation. One database that we have looked at is the Hong Kong External Trade data. It has a collection of 24 months of trade data in the period of 1995 - 1996. Total size is around 400M. It has four dimensional attributes : trade type, region, commodity and period. Trade types are import, export, re-export. Region, commodity and period are hierarchical attributes. There are 188 countries grouped into 9 regions. Trade period is organized by months, quarters and years. The commodity dimension uses both the Hong Kong Harmonized System and the Standard International Trade Classification code system. It has a four level hierarchy and 6343 commodity types in the lowest level. There are two measure attributes including commodity value and commodity quantities. When we examined this database, we saw clear dense regions. For example, in the re-export trade, there is a density region showing active export from China via Hong Kong to US on gift commodities in the period from June to Sept. We also see interesting dense regions of special type import such as special fruits from US to Hong Kong in some particular periods. On the whole, we observed that the data cube is quite sparse; however, when it is restricted to certain regions, periods, commodities, then we saw a concentration of data from which dense regions can be identified.

In the rest of the paper, we will develop an efficient chunk scanning algorithm, called **ScanChunk**, for the computation of dense regions in a data cube. ScanChunk divides a data cube into chunks and grows dense regions along different dimensions within these chunks. It then merges the found regions across chunk boundaries. ScanChunk is a greedy algorithm in that it first looks for a *seed region* and then tries to extend it as much as possible provided that all density constraints are satisfied. As we will see later, ScanChunk provides a reliable *accuracy*, leaving few sparse points outside the dense regions found.

The reminding of the paper is organized as follows. Related works are discussed in Section 2. In Section 3, the algorithm ScanChunk is described. In Section 4, we analyze the computation cost and the accuracy of the algorithm. A lower bound on the accuracy is presented together with a sensitivity analysis. An improved version of ScanChunk is presented in Section 5. Section 6 gives the results of an extensive performance study. Section 7 is the discussion and conclusion.

2 Related Works

Having established the advantages of the dense-region-based structure, our next task is to devise an algorithm to identify the dense regions given a data cube. This is a non-trivial problem, especially in a high dimensional space. Some approaches have been suggested in [10, 15]. One suggestion is to identify dense regions at the application level by a domain expert. Depending on the availability of such experts, the feasibility of this approach varies among applications. Other approaches include clusterization, image analysis techniques, and decision tree classification.

Clusterization

It is natural to consider dense region computation as a clusterization problem [15]. However, general clustering algorithms do not seem to be a good fit for finding dense regions. First, most clustering algorithms require an estimate of the number of clusters, which is difficult to determine without prior knowledge of the dense regions. Also, by definition, dense regions are non-overlapping rectangular regions with high enough density. Unfortunately, traditional clustering algorithms are not density-based. For those density-based clustering techniques such as DBSCAN [6] and CLIQUE [1], the found clusters are not rectangular. Although rectangular clusters can be obtained by finding the minimum bounding box of an irregular-shaped one, the density of the found clusters cannot be guaranteed. Furthermore, clusters with overlapping bounding boxes would trigger merges which could reduce the density of the clusters. If the region of the minimum bounding box of a cluster does not satisfy the density threshold, further processing must be performed. One possibility is to shrink the bounding box on some dimensions. However, shrinking may not be able to achieve the required density. Another approach is to split and re-cluster the found clusters by using recursive clustering techniques. Besides the high cost of performing recursive clustering, splitting could break the dense clusters into a large number of small clusters, which eventually would trigger costly merges. Finally, absorbing a few sparse points during the clusterization process could disturb tremendously the density of the found clusters. The problem is that a clustering algorithm cannot distinguish between the points that belong to some dense regions and the sparse points of the cube.

Image analysis

Some applications such as grid generation in image analysis are similar to finding dense regions in a data cube [3]. However, the number of dimensions that are manageable by these techniques is restricted to 2 or at most 3, while it is many more in a data cube. Most image analysis algorithms do not scale up well for higher dimensions, and they require scanning the entire data set multiple times. Since a database has much more data than a single image, the multiple-pass approach of image analysis clearly is not applicable in a data cube environment.

Decision tree classifier

Among the three alternatives, the decision tree classifier approach is the more applicable one in terms of effectiveness. Unfortunately, it suffers major efficiency drawbacks. For example, the SPRINT classifier [16] generates a large number of temporary files during the classification process. This causes numerous I/O operations and demands large disk space. To make things worse, every classification along a specific dimension may cause a splitting on some dense regions resulting in serious fragmentation of the dense regions. Costly merges are then performed to remove the fragmentation. In addition, many decision tree classifiers cannot handle large data sets because they require all or a large portion of the data set reside permanently in memory.

3 Dense Region Computation

In this section we give a formal definition of the dense region computation problem and the algorithm ScanChunk for finding dense regions in a data cube.

3.1 Problem definition

For convenience, we simply call all the rectangular regions in a data cube *regions*. (This would not cause any confusion in the context of finding dense region.) We denote the volume of a region r by V_r, and its density by ρ_r. (We sometimes use $\rho(r)$ to denote ρ_r for presentation.) Computing the dense regions in a d-dimensional data cube is to solve the optimization problem shown in Table 1.

Objective:	Maximize $\sum_{i=1}^{n} V_{dr_i}$, for any set of non-overlapping regions dr_i, $(i = 1, \cdots, n)$, in the cube
Constraints:	$\rho_{dr_i} \geq \rho_{min}$, $\forall i = 1, \cdots, n$;
	$V_{dr_i} \geq V_{min}$, $\forall i = 1, \cdots, n$;
	$\rho^h_{dr_i} \geq \rho_{low}, \forall i = 1, \cdots, n$, where $\rho^h_{dr_i}$ is the density of any $(d-1)$-dimension hyperplane of the region dr_i.

Table 1: Problem definition of dense region computing

In the definition, ρ_{min} is a user specified minimum density threshold a region must have to be considered dense. Since the density may not be uniform inside a dense region, the parameter ρ_{low} is specified to avoid the inclusion of empty and low density sub-regions. (Note that the user should ensure that $\rho_{low} \leq \rho_{min}$.) The parameter V_{min} is used to specify a lower bound on the volume of a dense region (to avoid trivial cases such as each data point being considered as a dense region by itself). According to the problem definition, the dense regions found and their actual densities depend on the values of these three parameters. A set of dense regions which has the maximum total volume satisfying the programming definition for a given set of parameters is called an *optimal dense regions* set. We will propose an efficient algorithm to compute the set of dense regions which satisfies all the constraints in Table 1 to *approximate* the optimal dense regions set.

3.2 The ScanChunk algorithm

Before computing the dense regions, we first build a cube frame by scanning the database once. The cube frame is a multidimensional bitmap for the data cube in which the value of a data point is either 1 or 0 indicating whether there is a valid measure attribute value associated with the point in the cube. The cube frame is then partitioned into equal-sized chunks [17]. Each chunk is further partitioned into small counting units called *cells*, and the number of points in these cells which have bitmap value equal to 1 are counted using the cube frame. Each chunk should be small enough such that the counts of all its cells can be fit into the memory. For illustration purpose, in Figure 3, we have shown a chunk with boundaries on X axis, Y axis, the lines X = L_x and Y = L_y. The chunk is divided into cells in the figure.

ScanChunk then scans the cube space chunk by chunk following a specific dimension order. In each chunk, the cells are scanned with the same dimension order. Any cell with a high enough density (larger than ρ_{min}) is used as a *seed* to grow a dense region. ScanChunk tries to expand the seed region on every dimension in both the positive and the negative directions until the seed region cannot be enlarged any further. After a dense region is found, ScanChunk repeats the growing process using another seed until all the seeds in the chunk are processed. ScanChunk then processes another chunk in the same manner. After the growing phase, ScanChunk proceeds into a merging phase to merge dense regions that are found on chunk boundaries. A high-level description of ScanChunk is shown in Figure 2.

/* $O(D_d, \cdots, D_1)$ is a specified dimension order */
for each chunk chk scanned in the dimension order $O(D_d, \cdots, D_1)$\{
    count the number of points in every cell in chk from the cube frame;
    for each cell cl scanned in chk in the dimension order $O(D_d, \cdots, D_1)$\{
        if (cl is not inside any found dense region and $\rho_{cl} \geq \rho_{min}$) then
            $grow\_dense\_region(cl)$;
    }
    perform inter-chunk merging on the dense regions found on the chunk boundary;
}
remove dense regions which are smaller than V_{min};
return the remaining found dense regions ;

Figure 2: The ScanChunk Algorithm

ScanChunk scans chunks and cells in the dimension order $\mathcal{O}(D_d, \cdots, D_2, D_1)$. Both the chunks in the cube and the cells in a chunk can be indexed like a d-dimension array. They are scanned in a row-major order, i.e., the index increases first on D_1, then on D_2, etc. We use a tuple $(x_d, \cdots, x_i, \cdots, x_1)$ to denote the cell whose lowest-corner vertex has coordinate x_i on dimension D_i, $i = 1, \cdots, n$. (We follow the row major array indexing convention.) For example, in a 2-dimensional space with dimension order $\mathcal{O}(Y, X)$, the scanning order of the cells will be (0,0), (0,1), ..., (1,0), (1,1), .., where the second indices in these cells are indices on the X dimension. We further denote a region r by $[(l_d, \cdots, l_2, l_1), (h_d, \cdots, h_2, h_1)]$ where $(l_d, \cdots, l_2, l_1)$ is its lowest-corner cell and $(h_d, \cdots, h_2, h_1)$ is its highest-corner cell. For example, $r = [(0,0), (1,2)]$ is the region containing the six cells between $(0,0)$ and $(1,2)$.

Cell size

As we will see later in Section 5, the choice of a cell size affects the efficiency and the accuracy of ScanChunk. Since it is necessary to store the point counts of all the cells of a chunk in memory for direct access to support dense region growing, there is a limit on the cell size and the number of cells a chunk can contain. A large cell size would lead to large chunks and thus a faster computation (fewer chunks). However, the *accuracy* of the dense regions found (or how close they are compared to the optimal dense regions) will be less perfect. This is because a region is measured in units of cell. Large cells give poor quantization errors. On the contrary, a smaller cell size would lead to a higher computing cost but a better accuracy. We present a detailed discussion on the accuracy of ScanChunk in Section 4.

If a cell is too small, its density would have little correlation with the density distribution in the cube. For example, if the volume of a cell, V_{cl}, is chosen to be smaller than $1/\rho_{min}$, then any cell that contains a lone spare point would have a density of $1/V_{cl} > \rho_{min}$. The cell would thus be (wrongly) chosen as a seed for growing a dense region. In order to identify useful seed regions, we thus require that $V_{cl} \cdot \rho_{min} \geq 2$. In other words, it should be large enough to contain more than one point if the distribution is uniform. We determine the cell size by computing its edge size c_k on each dimension k by:

$$c_k = \lfloor \frac{l_k}{\sqrt[d]{V_{DCS}}} \sqrt[d]{V_{cl}} \rfloor = \lfloor \frac{l_k}{\sqrt[d]{V_{DCS}}} \sqrt[d]{2/\rho_{min}} \rfloor, \tag{3.1}$$

where V_{DCS} is the volume of the cube and l_k is the length of the k-th dimension of the cube space. Intuitively, we try to shrink the cube proportionally on each dimension to form a smallest possible cell which contains more than one point.

The dense region growing procedure

Given a dimension order $\mathcal{O}(D_d, \cdots, D_2, D_1)$ and a seed, the procedure *grow_dense_region* (called by ScanChunk, see Figure 2) grows the seed into a dense region. It grows the seed first in the positive direction of dimension D_1, then the negative dimension of D_1. It repeats the same process on D_2, and D_3, etc., until all the dimensions are examined. It iterates this growing process until no expansion can be found on any dimension.

Figure 3 shows a 2-D example of the growing process. Suppose the dimension order is $\mathcal{O}(Y, X)$. ScanChunk scans all the cells in the chunk along the order: $(0,0)$, $(0,1)$, $\cdots$, $(0, L_x)$, $(1,0)$, $\cdots$, $(1, L_x)$, $\cdots$, $(L_y, 0)$, $\cdots$, (L_y, L_x). For each cell cl scanned, it checks the condition $\rho_{cl} \geq \rho_{min}$ to locate seeds. The first seed found is cell (4,4), and the procedure first expands the region along the positive X dimension until cell (4,7) is examined. Then it tries the negative X dimension but achieves no increment. The current region is the rectangle [(4,4),(4,7)]. The procedure then switches to the positive Y dimension to expand the region. After 5 steps of expansion, the region is extended to the rectangle [(4,4),(9,7)]. It then checks the negative Y dimension and found no expansion. Iteratively, the region is expanded again on the X dimension until the region becomes [(4,4),(9,9)]. After that, the region cannot be expanded any more, and the dense region found is [(4,4),(9,9)].

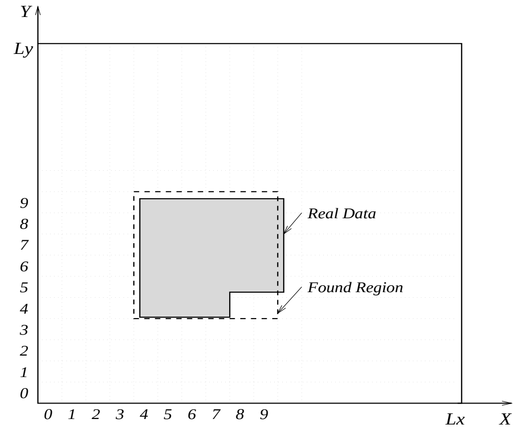

Figure 3: Dense region growing procedure

In the above example, it can be seen that the region grows faster when the procedure switches to a new dimension. For example, when it grows on the seed cell (4,4) along the X dimension, the increment is the cell (4,5). However, when the region becomes [(4,4),(4,7)] and grows in the Y dimension, the increment is the much larger rectangle [(5,4),(5,7)].

If a region $r = [(a_d, \cdots, a_2, a_1), (b_d, \cdots, b_2, b_1)]$ grows into the positive direction of dimension k, $1 \leq k \leq d$, we denote the *increment* by $\delta(r, k, 1)$. Similarly, the increment of r in the negative direction of dimension k is denoted by $\delta(r, k, -1)$. Hence, the increment $\delta(r, k, dir)$ of r on dimension k is the rectangle $[(u_d, \cdots, u_2, u_1), (v_d, \cdots, v_2, v_1)]$, where

$$\begin{cases} u_i = a_i, v_i = b_i, & \text{if } 1 \leq i \leq d, i \neq k; \\ u_k = v_k = b_k + 1, & \text{if } dir = 1; \\ u_k = v_k = a_k - 1, & \text{if } dir = -1. \end{cases}$$

For example, in Figure 3, the increment $\delta(r, 2, 1)$ of the region $r = [(4,4),(4,7)]$ along the positive Y dimension is the rectangle [(5,4),(5,7)].

The growing of a region r on dimension k is accepted when the increment $\delta(r, k, dir)$ satisfies the following two conditions:

$$\begin{cases} \rho(r \cup \delta(r, k, dir)) \geq \rho_{min} \\ \rho(\delta(r, k, dir)) \geq \rho_{low}/c_k \end{cases} \tag{3.2}$$

where $\rho(r \cup \delta(r, k, dir))$ and $\rho(\delta(r, k, dir))$ are the densities of the expanded region and the increment, respectively, and c_k is the cell length on dimension k. The parameter ρ_{low} is the density lower bound defined in Table 1.

```
/* input : seed cl, output : dense region r */
procedure grow_dense_region (input: cl, output: r){
    r := cl;
    repeat{
        for k from 1 to d do{ /* k is the dimension index */
            dir := 1 ; /* grow first on the positive direction */
            repeat{
                repeat{
                    δ := δ(r, k, dir); /* get the increment */
                    if (δ ≠ ∅ and ρ(r ∪ δ) ≥ ρ_min and ρ(δ) ≥ ρ_low/c_k)
                        then r = r ∪ δ;
                } until (r cannot be expanded anymore);
                dir := dir - 2; /* grow on the negative direction (dir = -1) */
            }until dir < −1;
        } /* for loop end */
    }until (r does not change);
    return r;
}
```

Figure 4: The grow_dense_region procedure

The first condition in equation 3.2 guarantees the basic density requirement of a dense region. The second condition controls the density of the newly added increment. On one hand, we reject an increment whose density is too small (e.g., an empty increment). On the other hand, we accept an increment if the hyperplane in the increment which touches the current region has a density larger than ρ_{low}, because this may be a boundary hyperplane of a dense region. The relaxed threshold (i.e., (ρ_{low}/c_k) instead of ρ_{low}) ensures that the border of the dense region would not be discarded. In Figure 3, the acceptance of the rectangle [(9,4),(9,9)] is due to the second condition. With the termination conditions defined, we present the growing procedure *grow_dense_region* in Figure 4.

Inter-chunk merging

Since ScanChunk grows dense regions inside chunks, some dense regions in neighboring chunks may need to be merged. For example, in Figure 5, all dense regions except A are partitioned into smaller regions across chunks. Note that merging dense regions would sometimes increase the total volume of the dense regions, rendering it closer to the optimal value. Also, with fewer (or bigger) dense regions, the R-tree built on them would be more compact and more efficient. Since the growing procedure inside a chunk is performed in a greedy fashion, there is no practical need to merge dense regions found inside a chunk. Therefore, ScanChunk only merges regions that are touching the shared boundaries between chunks.

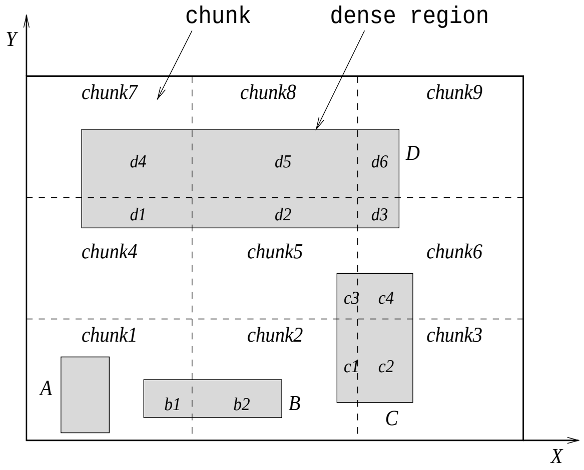

Figure 5: Merging dense regions across boundary

ScanChunk merges regions across boundary in a *level-wise* fashion following the dimension order while the chunks are being scanned. In Figure 5, assume the chunks are scanned beginning at $chunk_1$ following the dimension order $\mathcal{O}(Y, X)$. After the growing procedure is completed in $chunk_2$, merging is initiated across the boundary between $chunk_1$ and $chunk_2$ along dimension X, and the dense region B is found. After the growing procedure in $chunk_3$ is completed, merging across the boundary generates dense region $(c_1 \cup c_2)$. The same growing and merging procedure is performed on $chunk_4$ to $chunk_6$ returning dense regions $(c_3 \cup c_4)$ and $(d_1 \cup d_2 \cup d_3)$. Before the growing procedure starts in $chunk_7$, merging is now performed along dimension Y on dense regions $(c_1 \cup c_2)$ and $(c_3 \cup c_4)$ to generate dense region C. Eventually, after the growing procedure is completed in $chunk_9$, the two dense regions $(d_1 \cup d_2 \cup d_3)$ and $(d_4 \cup d_5 \cup d_6)$ are merged to form dense region D.

We briefly describe here the procedure used to merge the dense regions on the boundary of two chunks. Suppose S_1 and S_2 are the two sets of dense regions located from two neighboring chunks that touch a shared boundary. For a region $r \in S_1$, we select all the regions in S_2 which overlap with r on the boundary. We then extend the minimum bounding box containing all these selected regions recursively to include all the regions overlapping with the bounding box. If the region of the resulted bounding box is dense, we output the found dense region; otherwise, we output all the dense regions in the bounding box with no merging. We repeat the process on the remaining regions in S_1 and S_2 until both of them are empty.

4 Accuracy and Complexity

4.1 Accuracy analysis

In this section we define an accuracy measure to evaluate how close the dense regions, as computed by the ScanChunk algorithm, are from the optimal dense regions. We also prove a theoretical lower bound on the accuracy of ScanChunk.

Let S_f be the set of data points in a set of found dense regions Dr, and S_d be the set of points in the optimal dense regions. The *accuracy* $\mathcal{A}(Dr)$ of Dr is defined as $\mathcal{A}(Dr) = 1 - \frac{|S_d - S_f|}{|S_d|}$. The accuracy falls into the range of $[0, 1]$. Note that this is an aggressive approach: if all the points in the optimal regions are covered by the found regions, then the accuracy is 100%. The found regions may have included some points not in the optimal regions. However, this is acceptable because the found regions do satisfy the density thresholds imposed by equation 3.2 on both the regions and their sub-regions.

A lower bound on the accuracy

Let dr be an optimal dense region in a d-dimensional data cube. Let the length of the i-th dimension of dr be l_i, $1 \leq i \leq d$. Hence, the volume of dr is $\prod_{i=1}^{d} l_i$. Further assume that the density of dr is ρ_{dr}. Also let the length of the i-th dimension of a cell be c_i. According to the definition, the density in every $d-1$ hyperplane of an optimal dense region is not lower than ρ_{low}. To simplify the analysis, we further restrict the density distribution in every $d-1$ hyperplane of an optimal dense region such that the density of every $d-1$ sub-hyperplane of a $d-1$ hyperplane must not be smaller than ρ_{low}. Essentially, we have strengthened the condition in Table 1 so that the $d-1$ hyperplanes of an optimal dense region would not contain any low density sub-hyperplane.

The restriction made above ensures that ScanChunk can always grow inside an optimal dense region until it hits its boundary. Therefore, the worst case happens when ScanChunk discards the border hyperplanes on every side of the optimal region, and the discarded borders are slightly thinner than the dimension of a cell (Figure 6). In this case, the accuracy of the found dense region has the following lower bound:

$$1 - \frac{\rho_{dr} \prod_{i=1}^{d} l_i - \rho_{dr} \prod_{i=1}^{d} (l_i - 2(c_i - 1))}{\rho_{dr} \prod_{i=1}^{d} l_i} = \frac{\prod_{i=1}^{d} (l_i - 2(c_i - 1))}{\prod_{i=1}^{d} l_i}. \tag{4.3}$$

For example, if $d = 4$, $l_i = 50$ and $c_i = 2$, the lower bound of the accuracy would be about 85%.

Boundary conditions on accuracy

ScanChunk may discard either side of the borders of an optimal dense region dr. In Figure 7, let δ_1 and δ_2 be the increments on the two borders that ScanChunk attempts to expand to. Depending on the density distribution, three scenarios can happen: (1) Both increments δ_1 and δ_2 are absorbed. In this case, we say that the dense region dr has a *total-coverage*. (2) Only one of δ_1 or δ_2 is absorbed, and we say that dr has a *partial-coverage*. (3) None of δ_1 or δ_2 is absorbed, and we say that dr has a *weak-coverage*. **We will show that given a particular**

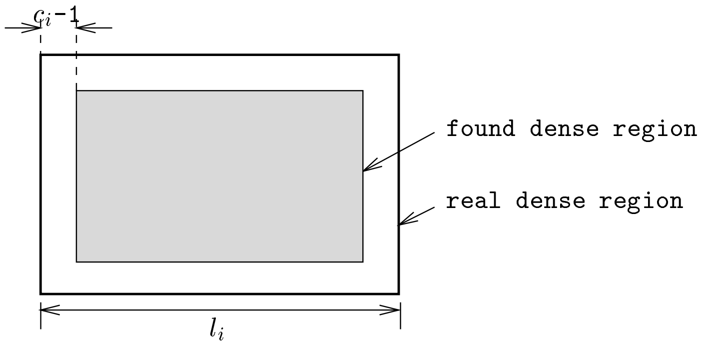

Figure 6: Accuracy of a found dense region

range of ρ_{min} relative to the value of ρ_{dr}, an optimal dense region dr will always have a total-coverage from ScanChunk.

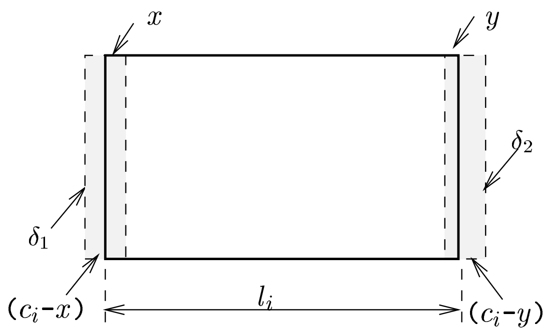

Figure 7: The sides of dense region

Theorem 1 Let dr be an optimal dense region (with the restricted condition on the $d-1$ hyperplane density distribution), whose density is denoted by ρ_{dr}. Let ρ_{min} be the user specified density threshold, and l_i and c_i be the lengths of dr and a cell in the i-th dimension, respectively.

(1) If $\rho_{min} \leq \frac{l_i}{l_i+2(c_i-1)}\rho_{dr}$, then dr has a total-coverage.

(2) If $\rho_{min} \leq (1 - \frac{c_i-1}{l_i})\rho_{dr}$, then dr has either a total-coverage or a partial-coverage.

Proof. See Appendix A. $\square$

Given a dense region dr, we have $\rho_{low} \leq \rho_{min} \leq \rho_{dr}$. Let us denote the two critical values in Theorem 1 as

$$\begin{cases} \rho_t = \frac{l_i}{l_i+2(c_i-1)}\rho_{dr} \\ \rho_p = (1 - \frac{c_i-1}{l_i})\rho_{dr} \end{cases} \tag{4.4}$$

Note that $\rho_t \leq \rho_p$ if $l_i \geq 2(c_i - 1)$. That is to say, if the dimension is partitioned into more than two cells, we are sure that the partial-coverage critical point ρ_p must be bigger than the total-coverage critical point ρ_t. It is reasonable to assume that this condition is true. The interval $[\rho_{low}, \rho_{dr}]$ thus consists of three sub-intervals $[\rho_{low}, \rho_t]$, $(\rho_t, \rho_p]$ and $(\rho_p, \rho_{dr}]$. According to the theorem, if $\rho_{min} \in [\rho_{low}, \rho_t]$, then the dense region always has a total-coverage. If $\rho_{min} \in (\rho_t, \rho_p]$, then the dense region will have either a total-coverage or a partial-coverage. If it falls into the third interval, then it may have a weak-coverage and the lowest accuracy. In general, c_i, the cell length in the i-th dimension, is much smaller than the full length of the dimension l_i. Therefore, ρ_t is very close to ρ_{dr}. In other words, the interval $[\rho_{low}, \rho_t]$ for us to pick a ρ_{min} such that the dense region has a total coverage is very large. **This shows that the growing procedure in ScanChunk will have a high probability of generating a total-coverage for the optimal dense regions defined in Table 1, provided that cells are not too large**.

Not only would the relative value of ρ_{min} affect the accuracy of the found regions, the value of ρ_{low} would also have certain effect. For example, in Figure 8, suppose sub-regions A and D have a high density of 60%, but the sub-regions B and C only have a low density of 10%. If $\rho_{min} = 30\%$, $\rho_{low} = 25\%$ and $c_i = 2$, then ScanChunk would not grow beyond regions A and D. However, if $\rho_{low} = 10\%$, then the found dense regions will cover all four regions A, B, C, D. Note that the optimal regions also contain all four regions. Therefore, if ρ_{low} is too high, then the found regions would have bad coverage. One solution is to start with a ρ_{low} that is close to ρ_{min} and then reduce it iteratively to compute dense regions until the percentage of points covered by the regions are large enough. On the other hand, ρ_{low} should not be too small else many sparse points would be absorbed into the dense regions. In the experiments we performed, we set $\rho_{low} = 0.5 \times \rho_{min}$. It was found that the dense regions found approximated the optimal dense regions very well.

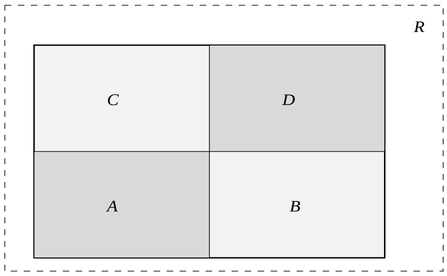

Figure 8: Effect of ρ_{low} on the accuracy

4.2 Complexity of ScanChunk

The cost of ScanChunk consists of three parts: (1) cell counting, (2) dense region growing, and (3) dense region merging.

Both the I/O and the computation cost for cell counting is linear to the total number of data points in the cube. In the growing process, a cell can be scanned and checked at most $2d+1$ times from different directions, where d is the number of dimensions of the data cube. Hence growing the dense regions takes $O\left((2d+1)\frac{V_{DCS}}{V_{cl}}\right)$ time, where V_{DCS} is the volume of the data cube and V_{cl} is the volume of a cell.

The merging cost is determined by the total number of merges and the cost of each merge. Suppose the i-th dimension of the data cube is divided into m_i partitions due to chunking. Then, there are $(m_1-1)\times m_2\times\cdots\times m_d$ merges along the first dimension. Applying this argument to all other dimensions and we have the total number of merges equal $\sum_{i=1}^{d}\left((m_i-1)\prod_{j=i+1}^{d}m_j\right)=\prod_{i=1}^{d}m_i-1=N_{chk}-1$, where N_{chk} is the number of chunks. If M is the maximum number of cells that can be stored in memory, then $N_{chk}=\frac{V_{DCS}}{M\times V_{cl}}$. Since the complexity of merging regions in two neighboring chunks is $O(N_{dr}\log(N_{dr}))$, where N_{dr} is the total number of dense regions to be merged, the total complexity of the merging process is

$$O\left(N_{dr}\log(N_{dr})\left(\frac{V_{DCS}}{M\times V_{cl}}-1\right)\right).$$

5 An optimization of ScanChunk

Since ScanChunk grows dense regions in units of cells, it could accumulate a non-trivial boundary error along the border of dense regions. We present here an optimization to improve the accuracy. The second condition in equation 3.2 is replaced by the condition: $\rho(\delta(r,k,dir))\geq\rho_{low}$. We call the regions found with this more restricted condition *cores*. Note that the cores found will be slightly smaller than the found dense regions in ScanChunk. However, their borders are guaranteed to be not too sparse (i.e., its density $\geq\rho_{low}$).

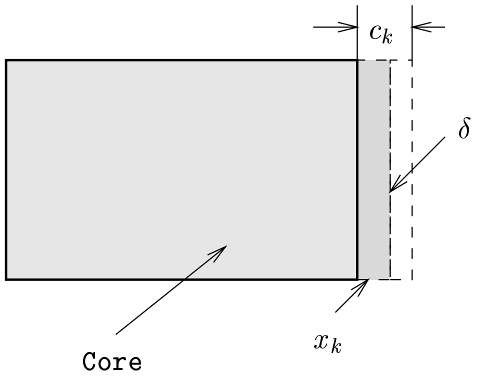

Figure 9: Estimate the border of a core

After the cores are found, we expand the borders around the cores with finer increments. In Figure 9, instead of taking in the whole increment δ, we only absorb the filled area touching the core. Note that the density of the increment δ is equal to

$$\rho(\delta)=\frac{x_k\times\rho_{dr}+(c_k-x_k)\rho_s}{c_k}, \tag{5.5}$$

where ρ_{dr} and ρ_s are respectively the densities of the dense area and the sparse area inside the increment. We can estimate x_k by using the value of $\rho(\delta)$, which can be computed from the cells in δ. From equation 5.5,

$$x_k = \frac{\rho(\delta) - \rho_s}{\rho_{dr} - \rho_s} c_k. \tag{5.6}$$

Assume that ρ_s is relatively small and since $\rho_{dr} \geq \rho_{min}$, we have

$$x_k \leq \frac{\rho(\delta)}{\rho_{min}} c_k. \tag{5.7}$$

The cores can then be grown more accurately with x_k as a finer increment. We call the algorithm with this optimization ScanChunk*. We found that ScanChunk* has a better accuracy than ScanChunk in our performance study. It runs, however, slightly slower than ScanChunk because it takes an additional step of growing the cores using finer increments.

6 Performance Studies

We have carried out extensive performance studies on a Sun Enterprise 4000 shared-memory multi-processors. The machine has twelve 250 MHz Ultra Sparc processors, running Solaris 2.6, and 1GB of main memory.

6.1 Data sets

In the performance studies, we use synthetic databases to evaluate our algorithms. The main parameters for the data set generation are listed in Table 2. The generation procedure gives the user a flexible control on all these parameters. The detailed procedure is discussed in Appendix B.

parameter	description
d	number of dimensions of the data cube
L_i	length of i-th dimension of the data cube
N_{dr}	number of dense regions
$\bar{l}_i$	average dimension length of potentially dense regions in the i-th dimension
$\bar{\rho}_{dr}$	average density of the potentially dense regions
ρ_s	sparse region density[a]

Table 2: Data generation parameters

[a] ρ_s controls the number of sparse points generated in the cube. See Appendix B

Other than the data generation parameters, ρ_{min}, ρ_{low} and V_{min} are the other inputs to the experiments. To simplify the experiment, we use a default edge size of 2 for the cells in ScanChunk and ScanChunk*, unless otherwise specified. In the performance studies, we simply refer to the average dense region density $\overline{\rho}_{dr}$ as the *dense region density*, and the minimum density threshold ρ_{min} as the *density threshold*. We also set $\rho_{low} = \rho_{min}/2$ and $V_{min} = 512$ in all the experiments.

6.2 Performance results

In the following, we present the results of the studies. Our main goal is to study the speed and the accuracy of ScanChunk in various situations, and compare it with ScanChunk*. Speed is measured by response time. For accuracy, since the optimal dense regions are not known, we have to use some related indicators. We call points in a data cube that are outside the found dense regions *sparse points*. The first indicator used to measure accuracy is the percentage of sparse points found over the total number of points in the data cube. With respect to a fixed set of parameters, a lower percentage of sparse points indicates that not too many points that should have been included into some dense regions are uncovered. The second indicator is the ratio of the volume of the found dense regions to the volume of the cube. If the percentage of sparse points found by different algorithms are very close, then the one with a smaller percentage of dense region volume has a better accuracy. This is because the algorithm has included fewer low density regions into the set of dense regions found.

6.2.1 Varying the number of dimensions

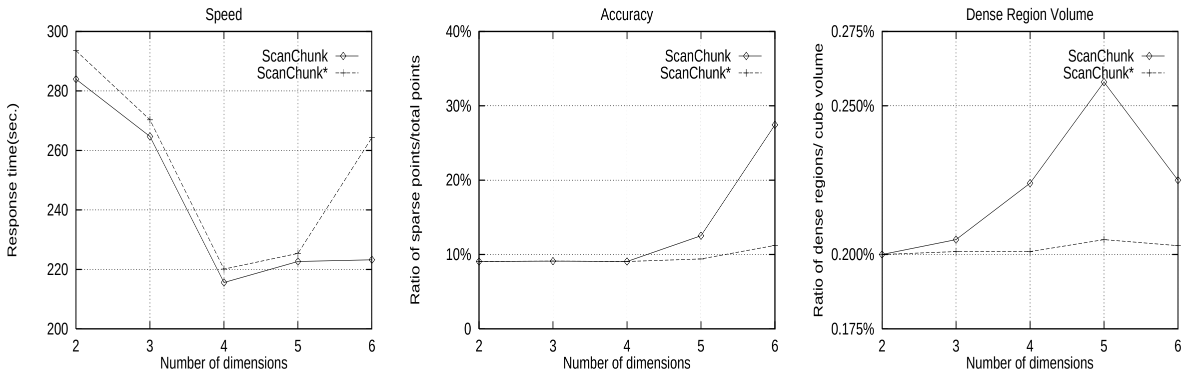

Figure 10: Effect of higher number of dimensions

In our first experiment, we fixed the size of the data cube and increased the number of dimensions from 2 to 6. The cube space was maintained at a total volume of 3×10^{10} with different dimension lengths. We set $N_{dr} = 20$, $\overline{\rho}_{dr} = 50\%$, and $\rho_s = 0.01\%$. The average volume of a potential dense region was 3×10^6, and the number of data points in the generated data cube was about 3.3×10^7, in which about $1/11$ were sparse points.

Figure 10 shows the result of running ScanChunk and ScanChunk* in these data sets with $\rho_{min} = 35\%$. The first graph shows that ScanChunk is faster than ScanChunk* in all cases as expected. As the number of dimensions increases, the cell size increases and the total number of cells decreases. Hence, the cost of growing dense regions decreases as well. On the other hand, increasing the number of dimensions would increase the number of scannings on each cell. This in turn would increase the cost of the growing phase. As a result, the response time goes down and then turns up. The response time of ScanChunk* increases more rapidly because it has to perform the additional step of fine growing dense regions from cores on border hyperplanes.

The second graph in Figure 10 shows the accuracy. In the low dimension cases, both algorithms have a percentage of sparse points around 9.1%, which is very close to 1/11, the expected sparse points ratio from the data generation. So the accuracy of both algorithms is very good in this case. However, when the number of dimensions increases beyond 4, the percentage of sparse points for ScanChunk starts to increase, which shows that its accuracy is dropping. On the other hand, ScanChunk* maintains a very stable percentage of sparse points which shows that the optimized algorithm is superior in terms of accuracy. The last graph in Figure 10 shows the ratio of the dense region volume to the cube volume. Once the number of dimension is larger than 3, the total volume covered by ScanChunk is larger than ScanChunk*. Together with the second graph, we conclude that ScanChunk absorbs more undesirable low density regions than ScanChunk* does during its growing process.

6.2.2 Varying the minimum density threshold

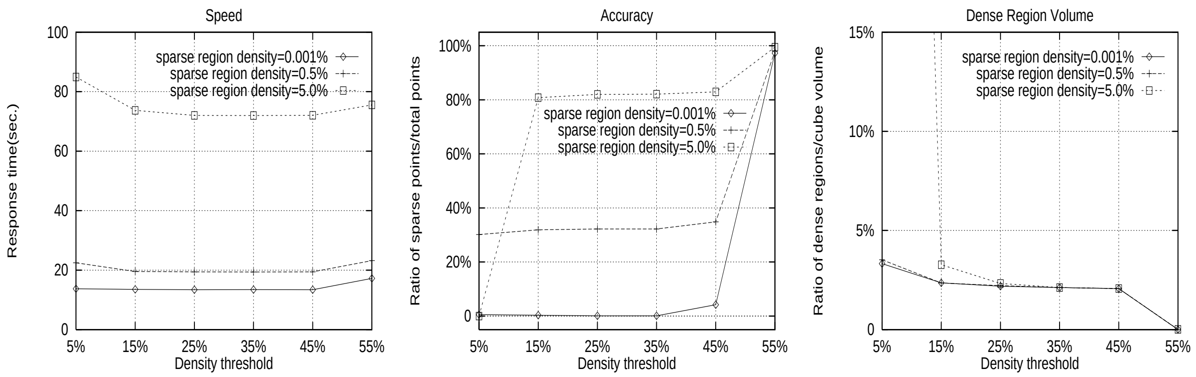

Figure 11: Effect of different density thresholds

In the second experiment we varied the value of the input density threshold ρ_{min} to study its effect on the performance of ScanChunk. We generated a 4-D cube whose space measured $300 \times 200 \times 100 \times 50$. We set $N_{dr} = 10$, $\overline{\rho}_{dr} = 50\%$, and the average size of dense regions to $50 \times 30 \times 20 \times 20$. The sparse region density ρ_s had three widely different values: 0.001%, 0.5% and 5.0%. The results over different ρ_{min} values are shown in Figure 11. For a fixed sparse region density, the first graph shows that the speed of ScanChunk is not very sensitive to ρ_{min} for the range of $\rho_{min} \in [15\%, 45\%]$. We further notice that the response time is larger when the sparse region density is bigger. This is because more sparse points are put into the data cube, leading to a higher counting cost. For $\rho_{min} = 55\%$, ρ_{min} is larger than (though very close to) $\rho_{dr} = 50\%$. In this case, ScanChunk has to spend more time trying to expand the regions in many dimensions unsuccessfully. Hence the response time has moved up. On the other hand, when ρ_{min} is small (e.g., 5%), many of the cells are taken as seeds. ScanChunk thus uses more time in growing regions. Hence, a larger response time.

The second graph in Figure 11 shows the percentage of the data points not included in any dense region found (sparse point percentage). In general, the more sparse points generated in the experiment data set (a higher sparse region density), the larger is the sparse point percentage. Moreover, we see that as long as the density threshold ρ_{min} stays below the average dense region density ($\overline{\rho_{dr}} = 50\%$), the sparse point percentage is not sensitive to ρ_{min}. This is consistent with Theorem 1 which states that the dense regions found by ScanChunk have a total coverage of the real dense regions as long as $\rho_{min} < \rho_{dr}$. Hence, no data points that should be included into some dense regions are counted as sparse points. The sparse point percentage curves thus stay flat. The only exception occurs when the sparse region density and ρ_{min} are both equal to 5%. This is a degenerated case in which "sparse" and "dense" share the same density threshold, and thus ScanChunk reports few points as sparse ones. At the opposite end of the spectrum, we see that when ρ_{min} is set to 55%, larger than $\overline{\rho_{dr}}$. The "dense regions" generated in the experiment actually do not have a density higher than the threshold value. Thus all data points are reported sparse by ScanChunk. The third graph of Figure 11 shows the total volume of the dense regions found by ScanChunk. The curves essentially show the accuracy of ScanChunk from a different perspective, and can be similarly explained. For example, the volume of the dense regions drops to zero when ρ_{min} exceeds $\overline{\rho_{dr}}$ as everything is sparse; the volume reaches 100% when both ρ_{min} and ρ_s equal 5% as everything is now dense.

6.2.3 Other results

We have performed many other experiments on ScanChunk and ScanChunk*. We briefly summarize the results here. Details of some of these experiments are discussed in Appendix C.

(1) Varying the sparse region density

We varied the number of sparse points in the data cube and studied how this affects the performance of the algorithms. We found that for small values of ρ_s, the algorithms performed very well in terms of response time and accuracy. However, when ρ_s approached ρ_{low}, the response times showed a sharper increment. This is again because the distinction between sparse and dense became unclear. Hence, ScanChunk and ScanChunk* works better when the data cube is clearly partitioned into dense regions and sparse fillings.

(2) Varying the data cube volume

We varied the volume of the data cube while keeping the total number of data points fixed. Our result was consistent with our complexity analysis in that the response time of the algorithms is linear with respect to the cube volume.

(3) Different cell sizes

In this experiment, we increased the size of a cell by uniformly increasing the length of the cell along each dimension. The result was consistent with our complexity and accuracy analysis which shows that a larger cell gives a smaller response time but lower accuracy.

(4) Varying the chunk size

We repeated our experiments using larger chunk sizes. The result showed that the accuracy of the algorithms was improved. With larger chunks, dense regions are partitioned into fewer fragments. This results in fewer dense regions merges and hence a better accuracy.

(5) Different number of dense regions

We varied the number of dense regions in our experiments. The result shows that the two algorithms have good scalability with respect to the number of dense regions.

(6) Relative performance

We have compared the relative costs of performing the three parts in building a dense-region-based data cube: cube frame building, dense region computing, and base cube construction. It was found that the cost of computing the dense regions using ScanChunk was about 20% of that of building the cube frame. On the other hand, the cost of computing the base cube was about 1.5 times the cost of building the cube frame.

7 Discussion and Conclusion

ScanChunk uses a bottom up approach, growing dense regions from seeds. The merit of this approach is that it is a localized procedure — inaccuracy in growing a dense region would not affect that in another region. In contrast, in top down approaches such as the decision tree classifier, a mild inaccuracy in the splitting procedure may impact many regions.

ScanChunk is sensitive to the choice of a cell size. Based on the approach of ScanChunk*, we can enhance the accuracy by using finer increments after the core regions are found. This refinement may require some more scannings on the cube frame. However, this is acceptable provided that information around the borders of the cores are collected so that further refinements can be done efficiently. Other candidates for iterative refinements are the choices of the density thresholds ρ_{min} and ρ_{low}. We have shown that by tuning these parameters, we can achieve a desired level of accuracy.

Another important design issue of ScanChunk is how dense regions that are cut up at chunk boundaries are merged. Even though the simple inter-chunk merging algorithm ScanChunk adopts may not perform the optimal merges in some cases, we remark that the simple algorithm is very efficient and the dense regions found are pretty accurate.

We believe that the data inside many data cubes exhibit the dense-regions-in-sparse-cube distribution. Building a dense-region-based OLAP system to compute and to manage the aggregates is thus superior than both the ROLAP and the MOLAP approaches. In this paper we have defined the dense region location problem as an optimization problem. We have developed and studied two efficient greedy algorithms — ScanChunk and its optimized version

ScanChunk* for computing dense regions. A lower bound on the accuracy of ScanChunk has been found. We have shown that the accuracy of the two algorithms is affected by the choice of the cell size and the density thresholds, and have suggested how these parameters are selected to control the accuracy of the algorithms. In particular, we have analyzed the sensitivity of the boundary errors on these parameters. Our performance studies confirm the behavior of the two algorithms.

As for future works, we will apply the results of our study to the building of a full fledge dense-region-based OLAP system, including query processing, and aggregates and data cube computing. Finally, we observe that the dense region location problem is highly related to data mining. Its solution may have important applications in mining multidimensional data.

References

[1] R. Agrawal, J. Gehrke, and D. Gunopulos. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, May 1998.

[2] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the International Conference on Very Large Databases*, pages 506-521, Bombay, India, September, 1996.

[3] M. Berger and I. Regoutsos. An algorithm for point clustering and grid generation. *IEEE transactions on systems, man and cybernetics*, 21(5):1278-1286, 1991.

[4] D. Cheung, B. Zhou, B. Kao, K. Hu and S.D. Lee. DROLAP – A Dense-Region-Based Approach to On-line Analytical Processing. *HKU CS Technical Report TR-99-02*, 1999.

[5] G. Colliat. OLAP, relational, and multidimensional database systems. *SIGMOD Record*, pages 64-69, Vol.25, No.3, September 1996.

[6] M. Ester, H. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining*, pages 226-231, Portland, Oregon, August 1996.

[7] J. Gray, A. Bosworth, A. Layman, and H. Piramish. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proc. of the 12th Intl. Conference on Data Engineering*, pages 152-159, New Orleans, February 1996.

[8] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for OLAP. In *Proceedings of the Intl. Conf. on Data Engineering*, pages 208-219, Burmingham, UK, April 1997.

[9] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 47-57, 1984.

[10] C.T. Ho, R. Agrawal, N. Megiddo and R. Srikant. Range Queries in OLAP Data Cubes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 73-88, Tucson, Arizona, May, 1997.

[11] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 205-216, 1996.

[12] T.A. Welch. A Technique for High-Performance Data Compression. In *IEEE Computer*, pages 8-19, June, 1984.

[13] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: organization of and bulk incremental updates on the data cube. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 89-99, Tucso, Arizona, May 1997.

[14] K.A. Ross and D. Srivastava. Fast computation of sparse datacube. In *Proceeding of the 23rd VLDB Conference,* pages 116-125, Athens, Greece, August, 1997.

[15] S. Sarawagi. Indexing OLAP data. *Bulletin of the technical committee on Data Engineering*, IEEE computer society, Vol. 20, No. 1, March 1997.

[16] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, pages 544-555, Bombay, India, September 1996.

[17] Y.H. Zhao, P.M. Deshpande, and J.F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 159-170, Tucson, Arizona, 1997.

[18] Y.H. Zhao, K. Tufte, and J.F. Naughton. On the Performance of an Array-Based ADT for OLAP Workloads. Technical Report CS-TR-96-1313, University of Wisconsin-Madison, CS Department, May 1996.

APPENDIX

A Proof of Theorem 1

Proof.

(1) Assume the increment in Figure 7 is done along the i-th dimension. Let x be the length of the i-th dimension of the overlapping regions of δ_1 and dr, y be that of δ_2 and dr. Note that $x, y \in [1, c_i - 1]$.

The first condition that must be satisfied in order to have a total coverage for dr is given by the following equation:

$$\frac{(l_i S)\rho_{dr}}{(l_i + c_i - x + c_i - y)S} \geq \rho_{min}, \tag{A.1}$$

where S represents the volume of the $(d-1)$ hyperplane in dr that is perpendicular to the i-th dimension. This is to ensure that the region including both δ_1 and δ_2 would have at least the required minimum density. Equation A.1 is equivalent to:

$$\frac{l_i}{(l_i + c_i - x + c_i - y)}\rho_{dr} \geq \rho_{min}.$$

For all the feasible values of $x, y \in [1, c_i - 1]$, the term $\frac{l_i}{(l_i+c_i-x+c_i-y)}$ has the minimum value of $\frac{l_i}{l_i+2(c_i-1)}$, when $x = y = 1$. Since $\frac{l_i}{l_i+2(c_i-1)}\rho_{dr} \geq \rho_{min}$, the condition in equation A.1 follows from the assumption.

What remains to be shown is that both $\rho(\delta_1)$ and $\rho(\delta_2)$ are larger than ρ_{low}/c_i. Note that $\rho(\delta_1) = \frac{x\rho_{dr}+(c_i-x)\rho_s}{c_i}$, where ρ_s is the density of the sparse region outside dr. Since $\rho_{dr} \geq \rho_{low}$, $\rho(\delta_1) \geq \rho_{low}/c_i$, and similarly, $\rho(\delta_2) \geq \rho_{low}/c_i$, therefore, the region can be expanded to include both $\rho(\delta_1)$ and $\rho(\delta_2)$.

(2) In order to have a partial coverage, one side must be absorbed and the other rejected. Suppose δ_1 is absorbed, and δ_2 is rejected, then the first condition that must be satisfied is:

$$\frac{(l_i - y)S\rho_{dr}}{(l_i - y + c_i - x)S} \geq \rho_{min}. \tag{A.2}$$

The minimum value of $\frac{(l_i-y)}{(l_i-y+c_i-x)}\rho_{dr}$ is $(1 - \frac{c_i-1}{l_i})$, when $x = 1$ and $y = c_i - 1$. Since $(1 - \frac{c_i-1}{l_i})\rho_{dr} \geq \rho_{min}$, the condition in equation A.2 follows from the assumption. The rest of the proof is the same as in (1).

B Details of the Data Generation Procedure

As mentioned in Section 6.1, the databases that we used for the experiments are generated synthetically. The data are generated by a 2-step procedure. The procedure is governed by several parameters, which give the user control over the the structure and distribution of the generated data tuples. These parameters are listed in Table 3. In the first step of the procedure, a number of non-overlapping *potentially dense regions* are generated. In the second step, points are generated within each potentially dense region, as well as the remaining space. For each generated point, a number of data tuples corresponding to that point are generated.

The data for the experiments are generated by a 2-step procedure. The user first specifies the number of dimensions (d) and the length (L_i) of each dimension of the multidimensional space in which data points and dense regions are generated. In the first step, a number (N_{dr}) of non-overlapping hyper-rectangular regions, called "*potentially dense regions*", are generated. The lengths of the regions in each dimension are carefully controlled so that they follow a normal distribution with the mean ($\bar{l}_i$) and variance given by the user.

parameter	meaning
d	no. of dimensions
L_i	length of dimension i
ρ_s	density of the sparse region
m	average multiplicity for the whole space
N_{dr}	no. of dense regions
$\bar{l}_i$	average length of dense regions in dimension i
σ_i	standard deviation of the length of d.r. in dimension i
$\overline{\rho}_{dr}$	average density of dense regions
$\overline{m}_{dr}$	average multiplicity for the dense regions

Table 3: Input parameters for data generation

In the second step, data points are generated in the potentially dense regions as well as the whole space, according to the density parameters ($\overline{\rho}_{dr}$, ρ_s) specified by the user. Within each potentially dense region, the generated data points are distributed uniformly. Each data point is next used to generate a number of tuples, which are inserted to an initially empty database. The average number of tuples per space point is specified by the user.

This procedure gives the user flexible control on the number of dimensions, the lengths of the whole space as well as the dense regions, the number of dense regions, the density of the whole space as well as the dense regions, and the size of the final database.

Step 1: generation of potentially dense regions

This step takes several parameters as shown in Table 3. The first few parameters determine the shape of the multidimensional space containing the data. The parameter d specifies the number of dimensions of the space, while the values L_i ($i = 0, 1, 2, \ldots, d-1$) specify the length of the space in each dimension. Valid coordinate values for dimension i are $[0, L_i)$. Thus, the total volume of the cube space V_{DCS} is given by

$$V_{DCS} = \prod_{i=0}^{d-1} L_i \tag{B.3}$$

The parameter ρ_s is the average density of the sparse region, which is the parts of the cube space not occupied by any dense regions. Density is defined as the number of distinct points divided by the total hyper-volume. On average, each point corresponds to m tuples in the final database. This parameter is called the "*multiplicity*" of the whole space. Therefore, the number of data tuples generated, N_t, will be

$$N_t = m \cdot N_p \tag{B.4}$$

where N_p is the total number of distinct points in the data cube.

The next parameter N_{dr} specifies the total number of potentially dense regions to be generated. The potentially dense regions are generated in such a way that overlapping is avoided. The length of each region in dimension i is a Gaussian random variable with mean $\bar{l}_i$ and standard deviation σ_i. Thus, the average volume of each potentially dense region is

$$\overline{V}_{dr} = \prod_{i=0}^{d-1} \bar{l}_i \tag{B.5}$$

The position of the region is a uniformly distributed variable, so that the region will fit within the whole multidimensional space. If the region so generated overlaps with other already generated regions, then the current region is shrunk to avoid overlapping. The amount of shrinking is recorded, so that the next generated region can have its size adjusted suitably. This is to maintain the mean lengths of the dense regions to be $\bar{l}_i$. If a region cannot be shrunk to avoid overlapping, it is abandoned and another region generated instead. If too many attempts have been made without successfully generating a new region which does not overlap with the existing ones even after shrinking, the procedure aborts. The most probable cause for this is that the whole space is too small to accommodate so many non-overlapping potentially dense regions of such large sizes.

To each potentially dense region are assigned two numbers—the density and the average multiplicity. The density of each potentially dense region is generated so that it follows a Gaussian random variable with mean $\overline{\rho}_{dr}$ and standard deviation $\overline{\rho}_{dr}/20$. This means that on average, each potentially dense region will have $\overline{\rho}_{dr} \cdot \overline{V}_{dr}$ points generated in it. The average multiplicity of the region is a Poisson random variable with mean $\overline{m}_{dr}$. These two assigned values are used in the next step of the data generation procedure.

Step 2: generation of points and tuples

The next step takes in the potentially dense regions generated in step 1 as parameter, and generates points in the potentially dense regions as well as the whole space. Tuples are then generated from these generated points according to the multiplicity values.

To generate the data, a random point in the whole space is picked. The position of the point is determined by uniform distribution. The point is then checked to see if it falls into one of the potentially dense regions. If so, it is added to that region. Otherwise, it is added to the list of "sparse points". This procedure is repeated until the number of points accumulated in the sparse point list has reached the desired value $\rho_s(V_{DCS} - N_{dr} \cdot \overline{V}_{dr})$

Next, each potentially dense region is examined. If it has accumulated too many points, the extra points are dropped. Otherwise, uniformly distributed points are repeatedly generated within that potentially dense region until enough points (i.e. $\overline{\rho}_{dr} \cdot \overline{V}_{dr}$) have been generated. After this, all the points in the multidimensional space have been generated according to the required parameters as specified by the user. The total number of points generated is the sum of the number of points generated in the sparse region as well as the dense regions. Thus,

$$\begin{aligned} N_p &= \rho_s(V_{DCS} - N_{dr} \cdot \overline{V}_{dr}) + \overline{\rho}_{dr} \cdot \overline{V}_{dr} \\ &= \rho_s \cdot V_{DCS} + N_{dr} \cdot \overline{V}_{dr} \cdot (\overline{\rho}_{dr} - \rho_s) \end{aligned} \tag{B.6}$$

Finally data tuples are generated from the generated points. For each point in a potentially dense region, a number of tuples occupying that point is generated. This number is determined by an exponentially distributed variable with mean equal to the value assigned as "multiplicity" for that region in the previous step. For each point in the sparse list, we also generate a number of tuples. But this time, the number of tuples is determined by an exponentially distributed variable with a mean which achieves an overall multiplicity of m for the whole space, so that equation B.4 is satisfied. From equations B.3, B.4, B.5 and B.6, we get

$$N_t = m \cdot \left(\rho_s \cdot \prod_{i=0}^{d-1} L_i + N_{dr} \cdot (\overline{\rho}_{dr} - \rho_s) \cdot \prod_{i=0}^{d-1} \overline{l}_i \right) \tag{B.7}$$

So, the total number of tuples (N_t) generated can be controlled by adjusting the parameters. Thus, the size of the database can be easily controlled.

C More Performance Studies

C.1 Varying the sparse region density

In section 6.2.2, we studied the effect of the minimum density threshold on ScanChunk's performance. Here, we investigate the effect of injecting different number of sparse points into the data cube by varying the sparse region density in our data generation procedure. We used the same data cube as mentioned in section 6.2.2 with the same set of parameters: $N_{dr} = 10$, $\overline{\rho}_{dr} = 50\%$, average size of dense regions $= 50 \times 30 \times 20 \times 20$. We varied ρ_s from 0.001% to 5.0%. The experiment was repeated a number of times using different values of ρ_{min} (ranging from 5% to 55%).

Figure 12 shows the response time of ScanChunk. From the figure, we see that the response time grows linearly with respect to the sparse region density except for the case when ρ_{min} is 5%. Since we set $\rho_{low} = 0.5 \times \rho_{min}$, when ρ_s approached 2.5%, many increments passed the density test and hence were processed by ScanChunk. This explains why the response time increases sharply when $\rho_s = 2.5\%$ and $\rho_{min} = 5\%$.

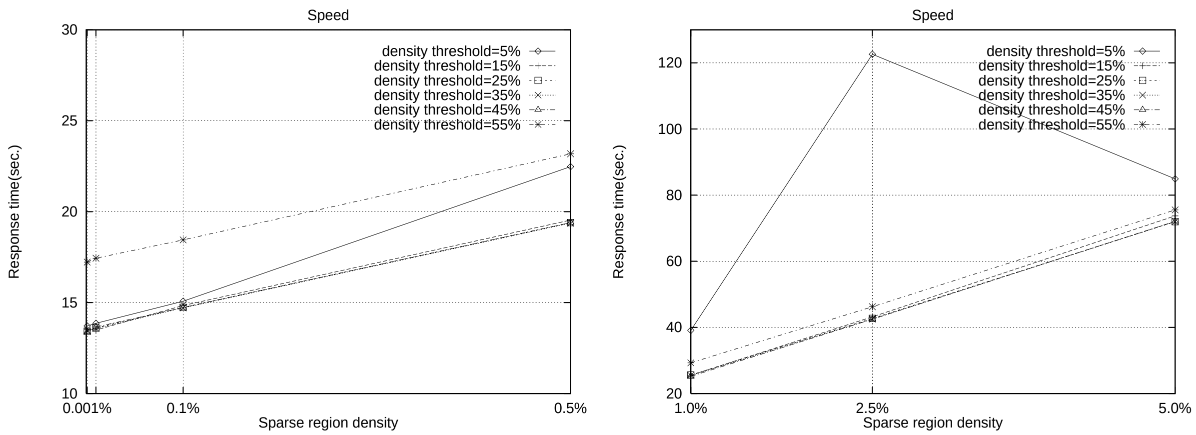

Figure 12: Effect of different sparse region densities on the speed

Figures 13 and 14 show the sparse point percentage and the sparse point volume respectively. We see that for a density threshold smaller than 50%, the sparse point percentage increases with ρ_s showing that ScanChunk did not incorrectly included the sparse points generated into dense regions. The only exception occurs when both ρ_{min} and ρ_s are 5%, since the whole cube is considered dense. For $\rho_{min} = 55\%$, the density threshold is larger than $\overline{\rho_{dr}}$. That is, not even dense regions passed the density requirement and the whole space is considered sparse.

In the experiment, we also compared the performance of ScanChunk and ScanChunk* for different sparse region densities. Figure 15 shows the result. As expected, ScanChunk* has

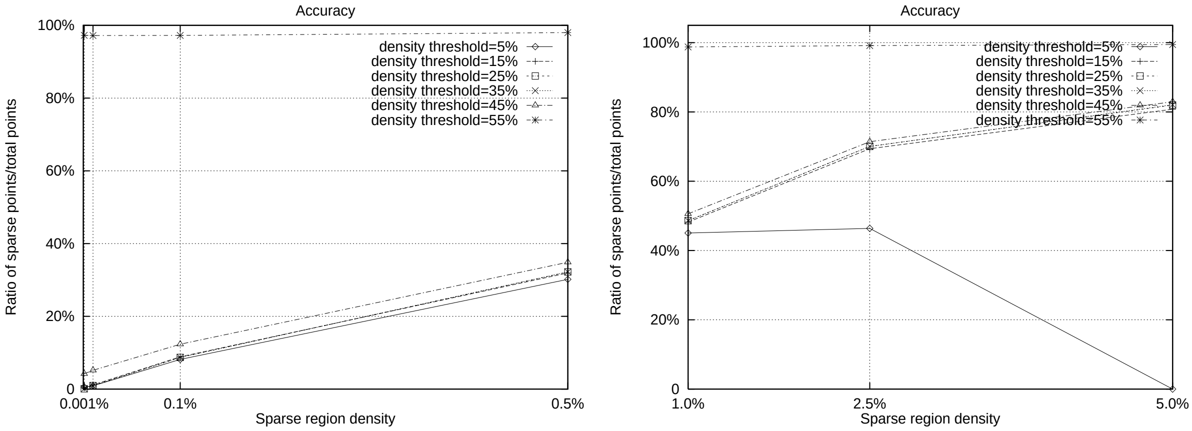

Figure 13: Effect of different sparse region densities on the accuracy

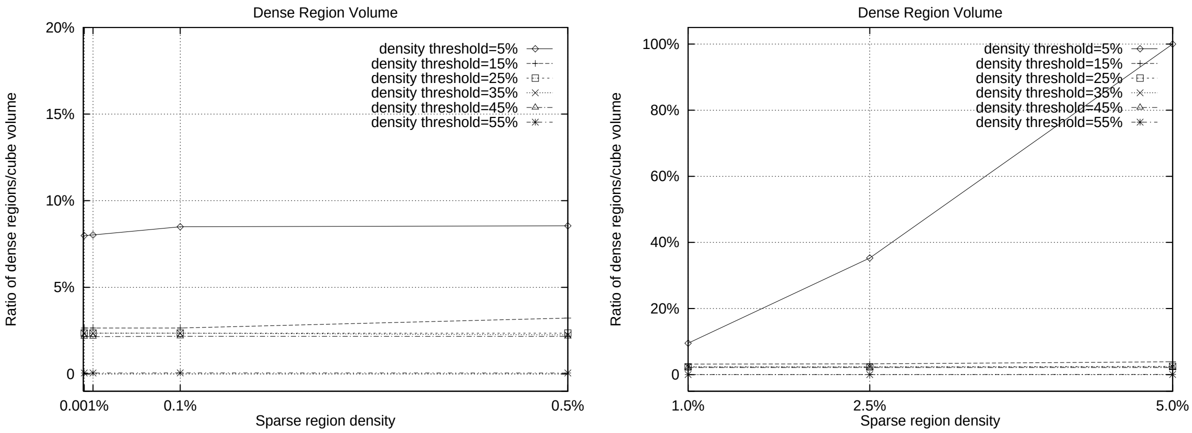

Figure 14: Effect of different sparse region densities on the dense region volume

a better accuracy but a slightly slower speed. ScanChunk gives a higher dense region volume since it uses a coarser increment over dense region boundaries.

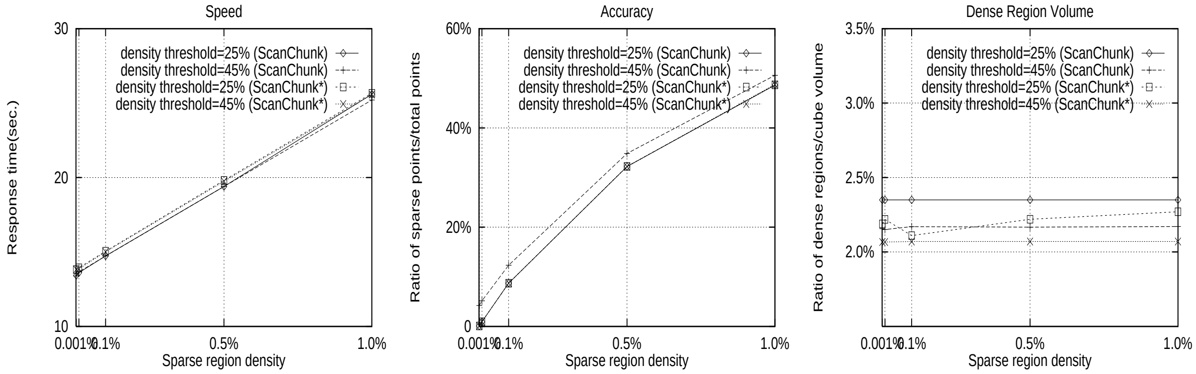

Figure 15: ScanChunk vs ScanChunk* on different sparse region densities

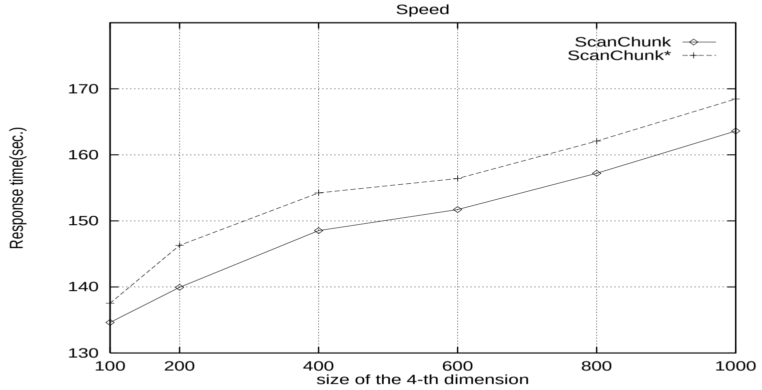

Figure 16: Speed to different of data cube volume

C.2 Effect of larger data cube volume

We have also studied the effect of increasing the volume of the data cube on the performance. We fixed the number of dimension d to 4, and the size of the first 3 dimensions to a constant, i.e., $500 \times 300 \times 200$, while varying the size of the 4-th dimension from 100 to 1000. We set $\overline{\rho}_{dr} = 50\%$, $N_{dr} = 20$ and the average dense region volume $= 3 \times 10^6$. Note that the total number of sparse points was unchanged. Therefore, the sparse region density ρ_s decreases when the cube volume increases. We fixed $\rho_{min} = 35\%$.

The response time of the algorithms is shown in Figure 16. We see that the cost of computing the dense regions increases more or less linearly with respect to the volume of the cube. This result is consistent with our complexity analysis (Section section 4.2).