

# DROLAP — A Dense-Region Based Approach to On-line Analytical Processing

David W. Cheung   Bo Zhou<sup>†</sup>   Ben Kao   Kan Hu<sup>‡</sup>   Sau Dan Lee

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

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

<sup>†</sup> Department of Computer Science and Engineering, Zhejiang University, Hangzhou, China.

<sup>‡</sup> Department of Automation, Tsinghua University, Beijing, China.

## Abstract

ROLAP (Relational OLAP) and MOLAP (Multidimensional OLAP) are two opposing techniques for building On-line Analytical Processing (OLAP) systems. MOLAP has good query performance but suffers when the data distribution in the multidimensional data cube is sparse. ROLAP can be built on mature RDBMS technology but its performance is not as competitive. Many data warehouses contain sparse but clustered multidimensional data. We propose a dense-region-based OLAP (DROLAP) system which surpasses both ROLAP and MOLAP in space efficiency and query performance. DROLAP applies the MOLAP approach on the dense regions discovered in the data, and handles the remaining small percentage of sparse points with the ROLAP approach. The core of building a DROLAP system lies in the mining of dense regions in a data cube. We have defined the dense region mining problem as an optimization problem. We show that conventional clustering techniques are not suitable for this problem, and have developed an efficient index-based algorithm EDEM to compute dense regions. Extensive performance studies have been performed. Our results clearly show that the DROLAP approach is superior than both the MOLAP and ROLAP approaches. The results also show that EDEM is efficient and effective in locating dense regions.

**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 [4, 8, 10, 13]. It supports queries and data analysis on aggregated databases built from data warehouses. It is a system for collecting, managing, processing and presenting multidimensional data for analysis and management purposes. Recently, Jim Gray et al. has introduced the *data cube* model for OLAP systems, and the **Data Cube** operator to support

multiple aggregates [6]. The **Cube** operator is an  $n$ -dimensional generalization of the group-by operator which computes all possible group-bys on  $n$  given attributes.

Currently, there are two dominant approaches to implement data cube: Relational OLAP (ROLAP) and Multidimensional OLAP (MOLAP) [1, 14, 18]. ROLAP stores aggregates in relation tables in traditional RDBMS; MOLAP, on the other hand, stores the aggregates in multidimensional arrays. In [19], the advantages and shortcomings of these two approaches are compared. Due to the direct access nature of arrays, MOLAP is more efficient in processing queries [9]. 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 performance. 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, sparsity is 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. 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. They are often “lumpy”, i.e., their data points are not distributed evenly throughout the multidimensional space, but are mostly clustered in some *dense regions*. The density of these regions are much higher than the average density of the whole space. 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.

It is easy to see that if these dense regions can be discovered some how, then each one of them can be represented by a highly dense array, and MOLAP fits in perfectly for these individual dense regions. It not only supports fast retrieval, but also has very good space efficiency. The *left-over* points, which usually constitute a small percentage of the whole data set, 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. The overhead of using pure ROLAP is much reduced. Our **Dense-Region-based OLAP (DROLAP)** approach to an efficient data cube representation is based on the above observation.

Following the above discussion, if dense regions can be identified, we can build a DROLAP system in the following way: (1) store each dense region in a multidimensional array (small MOLAP); (2) build a R-tree index on the small MOLAPs; (3) store all the sparse points not in any dense region in a ROLAP; Figure 1 illustrates such a structure. To answer a query, the R-tree index is searched to locate the relevant dense regions. The corresponding small MOLAPs are retrieved on which the query is applied as in a traditional MOLAP system. Also, the ROLAP for sparse points is searched via the supporting indices. We remark that for a data

cube with a small sparse point population, the ROLAP table and the R-tree index can possibly be stored in main memory for efficient query processing. A fast query response time and small I/O cost thus ensues.

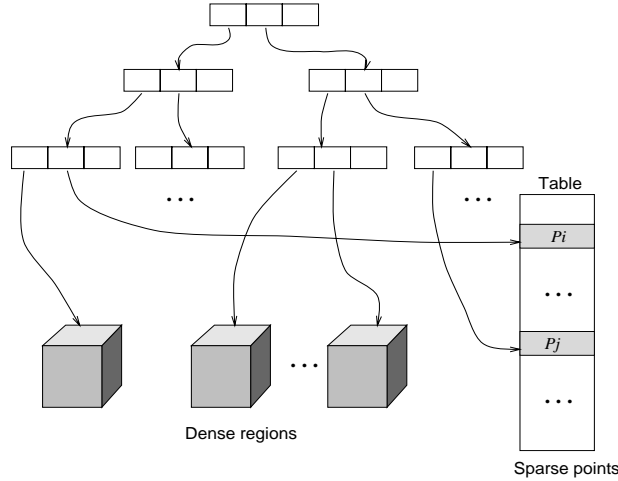


Figure 1: Dense regions and sparse points indexed by a R-tree like structure

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 is applied on sparse chunks to reduce the storage requirement [18]. Consequently, processing a query involves retrieving and decompressing many chunks. DROLAP, on the other hand, only stores (in MOLAPs) regions that are dense, thus avoiding the decompression problem.

Comparing with the ROLAP approach, DROLAP inherits the merit of fast query performance from MOLAP, because most of its data are stored in dense arrays. DROLAP 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 note that the DROLAP data organization is based on three well-studied structures, namely, MOLAP, ROLAP, and index trees. Realization is thus straight forward once dense regions are identified. Our contribution is in the study of three fundamental issues which make DROLAP possible. First, what constitutes a dense region? Second, given a data cube, how one could efficiently and effectively identify the dense regions. Third, how much performance gain (in terms of space and query response time) DROLAP can achieve over MOLAP and ROLAP.

The rest of this paper is organized as follows. In Section 2, we first define formally the dense-region-mining problem. We will show that many available techniques such as clustering or classification are inadequate in locating dense regions. A simple brute-force approach, such as traversing the whole cube checking density at each sub-space, is possible but vastly inefficient.

In Section 3 we develop an efficient algorithm **EDEM** (**E**fficient **D**ense region **M**ining) for mining dense regions in a data cube. The idea is to wisely reduce the search space to cover *only* those regions that can be parts of some dense regions. To study performance, we implemented a DROLAP system and compared it with MOLAP and ROLAP. We present the results in Section 4. Our results clearly show that DROLAP performs better than both MOLAP and ROLAP in space efficiency and query performance. Finally, we conclude our paper in Section 5.

## 2 Problem Definition and Related Works

In this section we formally define the dense-region-mining problem. 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.\* We consider the whole data cube space be partitioned into equal-sized *cells*. A cell is a *small rectangular* sub-cube. A cell that contains at least one data point is called a *valid cell*. The *volume* of a cell is the number of possible distinct tuples in the cell. A *region* consists of a number of cells. 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.

One may be tempted to define a dense region to be one whose density is larger than a certain user-supplied minimum threshold, lets say  $\rho_{min}$ . However, if the density distribution inside a region is non-uniform, anomaly occurs. For example, in Figure 2, region *C* has a high density ( $\gg \rho_{min}$ ), but the area around *C* inside region *D* is empty. (Like popcorn: not much substance on the outside, but real hard kernel within.) The overall density of region *D* could be larger than  $\rho_{min}$  and we would have regarded *D* as a dense region and we would have stored it as an array. This wastes space because storing the smaller region *C* is already sufficient. Also, any query that falls only on the region *D-C* would cause the system to retrieve and process the MOLAP representing *D*, only to find a null answer set. It thus wastes time as well.

One modification to the simple dense region definition is to require that every cell in the region be denser than  $\rho_{min}$ . The problem of this definition is that it may result in too many sparse points and too many small dense regions. For example, region *A* has a good-enough overall density, but unfortunately, a few cells of *A* just fall short of meeting the  $\rho_{min}$  density requirement (think of *A* as a piece of swiss cheese). Requiring the density of every cell of a dense region be greater than the threshold has two adverse effects. First, it would break the region up into many smaller regions, enlarging the R-tree index. Second, the data points of the “holes” are considered sparse, enlarging the ROLAP table. Both of these factors degrade the system’s performance.

The popcorn and swiss cheese anomalies can be avoided if, in addition to the total density requirement ( $\rho_{min}$ ), we also require that each cell in a region be denser than another (smaller) density threshold,  $\rho_{low}$ , for that region be considered a dense region.

---

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

Another concern about the definition is the minimum volume of a dense region. Technically, a cell whose density is larger than  $\rho_{min}$  can be considered as a dense region. Therefore, it is technically correct to consider the set of dense regions be the set of *dense cells*, and everything else goes to the sparse point ROLAP table. Although simple, this trivial approach again would result in a large R-tree index and large ROLAP table, hurting performance. To rectify, we require that each dense region be bigger than a minimum volume,  $V_{min}$ . Also, the objective of a dense-region finding algorithm should be to maximize the *coverage*, or the volume, of the dense regions found over the data points. This would reduce the number of dense regions as well as the number of sparse points, resulting in a leaner R-tree index and ROLAP table.

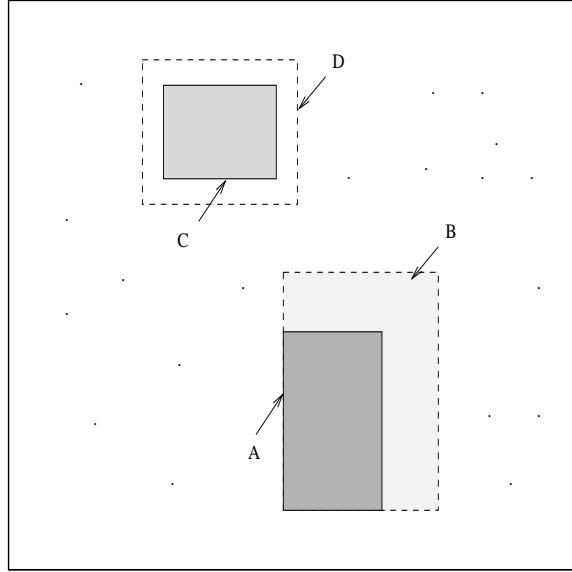


Figure 2: Two control density thresholds for dense region

## 2.1 Problem Statement

With the previous discussion, we formulate the dense region mining problem as the following optimization problem. Let  $S = D_1 \times D_2 \times \dots \times D_d$  be a  $d$ -dimension data cube such that, for  $1 \leq i \leq d$ ,  $D_i = \{x | x \in A_i, L_i \leq x < H_i\}$  is a range in a totally ordered domain  $A_i$ , bounded above and below by  $H_i$  and  $L_i$  respectively. The data cube contains a set of data points  $D = \{v_1, v_2, \dots, v_n\}$ , where  $D$  is a subset of  $S$ . The data cube  $S$  is partitioned into equal size cells such that the *cell length* on the  $i$ -th dimension is  $c_i$ . That is, the  $i$ -th dimension is divided into  $cn_i = (H_i - L_i)/c_i$  equal intervals. We use  $CP = \langle c_1, c_2, \dots, c_d \rangle$  to denote a *cell-based partition* of  $S$ . We use cell as the basic unit to reference the coordinates of regions. We use  $r = [(l_1, l_2, \dots, l_d), (h_1, h_2, \dots, h_d)]$  to denote a rectangular region representing a subspace whose projection on the  $i$ -dimension is the interval  $[L_i + c_i l_i, L_i + c_i h_i]$ . The volume of a region is the total number of data points that can be filled into the region. The density of a region  $r$

is the number of data points that fall in  $r$  divided by the volume of  $r$ . We denote the volume of a region  $r$  by  $V_r$ , and its density  $\rho_r$ . (We sometimes use  $\rho(r)$  to denote  $\rho_r$  for presentation.)

Given a data cube  $S$ , a set of data points  $D$  in  $S$ , a cell based partition  $CP$  on  $S$ , together with three input thresholds  $\rho_{min}$ ,  $\rho_{low}$  and  $V_{min}$ , computing the dense regions in  $S$  is to solve the following optimization problem:

---

Objective:	Maximize $\sum_{i=1}^m V_{dr_i}$ , for any set of non-overlapping rectangular regions $dr_i$ , ( $i = 1, \dots, m$ ), in the cube $S$
Constraints:	$\rho_{dr_i} \geq \rho_{min}, \forall i = 1, \dots, m;$ $V_{dr_i} \geq V_{min}, \forall i = 1, \dots, m;$ $\forall i = 1, \dots, m$ for each cell $cl$ in $dr_i, \rho_{cl} \geq \rho_{low}$ .

---

Table 1: Problem statement of dense region computing

The problem of mining dense regions is to find a maximal set of non-overlapping rectangular regions such that the density of each one of them is larger than  $\rho_{min}$ , volume larger than  $V_{min}$ , and the density of each cell in the regions is larger than  $\rho_{low}$ . In the above definition,  $\rho_{min}$  is a user specified minimum density threshold a region must have to be considered dense. The parameter  $\rho_{low}$  is specified to avoid the inclusion of empty and low density cell. The parameter  $V_{min}$  is used to specify a lower bound on the volume of a dense region.

## 2.2 Possible techniques

Finding dense regions in high dimensional space is a non-trivial problem. It has been suggested that techniques such as image analysis, decision tree classification, and clusterization can be used for these purpose [9, 15]. However, our investigation finds out that none of these can deliver a suitable solution.

The techniques of 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 at least 15-20 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, which makes them infeasible in handling large databases.

Decision tree classifier also suffers major efficiency drawbacks if it is used for mining dense regions. 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 required subsequently to remove the fragmentation. In addition, many decision tree classifiers cannot handle large data set because they require all or a large portion of the data set reside permanently in memory.

Dense region mining sounds and looks similar to clusterization. However, conventional

clustering technique was not designed for this purpose, and hence cannot deliver a suitable answer. Firstly, dense regions are non-overlapping rectangular regions with high enough density. General clustering algorithms produce non-rectangular clusters. Using minimum bounding boxes of the clusters may violate the density requirement, and overlapping may be introduced between the boxes. Secondly, conventional cluster algorithms are distance-based. They try to assign points to clusters by optimizing some distance-based constraint. In some cases, points on the border of a dense region may be assigned to the cluster of another dense region. Figure 3 shows a simple case. There are three dense regions and some sparse points. The conventional  $k$ -mean clustering algorithm [11], ( $k$  is set to 3), would divide the middle dense region into 3 parts, going to the 3 different clusters. This result is unacceptable for our purpose. Thirdly, conventional clustering techniques do not distinguish between points in the dense regions and sparse points; they are treated equally and are assigned to different clusters together. Hence, the sparse points could perturb the shape of the clusters significantly. It is essentially infeasible to use the results to form rectangular dense regions.

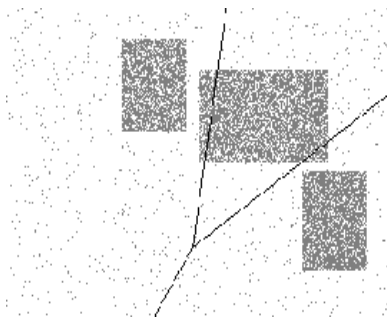


Figure 3: Use  $k$ -means clustering algorithm to find dense regions

### 2.3 Related Works

Besides the conventional clustering techniques, recently, there have been some works in the area of clusterization in large databases. CLARANS is a partition technique which improves the  $k$ -medoid methods [12]. BIRCH uses CF-trees to reduce the input size and adopts an approximate technique for clustering in large database [17]. CURE is another algorithm that uses sampling and partitioning to develop a more effective clustering technique [7]. However, all these techniques are distance-based and suffer from the same problems as the conventional technique if they were applied to solve our dense region problem. In the orthogonal direction, some density-based algorithms have been proposed recently for clustering on large databases. In the following, we will discuss their applicability in our problem.

CLIQUE is a density-based clustering algorithm [2]. Its main target is to find out high-density clusters in all potential subspaces in a multidimensional data space. It is also a cell-based algorithm. The resulted clusters are described in the form of DNF expressions. Eventhough the

dense clusters discovered by CLIQUES are constructed from dense cell, it does not guarantee the rectangle shape. Boundary of the clusters found can be saw-toothed and there may be holes inside a cluster. One way to make use of these clusters for our purpose is to apply bounding boxes on these clusters which very often would violate the density requirement. Another way to satisfy the requirement is to dissect the clusters into rectangular fragments. This fragmentation, as has been pointed out, would not be favorable in terms of query performance. Another issue of applying CLIQUE is that it requires a cluster and all its cells to satisfy one single density threshold. Therefore, it cannot deliver dense regions which have a high enough total density but have some less dense sub-regions. The last issue is the performance of CLIQUE. If we want to compute dense regions with good accuracy, then the cell size cannot be too big, and the number of cells would be very large, in particular, when the number of dimensions is high. In this case, the number of combinations that CLIQUE has to check in the first few passes could be overwhelming; also it will have to scan the database many times to go through all possible subspaces. The performance is not acceptable for our purpose.

Another density-based clustering algorithm DBSCAN has been proposed in [5]. The key idea of DBSCAN is that for each point to be absorbed into a cluster, (except those on the border), a fix size neighborhood of it must be dense — there must be enough data points in the neighborhood as measured against a threshold. DBSCAN has two nice features. (1) It can separate sparse points (noise) from the clusters. (2) It can discover clusters of unusual shapes such as those in Figure 4. For our dense region mining problem, it is possible to derive a neighborhood size and a density threshold from the given  $\rho_{min}$  for DBSCAN. However, in general, it would not be able to guarantee rectangular shape. With some modification, it may be possible for DBSCAN to deliver rectangular shape regions; however, it will be difficult to guarantee the density requirement for the clusters in that case. In particular, same as CLIQUE, DBSCAN can guarantee that the density of the neighborhoods and that of the whole cluster satisfy the threshold  $\rho_{min}$ ; however, it won't be able to find out those dense regions in which some sub-regions have density below  $\rho_{min}$  but above  $\rho_{low}$ .

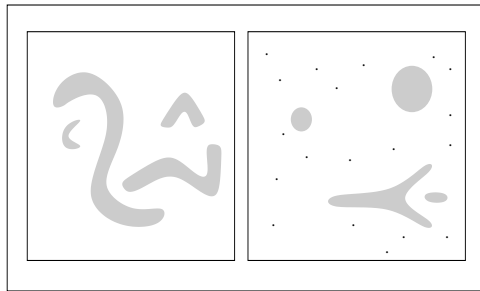


Figure 4: Clusters discovered by DBSCAN.

Eventhough many works have been done in clusterization, as has been pointed out, all distance-based technique cannot be used to solve our problem. The few proposed density-



based were neither designed for this purpose. Our contribution, besides proposing the DOLAP approach to integrate MOLAP and ROLAP, is the development of an efficient algorithm for mining dense regions in a data cube.

### 3 The EDEM Algorithm for Mining Dense Regions

A dense region is a connected set of cells each of which has a density higher than  $\rho_{low}$ . In the following, we will call this type of cells *admissible cells*. Among the admissible cells, those that have density higher than  $\rho_{min}$  is called *dense cells*. Also, any cell which is not empty is called a *valid cell*. Hence, a dense region must be a connected set of admissible cells. One way to discover dense regions is to use a multi-dimensional array to represent all the cells in the data cube and store the number of data points of each cell in the array. Then the cube space can be traversed along all possible directions to locate and grow dense regions. This is very inefficient, the array in general would be very large and could not be stored in the memory. Also, many cells in the array are empty and contribute nothing to the mining of dense regions. Another approach is to use a tree-like data structure to store and index all the valid cells. This would require much less memory space; however, locating neighbouring cells in a dense region may need to traverse many nodes on the tree. So, neither of these two approaches is an ideal solution. We have integrated these two approaches into the much more efficient algorithm EDEM.

Followings are the three main steps of the EDEM : (1) A cell-based k-d tree is built to store the valid cells in the data cube together with the number of points in each cell. We have observed that if the leaf nodes in the tree are small enough, then every dense region must touch some boundaries of some leaf nodes. Therefore, dense regions can be grown from the boundaries of the leaf nodes. (2) *Dense region covers* are then grown inside some leaf nodes from their boundaries, and these covers will contain all the dense regions in the nodes. Subsequent search of the dense regions are restricted in these covers. *Most importantly, the volume of these covers will be on the order of that of the dense regions and hence is much smaller than the whole cube.* (3) The cells in each cover can then be traversed to find out the exact dense regions in the cover. Note that most leaf nodes corresponding to sparse regions would not be involved in the covers grown in step (2). This effectively prune away most of the sparse regions in the searching in step (3) above. The algorithm EDEM has two important merits : (a) it can identify very efficiently a set of small subspaces, the covers, for finding the dense regions; (b) the searching is limited in each cover separately; there is no need to traverse between covers. In the following, we will explain the techniques in EDEM in details.

#### Build a K-d tree to store the valid cells

We build a k-d tree to store the valid cells in the data cube S. For every point in the set of data point D, the cell which it belongs to will be inserted on the tree. Besides the cell coordinates, the tree also keep track of the number of points in each cell. K-d tree is very

suitable for our dense region mining problem, because every node splitting is done along one dimension. Hence, the resulted nodes are in fact rectangular regions in the cube space such that the union of all the leaf nodes covers the whole space. The following theorem defines the splitting criteria on our k-d tree.

**Theorem 1** *Let  $V_{min}$  be the minimum volume of a dense region, and  $V_{cl}$  be the volume of a cell. Let  $R$  be a rectangular subspace in the data cube  $S$ . If the number of admissible cells in  $R$  is less than  $V_{min}/V_{cl}$ , then no dense region can be completely contained in  $R$ .*

**Proof.** Since the volume of a dense region must be larger than  $V_{min}$ , it must contain at least  $V_{min}/V_{cl}$  admissible cells. Hence no dense region can be completely contained in  $R$ .  $\square$

Following Theorem 1, we split a leaf node in the k-d tree whenever it has more than  $V_{min}/V_{cl}$  admissible cells. By doing that, we can guarantee that a dense region will always touch some boundary of some leaf node, i.e., it can never be contained in a single node without touching any boundary. Figure 5 shows the case that a dense region is split across four nodes on a k-d tree.

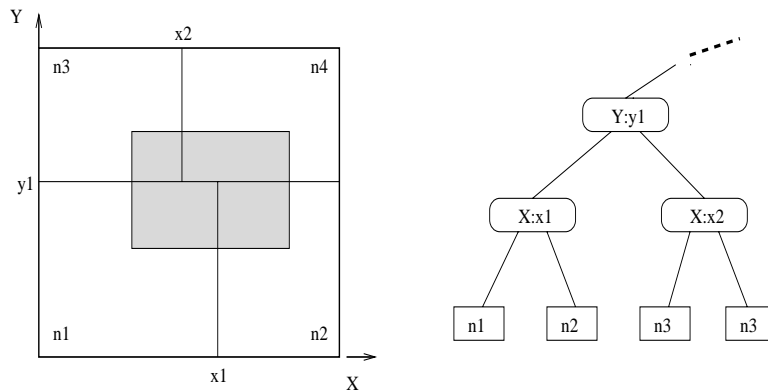


Figure 5: Dense regions split across boundary

### Grow dense region covers from boundary

Let  $R$  be the region associated with a leaf node of the k-d tree which has been built previously to store the valid cells. Let  $A$  be the set of admissible cells in  $R$ . (In the following, when we say dense region in  $R$ , we mean the part of a dense region that falls in  $R$ .) If the minimum bounding box  $MBR(A)$  of  $A$  does not touch any boundary of  $R$ , then according to Theorem 1,  $R$  cannot contain any dense region, and hence can be ignored in the mining of dense regions. On the other hand, if this is not the case, then we will grow covers from the boundary to contain the dense regions in  $R$ .

Suppose  $MBR(A)$  does interest with some boundary of  $R$ . Let  $k$  be a boundary (a  $d - 1$  dimension hyperplane) of  $R$ . Let  $B \subseteq A$  be the set of admissible cells touching  $k$ . Let  $P$  be the projection of  $MBR(B)$  on  $k$ . It is straight forward to see that the projection on  $k$  of any

dense region in  $R$  which touches  $k$  will be contained in  $P$ . Let  $X$  be the axis perpendicular to  $k$ . (Note that  $X$  has been divided into intervals by the cell partition.) Let  $F = \{I \mid I \text{ is an interval on } X, \exists \text{ a cell } c \in A - B \text{ such that the projection of } c \text{ on } X \text{ is } I\}$ . Let  $T$  be the maximal connected set of intervals in  $F$  which touches the boundary  $k$ . (The existence of  $T$  is guaranteed by  $B$  since it touches  $k$ .) The region  $P \times T$  is called the *dense region cover* grown from  $k$  in  $R$ .

**Theorem 2** *Let  $C$  be a dense region cover grown from a boundary  $k$  in a region  $R$ . If  $r$  is a dense region in  $R$  which touches  $k$ , then  $r \cap R \subseteq C$ .*

**Proof.** Follows directly from the definition of dense region cover. □

Figure 6 is an example of finding dense region cover in a leaf node. The filled cells are the admissible cells in the node. Assume the boundary  $(X, x_1)$  is the boundary from which the cover will be grown.  $P$  is the projection on the boundary, and  $T$  is the maximal connected set of intervals touching the boundary.  $P \times T$  is the cover from the boundary  $(X, x_1)$ . Note that the cover contains all dense regions which touch the boundary  $(X, x_1)$ .

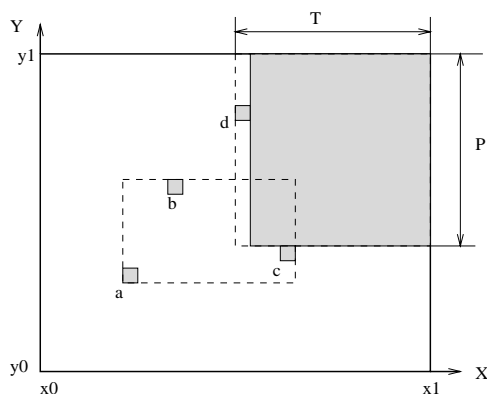


Figure 6: Finding dense region cover in the leaf node

Figure 7 is the procedure `GrowDenseRegCover` used to compute the dense region covers in a leaf node of the k-d tree. In step (8) of `GrowDenseRegCover`, the procedure `CellOnBoundary` returns the admissible cells in  $A$  that touch the boundary  $k$  and stores them in  $B$ . If  $B$  is not empty, then a dense region cover will be grown from  $k$  in step (10) with the procedure `GrowCover` according to the definition of dense region cover. After a cover has been grown, all admissible cells in the cover will be removed from the set of admissible cell  $A$ . Before the procedure is repeated on another boundary, the minimum bounding box of the remaining admissible cells not covered yet will be tested against all boundaries in step (6). If it does not touch any boundary, then no more cover is needed; and the procedure returns the found covers. This is also illustrated in Figure 6 : once the cover  $P \times T$  is removed, the MBR of the remaining admissible cells, (cells  $a, b, c$ ), would not touch any boundary; hence, no more cover is needed.

```

/* PROCEDURE : GrowDenseRegCover */
/* Input :  $N$ : a leaf node of k-d tree, with a set of valid cells */
/* Output:  $Dc$ : the set of dense region covers in  $N$  */
1)  $A = \{ \text{admissible cells in } N \}$ ;
2) let  $R$  be the region associated with  $N$ ;
3)  $\Theta = \text{set the boundaries of } R$ ;
4)  $Dc = \emptyset$ ;
5) for each boundary  $k \in \Theta$ , do {
6)   if  $\text{MBR}(A)$  does not touch any boundary in  $\Theta$  then
7)     return  $Dc$ ; /* all dense region covers have been found */
8)    $B = \text{CellsOnBoundary}(A, k)$ ; /*  $B$  contains all cells in  $A$  that touch  $k$  */
9)   if ( $B \neq \emptyset$ ) then {
10)     $D = \text{GrowCover}(A, B, R, k)$ ; /* grow cover from  $k$  */
11)     $Dc = Dc \cup \{D\}$ ; /* insert the cover found into  $Dc$  */
12)     $A = A - \{c \mid c \in D\}$ ; /* remove cells in  $D$  from  $A$  */
13)    if ( $A = \emptyset$ ) then return  $Dc$ ;
14)   }
15) }

```

Figure 7: The procedure `GrowDenseRegCover`

### Merge dense region covers

Since the k-d tree may split a dense region across several nodes, the dense region covers of the leaf nodes need to be merged at the split boundary. For example, in Figure 5, the dense region has been separated into 4 pieces at the split positions of the k-d tree. After the dense region covers in  $n_1$  and  $n_2$  have been found, they will be merged along the split position at  $X = x_1$ . The result of merging will be attached to the non-leaf nodes ( $X : x_1$ ). Subsequently, it will be merged further with the covers from  $n_3$  and  $n_4$ , and the resulted cover of the whole dense region will be attached to the node at ( $Y : y_1$ ).

When merging covers from two sibling nodes on the k-d tree, each node may have more than one covers touching the same boundary as a result of previous merging. In that case, we will merge any two from the two nodes which touch each other on the boundary and the cover will be extended to their minimum bounding box. This merging will be performed recursively until no more merging can be done between the two nodes. This merging procedure will guarantee that a dense region will not be divided up into different covers.

### 3.1 The EDEM algorithm

We have described the first two steps of EDEM above. In Figure 8, we presented the whole algorithm.

```

/* Input: S: Data cube; D: data points; CP: cell based partition;
     $\rho_{min}, \rho_{low}, V_{min}$ : thresholds;
     $O(D'_1, D'_2, \dots, D'_d)$ : dimension order.
/* Output: dense regions in S. */
/* step 1: build the k-d tree */
1)  $Tr =$  Initialize the k-d tree;
2) for each point  $p \in D$  do {
3)   if the cell  $c$  containing  $p$  has not been inserted on  $Tr$ , then insert  $c$  in  $Tr$ ;
4)   increase the count of  $c$  by 1 ;
    /* leaf nodes of  $Tr$  are splitted according to the criterion defined in Theorem 1; */
    }
/* step 2: grow dense region covers on the k-d tree */
5) for every leaf node of  $N$  of  $Tr$  do
6)    $GrowDenseRegCover(N)$ ;
    /* grow dense region covers in  $N$  */
7) for every non-leaf node of  $Tr$ , merge the dense region covers of its children;
8) assign the resulted dense region covers to  $DC$ ;
/* step 3: search dense regions in the dense covers in  $DC$  */
9)  $Dr = \emptyset$ ;
10) for each dense region cover  $dc \in DC$  do {
11)    $dr = FindDenseRegion(dc, O)$ ; /* find dense region in  $dc$ ,  $O$  is the dimension order */
12)    $Dr = Dr \cup dr$ ;
13) }
14) return  $Dr$ ;

```

Figure 8: Algorithm EDEM

EDEM has three main steps. The first step (1-4) is to read data points from  $D$  and build the k-d tree to store all valid cells. The second step (5-8) is to grow the dense region covers from the nodes on the tree. In general, we can assume that all the valid cells can be counted in the memory, because the number of cells is much smaller than the number of data points. We will have to use chunking to handle the cells if the memory is not enough. This will be discussed in Section 3.3. In the third step (8-12), we search for dense regions within each cover found. We use an array to store all the cells in a cover and use a greedy procedure  $FindDenseRegion$  to scan the cover and grow dense regions from the array. Since the covers have volumes on the

same order as the dense regions they contain, it is much smaller than the volume of the data cube. Hence, in general, we can assume that the array storing the cells in a cover can be built in the memory. Again, we discuss how to handle the case of not enough memory in Section 3.3. What remains to be discussed is the procedure FindDenseRegion.

### Procedure FindDenseRegion

Suppose  $C$  is a dense region cover. We store all the cells in  $C$ , (including cells with no data point), in a multi-dimensional array so that we can scan the cells in  $C$  to search for dense regions. The order of scanning in  $C$  is determined by a pre-selected dimension order  $O(D'_1, D'_2, \dots, D'_d)$ , i.e., first on dimension  $D'_1$ , then on  $D'_2$ , etc. During the scanning, FindDenseRegion first locates a *seed cell*, which is a dense cell, then uses the seed to grow a maximal dense region along the dimension order. After a dense region is found, FindDenseRegion repeats the searching in the reminding cells of  $C$  until all cells have been scanned. Figure 9 is the procedure FindDenseRegion.

In step 5 of FindDenseRegion, the procedure GrowRegion is called to grow a dense region from a seed. It grows the region in the same order as the scan order: first in the positive direction of  $D'_1$ ; then in the negative direction of  $D'_1$ ; then in the positive direction of  $D'_2$ ; etc.; until all dimensions and directions are examined. It iterates this growing process on all directions until no expansion can be found on any dimension. In the first dimension, GrowRegion grows a dense region by adding cells to the seed. Once after the first dimension, it grows by adding trunks of cells to the seed. We call the trunk of cells added to the dense region in each step an *increment*. If a dense region  $r = [(a_1, a_2, \dots, a_d), (b_1, b_2, \dots, b_d)]$  grows into the positive direction of dimension  $k$ , we denote the *increment* by  $\delta(r, k, 1)$ . Similiary, the increment of  $r$  on the negative direction of dimension  $k$  is denoted by  $\delta(r, k, -1)$ . GrowRegion determines the increment  $\delta(r, k, dir)$  of  $r$  on dimension  $k$  as the trunk  $[(u_d, \dots, u_2, u_1), (v_d, \dots, v_2, v_1)]$ , where

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

In step (7) of GrowRegion, the increment first grows into the positive direction, then the negative direction (step 12). It repeats this on all dimensions until  $r$  cannot grows anymore (step 15). Note that the growing is limited in the region defined by the cover. Since the covers are much smaller comparing with the whole data cube, FindDenseRegion is much more efficient comparing with scanning the whole data cube for dense regions. Note that different dimension order used in FindDenseRegion may result in different dense region configuration. However, the total volumes resulted should be very close.

## 3.2 Complexity of EDEM

In this section, we will analyse the complexity of EDEM. Let  $N$  be the number of data points, and  $N_c$  be the number of valid cells. Also, let the number of leaf node on the k-d tree  $Tr$  be  $N_d$ ,

```

/* Procedure : FindDenseRegion */
/* Input:  $C$ : a dense region cover ;
          $O$ : selected dimension order  $O(D'_1, D'_2, \dots, D'_d)$  */
/* Output:  $DR$  : set of dense regions in  $C$  */
1) build an array  $A$  to store all the cells in  $C$ ;
2)  $DR = \emptyset$ ;
3) for each cell  $c$  in  $A$  scanned in the order  $O$ , do{
4)   if ( $c$  is not inside any dense region in  $DR$  and  $\rho_{cl} \geq \rho_{min}$ ) then {
5)      $r = \text{GrowRegion}(c)$ ; /* grow dense region  $r$  from the seed  $c$  */
6)     if  $V(r) \geq V_{min}$  then
7)        $DR = DR \cup \{r\}$ ;
8)   }
9) }
10) return  $DR$ ;

/* Procedure : GrowRegion */
/* Input:  $c$ : a seed ;  $O(D'_1, D'_2, \dots, D'_d)$ : dimension order;
/* Output:  $r$  : a dense region in  $C$  */
1)  $r := c$ ;
2) repeat{
3)   for  $k$  from  $D'_1$  to  $D'_d$  do{ /*  $k$  is the dimension index */
4)      $dir := 1$  ; /* first grow on the positive direction */
5)     repeat{
6)       repeat{
7)          $\delta := \delta(r, k, dir)$ ; /* get the increment */
8)         if (all cells in  $\delta$  are admissable and  $\rho(r \cup \delta) \geq \rho_{min}$  )
9)           then  $r = r \cup \delta$ ; /* add the increment to the dense region */
11)        } until ( $r$  cannot be expanded anymore);
12)         $dir := dir - 2$ ; /* grow on the negative direction ( $dir = -1$ ) */
13)      }until  $dir < -1$ ;
14)    } /* for loop end */
15) }until ( $r$  does not change);
16) return  $r$ ;

```

Figure 9: Procedure FindDenseRegion

and  $C_p$  be the average number of valid cells in a leaf node. According to the splitting criterion and Theorem 1,  $C_p \leq V_{min}/V_{cl}$ .

The complexity of inserting a point into the k-d tree consists of two parts : (1) the time to locate a leaf node is  $\log(N_x)$ , where  $N_x$  is the number of leaf nodes; (2) the time to insert the point in the node is on the order of  $d \times \log(C_p)$ , where  $d$  is the number of dimensions. Hence the complexity of building the k-d tree is on the order of  $N \times (\log(N/C_p) + d \times \log(C_p))$ . Since  $C_p$  is bounded by  $V_{min}/V_{cl}$ ,  $O(N \times \log N)$  is an upper bound on the time complexity to build the k-d tree.

The complexity of computing the dense region covers in a leaf node is determined by the number of admissible cells in the node which is bounded by  $C_p$ . The cost of finding touching admissible cells for a boundary, minimum bounding box for a set of admissible cells, and computing a cover from a boundary, are all linear to the number of admissible cells. Therefore the time of computing all the dense region covers is  $O(N_d \times d \times C_p)$ , which is bounded by  $O(d \times N)$ .

Since the k-d tree is a binary tree, the number of nodes that require to perform dense-region-cover merging is at most  $N_d - 1$ . The complexity of the merge in each node is  $O(d \times N_x \log(N_x))$ , where  $N_x$  is the number of covers involved in each merge. In general,  $N_x$  is very small, and is bounded in the worst case by the number of dense regions, a small number in itself. Therefore, the complexity of the merging is bounded by  $O(d \times N)$ .

The procedure FindDenseRegion needs to scan all the admissible cells once in every cover. For each admissible cell, it will at most check all of its neighbouring  $2d$  cells. Hence the complexity of finding the dense regions in a cover is  $O(d \times N_c)$ , where  $N_c$  is the number of cells in the cover. Since the volume of the covers is on the same order as that of the dense regions, the time to find all dense regions is bounded by  $d \times N$ .

In summary, the complexity for the second and third steps of EDEM is linear to  $N$  and  $d$ . The dominating cost is in the building of the k-d tree, which is bounded by  $O(N \times \log(N/C))$ . This shows that EDEM is a very efficient algorithm.

### 3.3 Memory Limitation

In general, because we only record cell information not the data points on the k-d tree, the memory space required to store the tree should not be too big. This also depends on the cell size. However, if there is not enough memory, the tree can be stored on disk, and the computing of dense region covers can be performed separately on different branches. Covers from different branches can be merged afterward.

In the last step of finding dense regions from their covers, since dense regions have high density and their covers have same order of capacity, the array built from a cover in general should not be too big to fit in the memory. However, if it happens that a cover cannot be fully contained in the available memory, then chunking can be used to partition the cover.



Dense regions can be computed in each chunk separately [19]. At the end, an additional step of merging the dense regions found from the chunks is required.

## 4 Performance Studies

We have carried out extensive performance studies on a Sun Sparc 5 workstation running Solaris 2.6 with 64M main memory. Our first goal is to study the space efficiency and query performance of the DROLAP system by comparing it with both the MOLAP and ROLAP. The second goal is to study the speed and effectiveness of the EDEM algorithm.

### 4.1 System implementation and data generation

We implemented a MOLAP system using chunking. The data cube is partitioned into equal-size chunks. The chunk size is selected to utilize as much the available memory as possible to achieved a good query performance. Sparse Chunks are compressed using the chunk-offset technique; and the other chunks are stored as arrays. In our experiments, any chunk whose density is less than 20% is treated as a sparse chunk. Note that even a very sparse chunk needs at least one page to store it. Therefore, in a sparse data cube, there could be many low density pages. In the processing of a range query, chunks overlapping with the range are retrieved into the memory to answer the query.

For the ROLAP system, we store all the data points in a table randomly and build disk-based B+ tree index on each dimension. To answer a range query, search will be performed on the index of every dimension separately. Intersection on the results of the searching will then be performed to compute the answers of the query.

In the DROLAP system, we use an in-memory R-tree index to manage the dense regions. Each dense region is stored as an array on disk, and the sparse points are organized and accessed in the same manner as the tables in the ROLAP system. A query is processed on both the R-tree and the ROLAP table.

In the performance studies, we use synthetic databases to do the comparison. Parameters for data generation are listed in Table 2. The generation procedure gives the user flexible control on all these parameters. The detail procedure is in Appendix A.

The queries in our experiments are range queries generated randomly. Each query is corresponding to a rectangular region in the cube. Since we assume that the distribution of the dense regions in the cube reflects the distribution of the data, substantial percentage of the queries can be answered by a single dense region. To simulate this query distribution, we generate two types of queries : those that fall randomly into one dense region; those that appear randomly anywhere in the cube space. We call the first type of queries, *region query*, and the second type of queries, *space query*.

parameter	
$d$	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
$\rho_s$	sparse region density

Table 2: Data generation parameters

## 4.2 Performance results of DROLAP

We have compared the space efficiency and query performance of the DROLAP system with those of the MOLAP and ROLAP system in several situations. Query performance is measured by total response time and number of pages accessed. In each experiment, we took the measurement over the execution of 100 range queries. Space efficiency is measured by the *expansion ratio* of each OLAP system, which is the storage required divided by the size of original data set.

### Effect of the percentage of sparse points

In this experiments, we examine the impact of sparse points in the performance of DROLAP. We generated a database with 1 million data points in a 3-dimensional data cube which has a volume of  $2 \times 10^{10}$ . We varied the percentage of sparse points from 1% to 30%, and the percentage of region queries is 80%. Figure 10 shows the result.

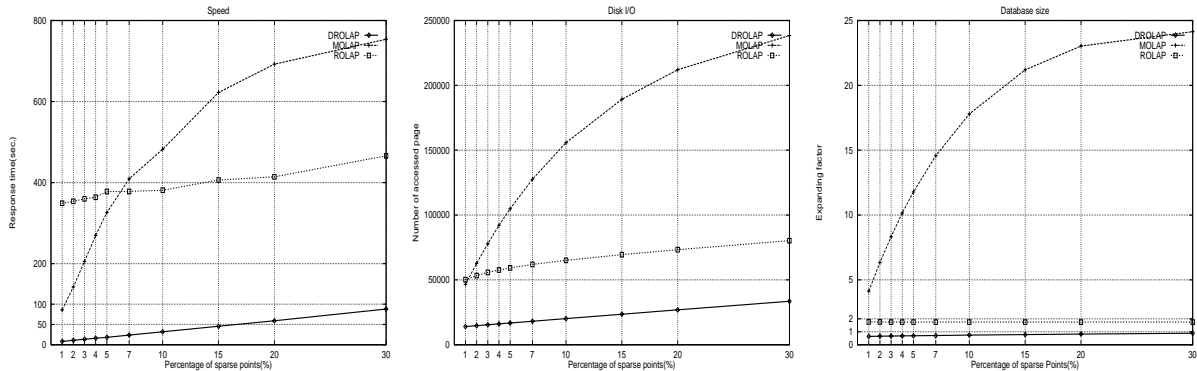


Figure 10: Effect of percentage of sparse points

The first figure is the total response times of 100 range queries. The second is the number of I/O pages. DROLAP is clearly superior than MOLAP or ROLAP. DROLAP's response time and disk I/O are linear to the percentage of sparse points, and increase very slowly. The increase in response time in DROLAP is attributed mainly to the processing on the sparse points table (the ROLAP table).

The response time of ROLAP is about 10 - 20 times higher than DROLAP, and its I/O is

about 3 - 4 times that of DROLAP. The result shows that ROLAP demands more computation in processing queries on its indices. It is interesting to note that the rate of increase in the response time on the ROLAP curve is very close to that on the DROLAP curve. This again shows that the increase in DROLAP response time is largely due to its ROLAP part.

The response time and I/O of MOLAP increase rapidly when the percentage of sparse points increases. When the percentage approaches to 30%, the increases start to slow down. When the percentage of sparse points increases initially, more data points are scattered around into the sparse region, which leads to fewer empty chunks. Hence, more I/O would be required for query processing. However, when the percentage reaches a critical value, the number of non-empty chunks increases very little and the I/O becomes stable. Hence, the response time starts to level off.

The results overall show that MOLAP has an intrinsic problem of low storage efficiency and needs more I/O. If the storage system has a slow I/O, MOLAP will be the least favorable. As for ROLAP, its efficiency is affected by the processing on the indices. The results convincingly demonstrated the merits of DROLAP. DROLAP has limited the shortcoming in ROLAP by building indices on a much smaller table containing only the sparse points. On the other hand, array access is built only on the dense regions which drastically reduces the storage requirement comparing with MOLAP. Also, the results show that DROLAP maintains a strong edge even when the percentage of sparse points is rather high.

The third graph in Figure 10 is the database *expansion ratio* of the three OLAP systems. DROLAP has the smallest storage space overhead. Its storage size is smaller than that of the original database because only the measure attributes are stored in the dense region arrays. The ratio for ROLAP is about 1.75. The extra 75% is due mainly to the indices. MOLAP has the highest ratio, this again is due to the large overhead in the chunking. For example, when the sparse points percentage is 20%, MOLAP needs 28 times more storage than DROLAP.

### **Effect of the database size**

We have studied the effect of the size of the input data on the performance. We varied the number of data points from 0.25 to 2.5 million, (5% of sparse point in each case), in a 3-dimensional data cube. Figure 11 is the result. DROLAP is consistently better than MOLAP and ROLAP; and ROLAP's performance deteriorates much faster than MOLAP. Even though the response time in MOLAP increases as the input size increases, the rate of increase does slow down after the size has reached a critical value. The reason is that the number of chunks required for a query does not change much once the database size has surpassed the critical value. On the other hand, once the data size becomes rather big, processing queries on the indices become very slow for ROLAP. Therefore, MOLAP is more efficient than ROLAP if the data cube has a very high data density. Overall, DROLAP is consistently the winner with very efficient performance. On the other hand, we anticipate that the performance of DROLAP and MOLAP would converge if the density of the whole cube is extremely high.

### **Effect of higher number of dimensions**

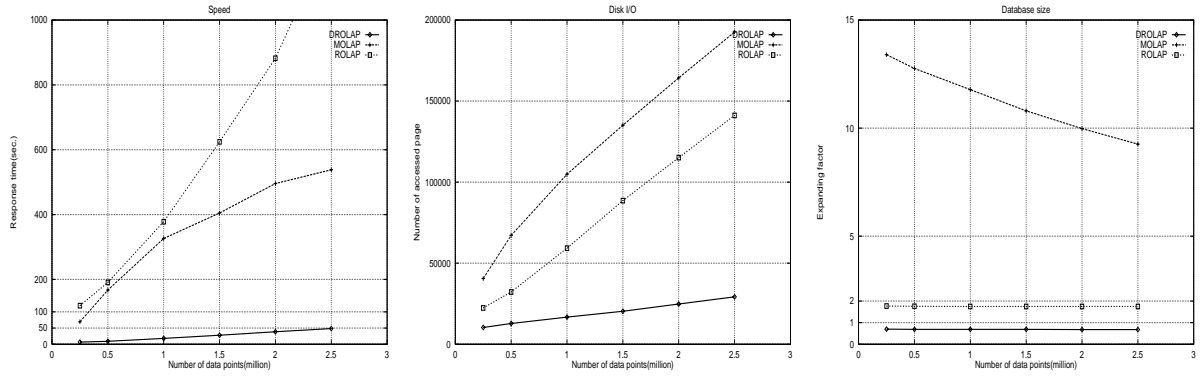


Figure 11: Effect of the database size

In this experiment, we fixed the size of the data cube and increased the number of dimensions from 2 to 6. To be fair, we also have fixed the total volume of dense regions involved in the queries. The I/O of MOLAP increases rapidly, because the number of chunks that a range query may need to retrieve increases with the dimension. On the other hand, ROLAP needs to perform searches on more indices in case the number of dimensions has increased. In total, ROLAP is faster than MOLAP in lower dimension, but MOLAP surpasses ROLAP in the higher dimension case. In all cases, DROLAP performs much better than both of them.

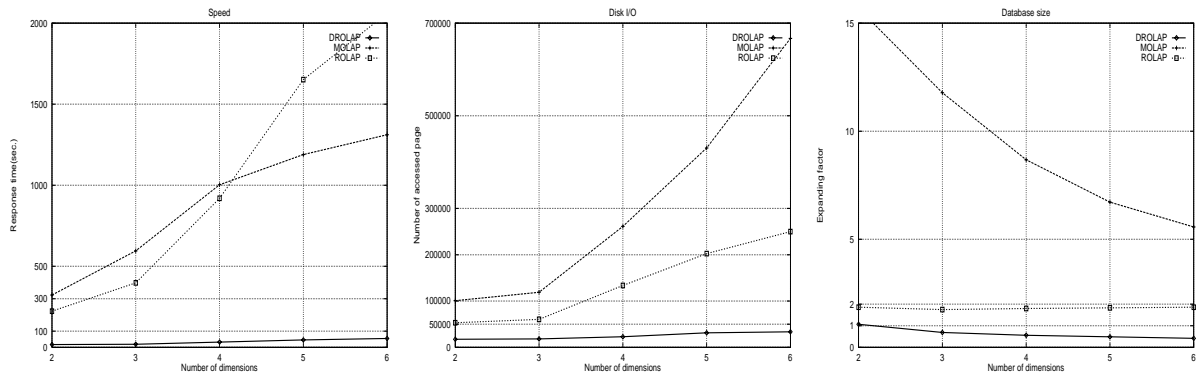


Figure 12: Effect of higher number of dimensions

### Effect of different query ratios

In a fixed 3-dimensional data cube with 1 million data points, (5% sparse points), we varied the percentage of space query from 0% to 100%. In Figure 13, the response time of DROLAP is linear to the percentage of space query. The result shows that the performance advantage in DROLAP is unaffected by the increased percentages of queries not directly addressed into dense regions. In this experiment, we found that DROLAP is consistently faster than MOLAP and ROLAP 16 times and 20 times respectively in almost all cases.

In conclusion, we found out that DROLAP performs much better than the other two OLAP systems in both space efficiency and query performance. In particular, it is superior in the case that the dimension is high, the data set is large and there is certain level of sparsity.



Figure 13: Effect of the percentage of whole space query

### 4.3 Performance of EDEM

In the following, we present the results of the performance studies on the EDEM algorithm. The main purpose is to study the speed and effectiveness of EDEM in mining dense regions. Speed is measured by response time, starting from the building of the k-d tree.

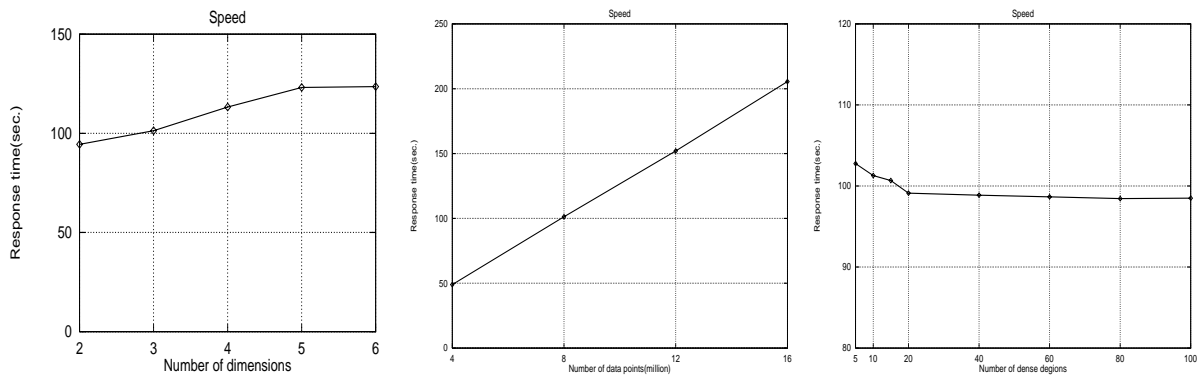


Figure 14: Performance studies of algorithm EDEM

In our experiments with EDEM, we have set  $\rho_{min} = 20\%$ ,  $\rho_{low} = 10\%$ , and  $V_{min} = 2048$ . We first examine the behavior of EDEM in cubes with different number of dimensions. We fixed the size of the data cube and increased the number of dimensions from 2 to 6. The total number of data points are close to 1 millions with 10 dense regions, and 5% of the data points are in the sparse regions. The first graph in Figure 14 is the result of running EDEM on these data sets. It shows that the response time is almost linear to the number of dimension as predicted in our complexity analysis.

In the second experiment, we studied the effect of increasing the size of input database on the performance. We varied the the size of the database from 0.5 to 4 million data points, (5% of sparse point in each case), in a 3-dimensional data cube. The second graph in Figure 14 is the result. It shows that the response time is linear to the data size, which again is compatible to our analysis.

In the third experiment, we examine the effect on EDEM if the number of dense regions

increases. We varied this number from 5 to 100 in a 3-dimensional data cube, while the total volume and number of data points of the dense regions do not change. The result shows that the performance of EDEM is not affected by the number of dense regions. In fact, as the number increases, the volume of each dense region is decreased. This reduces the amount of splitting of the dense regions across the k-d tree nodes. Hence, the response time of mining were improved slightly.

In terms of effectiveness, all of the dense regions in the data set were found in the output of EDEM. Something worthy to mention : the dense region covers found in the second step of EDEM in most cases are very close to the dense regions found at the end. It shows that the algorithm is very effective in reducing the search domains for dense regions.

## 5 Discussion and Conclusion

Many data cubes have data distribution which contains some dense regions and a small percentage of sparse points. DROLAP is a much more efficient approach for building an OLAP system in these cubes comparing with other approaches.

In this paper, we have made the following contributions: (1) Proposed the DROLAP approach and a data structure to support the processing of queries on a DROLAP system. (2) Defined the problem of mining dense regions. (3) Shown that conventional clustering technique is not suitable for finding dense regions. (4) Proposed a cell-based algorithm EDEM to compute dense regions. (5) Performed an in-depth performance study which shows that DROLAP is in fact superior than both MOLAP and ROLAP.

EDEM first scans the data base to build up a k-d tree to store the valid cells in the cube. It then uses a bottom up approach to compute the dense regions from the tree. The complexity of finding the dense regions is linear to both the database size and the number of dimensions, which gives the algorithm good scalability.

In computing dense regions, EDEM wisely avoids the traversal of the whole data cube. It keeps an upper bound on the size of the leaf nodes of the k-d tree so that each dense region must touch the boundary of some node. It then grows covers to contain all the dense regions with a linear algorithm. Since the volume of the cover is on the order of the volume of the dense regions, this significantly reduces the search space for dense regions. In fact, EDEM combines the top-down, bottom-up, and greedy approaches in one algorithm. It builds up the k-d tree top down from the database; then uses a bottom-up approach to grow the dense region covers; finally, it uses a greedy algorithm to compute the dense regions in each cover.

The performance studies have demonstrated that DROLAP is superior than both ROLAP and MOLAP if the data distribution has the assumed characteristics. We also have shown that EDEM has good scalability. As for future works, we observe that the dense region mining problem is highly related to data mining. Its technique may have important applications in the mining of multidimensional data.

## References

- [1] S. Agrawal, 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.
- [2] 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.
- [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] G. Colliat. OLAP, relational, and multidimensional database systems. *SIGMOD Record*, pages 64-69, Vol.25, No.3, September 1996.
- [5] 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.
- [6] J. Gray, A. Bosworth, A. Layman, and H. Piramish. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceeding of the 12th Intl. Conference on Data Engineering*, pages 152-159, New Orleans, February 1996.
- [7] S. Guha, R. Ratogi, and K. Shim. CURE: An Efficient Clustering Algorithm for Large Databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, May 1998.
- [8] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for OLAP. In *Proceedings of the 13th Intl. Conference on Data Engineering*, pages 208-219, Birmingham, UK, April 1997.
- [9] 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.
- [10] 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, Montreal, Quebec, June 1996.
- [11] L. Kaufman, and P.J. Rousseeuw. *Finding Groups in Data : An Introduction to Cluster Analysis*, 1990. John Wiley & Sons.

- [12] R.T. Ng, and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. In *Proc. of the 20th Int'l Conference on Very Large Databases*, pages 144-155, Santiago, Chile, 1994.
- [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, Tucson, Arizona, May 1997.
- [14] K.A. Ross and D. Srivastava. Fast computation of sparse datacube. In *Proc. of the 23rd Int'l Conference on Very Large Databases*, 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] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH : An Efficient Data Clustering Method for Very Large Databases In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 103-114, Montreal, Quebec, June 1996.
- [18] 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, May 1997.
- [19] 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 Details of the Data Generation Procedure

As mentioned in Section 4.1, 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.

In the second step, data points are generated in the potentially dense regions as well as the whole space, according to the density parameters ( $\bar{\rho}_{dr}, \rho_s$ ) 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  $L_i$  ( $i = 0, 1, 2, \dots, d - 1$ ) 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

$$V_{DCS} = \prod_{i=0}^{d-1} L_i \quad (\text{A.1})$$

The parameter  $\rho_s$  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

parameter	meaning
$d$	no. of dimensions
$L_i$	length of dimension $i$
$\rho_s$	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 $i$
$\sigma_i$	standard deviation of the length of d.r. in dimension $i$
$\bar{\rho}_{dr}$	average density of dense regions
$\bar{m}_{dr}$	average multiplicity for the dense regions

Table 3: Input parameters for data generation

number of data tuples generated,  $N_t$ , will be

$$N_t = m \cdot N_p \quad (\text{A.2})$$

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  $\sigma_i$ . Thus, the average volume of each potentially dense region is

$$\bar{V}_{dr} = \prod_{i=0}^{d-1} \bar{l}_i \quad (\text{A.3})$$

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  $\bar{\rho}_{dr}$  and standard deviation  $\bar{\rho}_{dr}/20$ . This means that on average, each potentially dense region will have  $\bar{\rho}_{dr} \cdot \bar{V}_{dr}$  points generated in it. The average multiplicity of the region is a Poisson random variable with mean  $\bar{m}_{dr}$ . 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  $\rho_s(V_{DCS} - N_{dr} \cdot \bar{V}_{dr})$

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.  $\bar{\rho}_{dr} \cdot \bar{V}_{dr}$ ) 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,

$$\begin{aligned} N_p &= \rho_s(V_{DCS} - N_{dr} \cdot \bar{V}_{dr}) + \bar{\rho}_{dr} \cdot \bar{V}_{dr} \\ &= \rho_s \cdot V_{DCS} + N_{dr} \cdot \bar{V}_{dr} \cdot (\bar{\rho}_{dr} - \rho_s) \end{aligned} \quad (\text{A.4})$$

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 A.2 is satisfied. From equations A.1, A.2, A.3 and A.4, we get

$$N_t = m \cdot \left( \rho_s \cdot \prod_{i=0}^{d-1} L_i + N_{dr} \cdot (\bar{\rho}_{dr} - \rho_s) \cdot \prod_{i=0}^{d-1} \bar{l}_i \right) \quad (\text{A.5})$$

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.