Region Clustering Based Evaluation of Multiple Top-*N* Selection Queries

Liang Zhu [a, b], Weiyi Meng [c, *], Wenzhu Yang [a], Chunnian Liu [b]

[a] *School of Mathematics and Computer Science, Hebei University, Baoding, Hebei 071002, China*
[b] *College of Computer Science and Technology, Beijing University of Technology, Beijing 100022, China*
[c] *Department of Computer Science, State University of New York at Binghamton, Binghamton, NY 13902, USA*

ABSTRACT

In many database applications, there are opportunities for multiple top-*N* queries to be evaluated at the same time. Often it is more cost effective to evaluate multiple such queries collectively than individually. In this paper, we propose a new method for evaluating multiple top-*N* queries concurrently over a relational database. The basic idea of this method is region clustering that groups the search regions of individual top-*N* queries into larger regions and retrieves the tuples from the larger regions. This method avoids having the same region accessed multiple times and reduces the number of random I/O accesses to the underlying databases. Extensive experiments are carried out to measure the performance of this new strategy and the results indicate that it is significantly better than the naïve method of evaluating these queries one by one for both low-dimensional (2, 3, and 4) and high-dimensional (25, 50, and 104) data.

Keywords: Top-*N* Query; Multiple Queries Evaluation; Region Clustering

1. INTRODUCTION

Researches on top-*N* selection queries have intensified in recent years (since late 1990s). Finding efficient strategies to evaluate top-*N* queries has been one of the primary focuses of top-*N* query research and has received much attention [3, 5, 6, 7, 11, 12, 13, 25, 37, 39, 41, 43]. Most current work on top-*N* query evaluation considers one query at a time. But there are many applications where multiple top-*N* queries are available for processing. For example, a popular website may receive multiple top-*N* queries (say finding the best used cars based on various conditions) at about the same time. As another example, a headhunting company (website) may have many job

[*] Corresponding author.
E-mail addresses: zhu@mail.hbu.edu.cn (L. Zhu), meng@cs.binghamton.edu (W. Meng), wenzhuyang@mail.hbu.edu.cn (W. Yang), ai@bjut.edu.cn (C. Liu)

openings and may receive many resumes for these openings, and the company wants to find the top-*N* matches (the best matching resumes) for every job opening [26]. In these and other similar situations, one can either employ an existing technique to evaluate the top-*N* queries one by one or use an algorithm to evaluate these queries collectively. In this paper, we propose a new method to evaluate multiple top-*N* queries collectively. We show that this method is significantly more efficient than the one-query-at-a-time method.

Example 1. Consider a database of used cars. Let the schema of the used cars be: Usedcars(id#, make, model, year, price, mileage). Many prospective used car buyers may submit their queries with conditions on *year*, *price*, and *mileage* concurrently, and want to find the top 20 matches to each of these queries. Naturally, how to respond to these prospective buyers as quickly as possible is a problem that needs a solution. □

A top-*N* query may be evaluated in three steps. (1) A search range is determined based on some technique [5, 11, 43] for each attribute in the query condition. If *n* attributes are involved, then these ranges form an *n*-dimensional hyper rectangle (or ***region***). (2) The tuples (data points) in the search region are retrieved. (3) The retrieved tuples are ranked based on their distances with the query and the top *N* tuples are displayed.

When evaluating multiple top-*N* queries, one way is to evaluate them one by one independently, we call this method the *Naïve Method* (**NM**) in this paper. Another method is to analyze the relationships among the search regions of these queries, group those that are close to each other (say having large overlaps) into clusters and then evaluate the queries in each cluster collectively. This is the main idea of our *Region Clustering Method* (**RCM**) to be presented in this paper. When a cluster has multiple regions, we find the *smallest containing region* (*SCR*) that contains all these regions, and retrieve all the tuples from this *SCR* into memory; then we identify the tuples for each individual region, compute their distances with respect to their queries to find the top-*N* tuples for each query. Intuitively, if many regions in a cluster are close to each other and have overlaps, **RCM** may be more efficient for the following reasons. First, the number of random I/Os can be reduced. Retrieving tuples from each region incurs at least one random I/O and when there are many regions, many random I/Os will occur. In contrast, when a (larger) continuous space (the *SCR*) is searched, most I/Os are likely to be sequential I/Os. As random I/Os are much more expensive than sequential I/Os, **RCM** is likely to yield lower I/O cost. Second, the possibility of retrieving the same tuple multiple times is avoided. If a tuple appears in *k* different regions, then the tuple will be retrieved *k* times using **NM**, but only once by **RCM** if the *k* regions are clustered together. On the other hand, **RCM** may retrieve more tuples that are not needed by any query as the *SCR* may cover areas that are not covered by any of the regions.

Therefore, for **RCM** to be effective, the clustering must be carried out in such a way that maximizes the benefits while minimizing the loss. The main focus of this paper is to develop effective region clustering strategies. We assume that the search region for each top-*N* query has been obtained using an existing method [5, 11, 43].

In the literature, the phrase "multiple top-*N* queries" has been used in different contexts with different meanings. In [17], it means to find the top-*N* results that are closest to a set of queries based on a collective distance measure. This type of queries is also known as group nearest neighbour queries [31] or multipoint queries [8]. In [29], multiple scoring functions are defined for the same query and "multiple top-*N* queries" means to find a set of top-*N* results with respect to each scoring function for the same query at the same time. In this paper, we consider the simultaneous evaluation of multiple independent top-*N* queries. Note that for the most part our approach is independent of the scoring function(s) used because the input to our approach is a set of search regions of individual top-*N* queries; it does not matter to us if these regions are computed based on different scoring functions. Thus, the kind of multiple top-*N* queries we consider in this paper are different from those considered in previous work. More details about the differences will be provided in Section 2.

The main contribution of this paper is the development and evaluation of a region clustering based method for evaluating multiple top-*N* queries collectively. Note that even though this method is presented in the context of evaluating multiple top-*N* queries, it is directly applicable to evaluating multiple range queries in multi-dimensional spaces. This method can be used in arbitrary dimensional spaces and it works well for both low-dimensional and high-dimensional data. In our experiments, datasets with 2, 3 and 4 dimensions (low dimensions) and 25, 50 and 104 (high dimensions) are used. Our experimental results indicate that this method can lead to significant cost savings over the naïve one-query-at-a-time method. To the best of our knowledge, the problem of evaluating the kind of multiple top-*N* queries that we consider in this paper has not been investigated before.

The rest of the paper is organized as follows. In Section 2, we briefly review some related works. In Section 3, we introduce some notations and provide a brief analysis on the cost of evaluating multiple top-*N* queries. In Section 4, we present our *Region Clustering Method* (**RCM**) to top-*N* query evaluation. In Section 5, we present the experimental results. Finally, in Section 6 we conclude the paper.

2. RELATED WORK

The need to rank the results of database queries has long been recognized. Motro [28] gave the definitions of vague queries. He emphasized the need to support approximate and ranked matches in a database query language, and introduced vague predicates. Carey and Kossmann [6, 7] proposed techniques to optimize top-*N* queries when the scoring is done through the SQL "Order By" clause by reducing the size of intermediate results. *Onion* [9] and *Prefer* [18, 19] are preprocessing-based techniques for top-*N* queries. For a linear preference function, the basic observation of *Onion* is that the tuples with the highest score can be found within the convex hull of the dataset. *Prefer* considers both linear [18] and non-linear [19] scoring functions, provided that a different set of views is maintained for each function type. In addition, a variety of algorithms for materialized top-*N* views have been proposed in [12, 40]. The authors of [13] use a probabilistic approach to optimize top-*N* queries. The ranking condition in [13] involves only a single attribute. [39] proposed a computational model, *ranking cube*, for efficient answering of top-*N* queries with multidimensional selections.

Fagin et al. [15] introduced the threshold algorithms (TA) that perform index scans over pre-computed index lists, one for each attribute or keyword in the query, which are sorted in descending order of per-attribute or per-keyword scores. Variations of TA have been proposed for several applications, including similarity search in multimedia repositories [10], approximate top-*N* retrieval with probabilistic guarantees [37], scheduling methods based on a Knapsack-related optimization for sequential accesses and a cost model for random accesses [3], and the distributed TA-style algorithm has been presented in [25, 27].

The query model for top-*N* queries in [5, 11, 43] is consistent with the definitions in [28]. In [5, 11, 43], the key focus is on exploring the strategies that map a top-*N* query into a traditional range selection query. The basic idea of these strategies is to find an appropriate search region centered at the query point of any given top-*N* query such that all of the desired tuples (i.e. the top-*N* tuples) but very few undesired ones are contained in the search region.

Most existing works focus on the evaluation of top-*N* queries with only selection conditions (no joins). In this paper, we present a method to evaluate multiple such selection queries efficiently. This paper does not focus on how to find the best search region for each top-*N* query, instead the search regions of all input queries are used as input to our region clustering method. In other words, our method relies on existing techniques to find the search regions for individual top-*N* queries. Specifically, we can say that the work to be reported in this paper is built on top of the techniques described in [5, 11, 43]. Since the histogram-based approaches in [5] are not suitable for high dimensional data sets [5, 23] and the sampling-based method in [11] only provides an approximate answer to a given top-*N* query (i.e., it does not guarantee the retrieval of all the top-*N* tuples), we use the method described in [43] to provide the search region for individual queries in this paper. The learning-based method in [43] can learn from either randomly generated training queries or real user queries. Furthermore, it delivers good performance for either low-dimensional data or high-dimensional data and guarantees the retrieval of all top-*N* tuples for each query.

Yu et al. [41, 42] introduced the methods for processing top-*N* queries in multiple database environments. The techniques are based on ranking databases. [42] uses histograms and [41] considers the information of past queries in top-*N* query evaluation. The authors of [2] developed methods to optimize the communication costs in P2P networks. In this paper, our environment consists of a single database at a central location.

Top-*N* queries involving joins have also been considered [6, 17, 20, 21, 22, 24, 44]. Ilyas et al. [20] introduced a pipelined rank-join operator, based on the ripple join technique. [21] proposed an algorithm that is suitable for evaluating a hierarchy of join operators. The RankSQL work [22, 24] considered the order of binary rank joins at query-planning time. For the planning time optimization, RankSQL uses simple statistical models, assuming that scores within a list follow a *normal distribution* [22]. We consider only selection queries in this paper and join queries will be considered in the future.

Multiple-query processing was first studied in late 1980s [32, 33], and it still is an active research area [1, 14, 35, 36, 38]. But existing works do not consider multiple top-*N* queries. The authors of [16] proposed a UB-tree based method to process a set of query boxes (i.e., range queries), then used query boxes to approximate a non-rectangular shape (say, a triangle). The goal of this work was to minimize the number of loaded pages that overlap and the experiments were based on 2-dimensional data. As mentioned in [4], index-based method is efficient for databases with a small number of attributes, but not suitable for high-dimensional space (larger than 12). Our *Region Clustering Method* (**RCM**) does not use any index and is suitable for high-dimensional dataset (104 dimensions in our experiments).

The phrase “multiple top-*N* queries” has appeared in several recent papers but it does not have the same meaning as the type of multiple top-*N* queries we consider in this paper. In [17], it means to find the top-*N* results that are closest to a set of queries based on a collective distance measure. This type of queries is also known as group nearest neighbour (GNN) queries [31] or multipoint queries [8]. As an example, in [31], the problem is to find the N (≥ 1) data point(s) with the smallest *sum of distances* to *all query points* in a given query set $\mathsf{Q} = \{Q_1,\dots,Q_m\}$. Clearly, this problem is very different from the problem we are trying to solve in this paper, namely, find the top-*N* tuples for *each query* of a given set of queries. The example in Figure 1(a) can be used to illustrate the difference. In Figure 1(a), $\{P_1, P_2, P_3\}$ are three data points and $\{Q_1, Q_2\}$ are two query points in a 2-dimensional Euclidean space.

Based on the distances given in the figure, P_3 is the desired data point for the GNN problem with $N = 1$, because $d(\{Q_1, Q_2\}, P_3) = d(Q_1, P_3) + d(Q_2, P_3) = 4$ is the minimum value in $\{d(\{Q_1, Q_2\}, P_i), i=1,2,3\}$. For our problem, we need to find the top-N results for *each query*; when $N = 1$, the top-1 tuples for Q_1 and Q_2 are P_1 and P_2, respectively. In [8], the *sum of weighted distances* is used instead of the *sum of distance* in [31]. The collective distance measure used in [17] is the *minimum of distances*, i.e., $d(P, \mathbf{Q}) = \min_N \{d(P, Q_i) \mid Q_i \in \mathbf{Q}, i = 1,\ldots, m\}$ is used to define the top-N result tuple(s). In Figure 1(b), for instance, the set of multiple query points is $\mathbf{Q} = \{Q_1, Q_2, Q_3\}$ and the three tuples are P_1, P_2 and P_3. The top-1 tuple of the set $\mathbf{Q} = \{Q_1, Q_2, Q_3\}$ is P_3 according to the definitions in [17]; however, in our paper, the top-1 tuples for Q_1, Q_2 and Q_3 are P_1, P_2 and P_3 respectively.

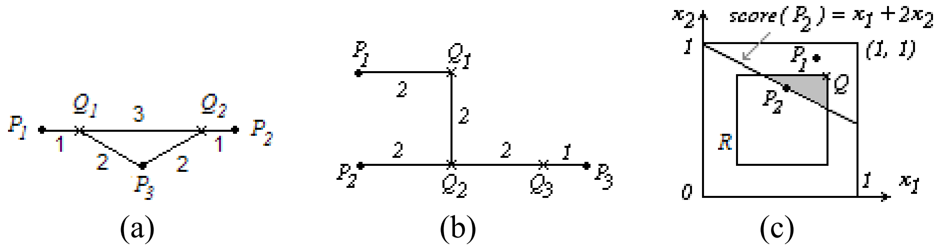


Fig.1. Illustrations of the differences of top-*N* queries

The authors of [29] studied continuous monitoring of top-N queries over a fixed-size window of the most recent data. This paper also discussed the processing of "multiple top-N queries". However, there are two major differences between the work in [29] and our work reported in this paper. First, in [29], the query point is always at the upper-right corner of the restricted region (see region R in Figure 1(c) for a 2-dimensional example) and only top-N results in a specific area (see the grey triangle in Figure 1(c)) are considered. Our work does not have these restrictions. Second, in [29], only one query point is considered at a time but multiple scoring functions are defined for the same query; the objective is to find a set of top-N results with respect to each scoring function for the same query at the same time. In contrast, our work considers multiple query points and tries to find the top-N results for each query.

3. PROBLEM DEFINITION AND ANALYSIS

Let $\mathfrak{R}^n$ be an n-dimensional metric space with distance function $d(\,)$, where $\mathfrak{R}$ is the real line. Suppose $\boldsymbol{R} \subset \mathfrak{R}^n$ is a relation (or dataset) with n attributes $(A_1, \ldots, A_n)$. Consider a query point $Q = (q_1,\ldots, q_n) \in \mathfrak{R}^n$ and an integer $N > 0$. A top-N selection query (Q, N), or *top-N query* for short, is to find a sorted set of N tuples in $\boldsymbol{R}$ that are closest to Q according to the given distance function. The results of a top-N query are called *top-N tuples*. We assume $N < |\boldsymbol{R}|$ (the size of $\boldsymbol{R}$); otherwise, we just retrieve all tuples in $\boldsymbol{R}$.

As mentioned in Section 1, a number of approaches exist for mapping a top-N query on a relational database into a traditional range selection query. Specifically, for a given top-N query (Q, N), these approaches can determine a search region, denoted $R(Q, N)$, for the top-N query.

In this paper, a region $R = \prod_{i=1}^{n} [a_i, b_i]$ is always an n-dimensional hyper rectangle, and we use $v(R) = \prod_{i=1}^{n} (b_i - a_i)$ to denote the volume of R. Let $\{Q_1,\ldots, Q_m\} \subset \mathfrak{R}^n$ be a set of m points, and $N_1,\ldots, N_m$ be m positive integers, $m > 1$. In this paper, we discuss how to efficiently evaluate the set of m top-N queries, $\mathsf{Q} = \{(Q_1, N_1), \ldots, (Q_m, N_m)\}$.

We now use an example to explain the differences in evaluation costs between **NM** and **RCM**. Figure 2 shows the search regions $R(Q_1, N_1)$ and $R(Q_2, N_2)$ of two top-N queries (Q_1, N_1) and (Q_2, N_2) in $\mathfrak{R}^2$, respectively. The smallest containing region (*SCR*) of these two regions is the rectangle with the dotted line.

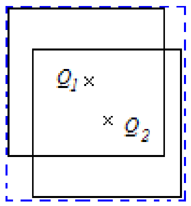


Fig.2. The *SCR* and search regions of Q_1 and Q_2

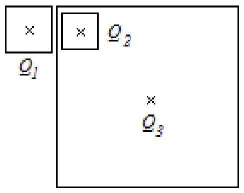


Fig.3. Clustering of three top-*N* queries

As explained in Section 1, a typical top-N query can be processed in three steps. Let *RC*( ) denote the cost of constructing the search region, *IOC*( ) the cost of evaluating the query (mostly I/O cost), and *SC*( ) the cost of sorting and displaying the results. Using **NM**, the total cost for evaluating (Q_1, N_1) and (Q_2, N_2) is as follows:

$$\text{Cost\_NM} = RC(Q_1, N_1) + IOC(Q_1, N_1) + SC(Q_1, N_1) + RC(Q_2, N_2) + IOC(Q_2, N_2) + SC(Q_2, N_2) \quad (1)$$

When **RCM** is used, we still need to construct the search regions for the two queries as in **NM**. Then we construct the *SCR* and retrieve all tuples from *SCR* into the memory. Next, we identify the retrieved tuples that belong to $R(Q_1, N_1)$ and $R(Q_2, N_2)$, respectively. Finally, sort the tuples in $R(Q_1, N_1)$ and $R(Q_2, N_2)$, and output the top-N tuples for Q_1 and Q_2, respectively. Thus, using **RCM**, the total cost will be:

$$\text{Cost\_RCM} = RC(Q_1, N_1) + RC(Q_2, N_2) + IOC(SCR) + SC(Q_1, N_1) + SC(Q_2, N_2) + \delta \quad (2)$$

where δ denotes all the extra cost (including the cost for constructing the *SCR*, performing the clustering and identifying the retrieved tuples that belong to each individual search region). In Expression (1), there are two independent I/O requests and in Expression (2), there is only one I/O request. Suppose each I/O request incurs one random I/O plus some sequential I/Os. Since the cost of a random I/O is about 10 times higher than that of a sequential I/O [30], query evaluation strategies that incur lower random I/Os are likely to be more efficient. As I/O cost is usually the main cost in most database applications, reducing the I/O cost, especially the random I/O cost, is an effective way to improve the efficiency of query evaluation.

The **RCM** we propose in this paper aims to identify situations where the total cost for **RCM** is smaller than the total cost for **NM**. The reason we employ region clustering rather than (query) point clustering is because top-N queries are better characterized by their search regions instead of just the query points. Consider the example in Figure 3. Since $R(Q_3, N_3) \supset R(Q_2, N_2)$, Q_3 and Q_2 should be in the same cluster although the distance $d(Q_1, Q_2)$ is less than $d(Q_3 , Q_2)$. In this case, if we have retrieved all the tuples in $R(Q_3, N_3)$ from a database, we can obtain the top-N tuples for Q_2 from those in $R(Q_3, N_3)$ without additional I/Os. If we place Q_1 and Q_2 in the same cluster, the above savings on I/Os cannot be obtained.

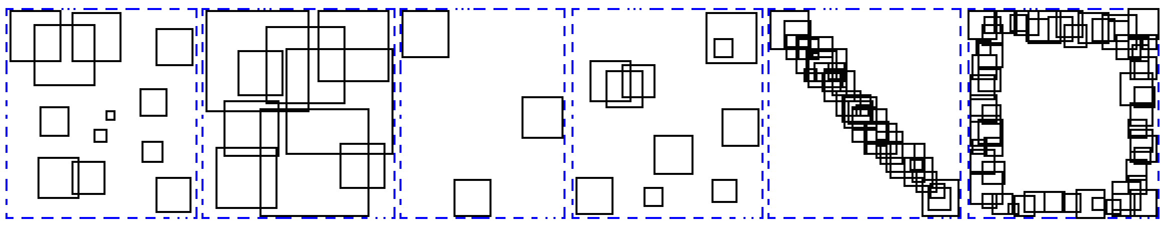


Fig. 4. Examples of the distributions of regions

Clearly, the distribution of the search regions of the input queries plays an important role in how to evaluate these queries collectively. We now provide some measures to characterize the distribution of a set of regions. Figure 4 shows several examples of the distributions of some 2-dimensional regions. Let R_1, R_2, …, R_k be k regions. By comparing the three volumes $\sum_{i=1}^{k} v(R_i)$, $v(\cup_{i=1}^{k} R_i)$ and $v(SCR(\cup_{i=1}^{k} R_i))$, we can gain useful insight about the distribution of these regions. Denote

$$\alpha = v\left(\bigcup_{i=1}^{k} R_i\right) \Big/ \sum_{i=1}^{k} v(R_i)\,,\ \beta = v\left(\bigcup_{i}^{k} R_i\right) \Big/ v\left(SCR(\bigcup_{i=1}^{k} R_i)\right),\ \tau = v\left(SCR(\bigcup_{i=1}^{k} R_i)\right) \Big/ \sum_{i=1}^{k} v(R_i)\,.$$

It is easy to see that $\alpha \leq 1$ and $\beta \leq 1$. The values of α indicate the degree of overlap of the regions and a smaller α indicates more overlap. The values of β reflect the closeness between the regions and the size of the lacuna in the SCR; a larger β indicates closer regions and smaller lacuna in general. The value of τ may be greater than, less than or equal to 1, and it indicates how much of the SCR is occupied by the R_i's; a smaller τ denotes larger volumes of R_i's in the SCR. If $\tau < 1$, then there must be some R_i's that overlap one another.

In some cases, using the *SCR* of $\cup_{i=1}^{k} R_i$ as a single search region for all R_1, R_2, …, R_k may be an effective strategy like in the case shown in Figure 4(b), which has "good measures", that is, a small α, a large β and a small τ. However, in other cases, it may not be appropriate to form just one cluster as it may lead to the retrieval of many useless tuples, like in the cases shown in Figure 4(e)-(f), which have "good α and τ" (small α and τ), but a "bad β" (small β). Intuitively, it seems to be more appropriate to form several smaller clusters for the regions along the diagonal in Figure 4(e) and 4 clusters for the regions in Figure 4(f) along the 4 edges. Note that if clusters were formed simply based on overlapping regions, then all the regions in Figure 4(e) and (f) would form a single cluster, and such *SCRs* would contain too many useless tuples. Figure 4 (a), (c) and (d) represent more random cases and they all have "bad measures", that is, a large α, a small β and a large τ. Figure 4(c) is the worst case with $\alpha = 1$. Therefore, the three regions should not be grouped into a single cluster; in other words, they should be evaluated individually, i.e., each region is considered as a separate cluster. Figure 4(a) and (d) are similar in that in each case some regions overlap and some don't, and $\alpha < 1$. For the regions in Figure 4(d), it is intuitively reasonable for the overlapping regions to form two clusters and the remaining regions to remain un-clustered. However, it is less clear how the regions in Figure 4 (a) should be clustered. A general algorithm for region clustering will be presented in Section 4.3.

4. MULTIPLE TOP-*N* QUERY PROCESSING

The key step of our approach is to group the search regions of *m* input top-*N* queries into clusters. All current clustering techniques employ some similarity measure to compute the distance among the objects to be clustered. In most cases, the objects are points. However, the objects are regions in this paper.

4.1 Algorithms and Terminologies

In this section, we introduce the basic algorithms and terminologies that will be used by **RCM**. The algorithms include: (1) **VUR**, that computes the volume of the union of multiple regions; (2) **DTR**, that calculates the difference of two regions; (3) **PPS**, that partitions multiple points in *n*-dimensional space; (4) **CST**, that gets the set of candidate top-*N* tuples of a top-*N* query; and (5) **TTC**, that gets the top-*N* tuples from the set of candidate top-*N* tuples.

4.1.1 The Volume of the Union of *m* Regions (VUR)

Given m regions $R_1, R_2, \cdots, R_m$, $v(R_i)$ denotes the volume of R_i. Let $v_1 = \sum_{1\le i\le m} v(R_i)$, $v_2 = \sum_{1\le i<j\le m} v(R_i\cap R_j)$, $v_3 = \sum_{1\le i<j<k\le m} v(R_i\cap R_j\cap R_k)$, …, and $v_m = v(R_1\cap R_2\cap \cdots \cap R_m)$, then by induction, we have

$$v(\cup_{i=1}^{m} R_i) = v_1 - v_2 + v_3 - v_4 + \ldots + (-1)^{m-1} v_m\,. \qquad (3)$$

Suppose $R_1 = \prod_{i=1}^{n} [a_i, b_i]$ and $R_2 = \prod_{i=1}^{n} [c_i, d_i]$ are two n-dimensional regions. Their intersection $R_1\cap R_2$ is also an n-dimensional region. Let $p_i = \max\{a_i, c_i\}$, and $q_i = \min\{b_i, d_i\}$, $i = 1, 2, \ldots, n$. If there is an i_0 such that $p_{i_0} > q_{i_0}$, then $R_1\cap R_2 = \phi$, and $v(R_1\cap R_2) = 0$; otherwise, if $p_i \le q_i$ $(i=1,2, \ldots,n)$, then $R_1\cap R_2 \ne \phi$, $R_1\cap R_2 = \prod_{i=1}^{n} [p_i, q_i]$, and

$$v(R_1\cap R_2) = \prod_{i=1}^{n} (q_i - p_i). \qquad (4)$$

Note that $v(R_1\cap R_2) = \prod_{i=1}^{n} (q_i - p_i) = 0$ if there is an i_0 such that $p_{i_0} = q_{i_0}$, although $R_1\cap R_2 \ne \phi$.

Based on the associativity of "$\cap$", Formulas (3) and (4), the exact volume of $\cup_{i=1}^{m} R_i$ can be computed. However, Formula (3) has a complexity of $O(2^m)$. In this paper, we deal with this problem by first partitioning the set of regions into smaller sets and then process each subset separately. Note that since $R \cap \phi = \phi$, many items in Formula (3) are empty. As a result, the number of non-empty items in Formula (3) is usually much smaller than 2^m. For example, if $R_i\cap R_j = \phi$ or $v(R_i\cap R_j) = 0$, $1\le i<j<m$, then $R_i\cap R_j\cap R_k = \phi$ or $v(R_i\cap R_j\cap R_k) = 0$, for all $k, j<k\le m$, and all such items can be discarded from Formula (3).

4.1.2 The Difference of Two Regions (DTR)

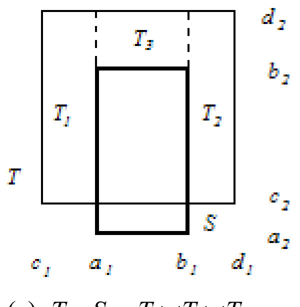
Let $S = \prod_{i=1}^{n} [a_i, b_i]$ and $T = \prod_{i=1}^{n} [c_i, d_i]$ be two regions. If $S\cap T \ne \phi$ and T is not contained in S, then the difference $T - S$ is the union of some n-dimensional sub-regions of T, that is, $T - S = \cup_{j=1}^{p} T_j$, where $T_j \subset T$, $j = 1, \ldots, p$, $v(T_i\cap T_j) = 0$, $i \ne j$, can be constructed by the following algorithm:

```
Algorithm DTR(T, S)                          /* T – S = T[1] ∪…∪ T[p] */
Input: $T=\prod_{i=1}^{n}[c_i, d_i]$, $S=\prod_{i=1}^{n}[a_i, b_i]$;   /* $S \cap T \neq \phi$ are two hyper-rectangles */
        int n;                               /* the number of dimensions */
Output: T[p]                                 /* the array of hyper-rectangles */
        int p                                /* the size of the array T[p] */
Local variables:  int i;  double t;  Hyper-rectangle H;
p:= 1; H:= T;                                /* then $H == \prod_{i=1}^{n}[c_i, d_i]$ */
for (i = 1; $i \le n$; i++)
   if $H.c_i < S.a_i$ then
      {t := $H.d_i$;  $H.d_i$ := $S.a_i$;  T[p]:= H;  $H.c_i$:= $S.a_i$;  $H.d_i$:=t;  p++;}
   end if
   if $S.b_i < H.d_i$ then
      {t :=$H.c_i$;  $H.c_i$ := $S.b_i$;  T[p]:= H;  $H.d_i$:= $S.b_i$;  $H.c_i$:=t;  p++;}
   end if
end for
p--;
```

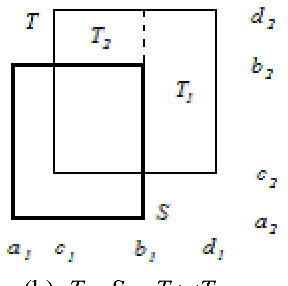
It is clear that Algorithm **DTR** also holds if T is contained in S and $T - S = \phi$. Figure 5 shows three cases of $T - S$ in 2-dimensional space.

We can use Algorithm **DTR** to reduce the query evaluation time in many cases. For example, if the tuples in S in Figure 5(a) have been retrieved, it is sufficient to retrieve the tuples in $T_1 \cup T_2 \cup T_3$ and combine them with the tuples from S that are in T to obtain the tuples in T.

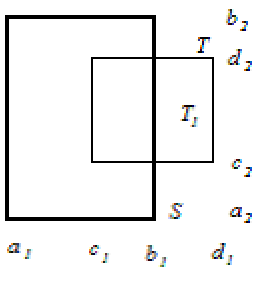
In general, the number of such T_j's (i.e., $|\{T_j\}|$) is not larger than $2n$ in an n-dimensional space. In many cases, $|\{T_j\}|$ can be much smaller than $2n$. For example, in Figures 5(b) and 5(c), $|\{T_j\}|$ is 2 and 1, respectively. Intuitively, **DTR** is effective only when $|\{T_j\}|$ is small because a large $|\{T_j\}|$ can lead to more expensive random I/Os. In our experiments, **DTR** is used only when $|\{T_j\}|$ is not larger than 2.



(a). $T - S = T_1 \cup T_2 \cup T_3$



(b). $T - S = T_1 \cup T_2$



(c). $T - S = T_1$

Fig. 5. Three cases of *T* – *S* in 2-Dimensional Space

4.1.3 The Partition of Points in *n*-dimensional Space (PPS)

Let T = $\{t^{(1)}, ..., t^{(m)}\}$ be a set of m points in $\Re^n$ and $R = \prod_{i=1}^{n}[a_i, b_i]$ be a region that contains these points.

Given a threshold m_0, there is a number p such that R is divided into p cells $\{P_i: i=1,..., p\}$ and each cell P_i contains at most m_0 points in T. The following algorithm can achieve this:

```
Algorithm PPS(T, $R$, $m_0$, $n_1$, …, $n_k$)  /*  $n_i > 1$, $i = 1, …, k$  */
(1) Find the $k$ longest edges of $R$, denote them as $e_1, e_2, ..., e_k$, and divide them into $n_1, n_2, …, n_k$ equal-length segments respectively; then $R$ is partitioned into $h = n_1 n_2 … n_k$ cells $\{P_i: i=1,…, h\}$.
(2) For each cell $P_i$ containing more than $m_0$ points in T, let $R := P_i$ and goto (1).
```

It is clear that $p = h + (N_{ite}-1) * (h-1) = h * N_{ite} - N_{ite} + 1$, where N_{ite} is the number of times step (1) is executed. It is easy to know the relationship between the depth d of the partitions in **PPS** and the volume of the cells. Let $v_{(d)}$ denote the volume of a cell after d consecutive partitions from R. Then $v_{(d)} = v(R)/h^d$. The convergence of the algorithm **PPS** is very fast, in fact, it is not slower than $O(1/2^d)$, which is the slowest case with $k = 1$ and $n_1 = 2$.

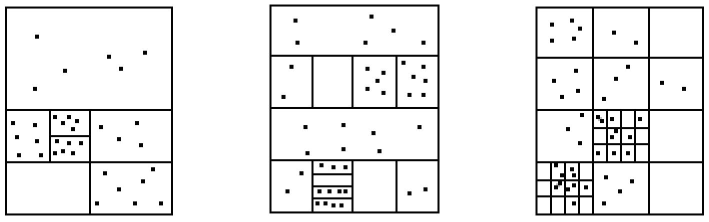

(a) $k = 1$, $n_1 = 2$ (b) $k = 1$, $n_1 = 4$ (c) $k = 2$, $n_1 = 4$, $n_2 = 3$

Fig. 6. PPS in 2-Dimensional Space, $m_0 = 8$

Let $m_0 = 8$. Figures 6(a)-(c) illustrate three cases of the algorithm **PPS** in 2-dimensional space. Note that, in Figure 6(c), R is divided into 3 segments along the x-axis and 4 segments along the y-axis; moreover, the two smaller cells have 4 segments along the x-axis and 3 segments along the y-axis.

In our experiments, threshold $m_0 = 8$ is used for all datasets of both low and high dimensions. The reason we choose $m_0 = 8$ is as follows. With $m_0 = 8$, each cell will have no more than 8 points. Each point corresponds to a top-N query, and as a result, it has a corresponding search region. This means that by considering the points in different cells separately, no more than 8 regions will be considered at a time. Note that the **VUR** algorithm has a complexity of $O(2^m)$, where m is the number of regions. When no more than 8 regions are considered at a time, the **VUR** algorithm can run very fast. For the 2 dimensional case, we take $k = 2$, $n_1 = 4$, and $n_2 = 3$ as in Figure 6(c). For the n dimensional case ($n \geq 3$), we take $k = 3$, $n_1 = 3$, $n_2 = 2$ and $n_3 = 2$. Thus, each time, the *SCR* (or a cell) is partitioned into 12 smaller cells. For example, for the *104*-dimensional dataset Lsi104D, we will find the 3 longest edges e_1, e_2 and e_3 of *SCR* from all *104* edges in each partition. In the first partition, suppose e_1, e_2 and e_3 are the 5th, the 17th, and the 89th dimensions, respectively, and by dividing them into *3*, *2*, and *2* equal-length segments respectively, the *SCR* is divided into $h = 3*2*2 = 12$ cells $\{P_i: i=1, \ldots, 12\}$. In the next partition, for a cell, say P_3, e_1, e_2 and e_3 may belong to other dimensions.

4.1.4 Get Candidate Set of Top-*N* Tuples (CST)

Let (Q, N) be a top-N query, and R be its search region. If a tuple $t \in R$, it is said to be a ***candidate*** of the top-N tuples of (Q, N) (or *candidate* for short). The set of all candidates is called the ***candidate set*** of (Q, N).

Given two regions $S = \prod_{i=1}^{n} [a_i, b_i]$ and $T = \prod_{i=1}^{n} [c_i, d_i]$, if the candidate set of S has been obtained, then we can get the candidate set of T using the following algorithm:

```
Algorithm CST(T, S)
(1) if $T \subset S$, then the candidate set of $T$ is already obtained; simply identify the tuples from $S$ that are in $T$: $\{t : t = (t_1, \ldots, t_n) \in S$ and $c_i \le t_i \le d_i \}$;
(2) if $T \not\subset S$ and $T \cap S \neq \phi$, then obtain $T - S = \cup_{j=1}^{p} T_j$ using DTR(T, S); if $p$ is less than a threshold, get tuples $\{t^{(i)}\}$ from $\cup_{j=1}^{p} T_j$ and the candidate set of $T$ is $\{t^{(i)}\} \cup \{t : t = (t_1, \ldots, t_n) \in S$ and $c_i \le t_i \le d_i\}$;
(3) if $T \cap S = \phi$ or $p$ is not less than a threshold, get tuples in $T$ from the database directly;
```

Now we introduce the concept of ***best-super-region*** (**BSR**). Suppose that $\{R_i: i=1, \ldots, m\}$ is a set of *regions*, R_k is called the BSR of R_i if $R_i \subset R_k$ and $v(R_k) = \min\{v(R_j) : R_i \subset R_j, 1 \le j \le m \text{ and } i \neq j\}$, i.e., R_k is the smallest region that contains R_i. The index k of R_k is called the best-super-region number (BSR number) of R_i, denoted as $R_i.bsn$. In Figure 7 (where $R_3 = R_7$), $R_5.bsn = 3$; $R_8.bsn = 3$; $R_3.bsn = 7$; $R_7.bsn = 10$; $R_2.bsn = 10$. If the regions in Figure 7 are the search regions of six top-N queries, they can be considered as a cluster. It is sufficient to retrieve the tuples from R_{10} using only one I/O request (one I/O request may incur multiple likely sequential I/Os), and then we can obtain the candidate sets from the BSRs for others. Note that $R_3 = R_7$, $R_3.bsn = 7$, but $R_7.bsn = 10$. If $R_3.bsn = 7$ and $R_7.bsn = 3$, there would be an infinite loop.

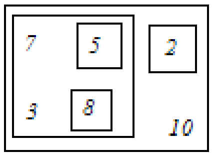


Fig.7. The best-super-region and best-super-region number

In addition, if we have $R_1 \subset R_2 \subset \ldots \subset R_k$, i.e., each R_i is the best-super-region of R_{i-1}, then we can get the candidate sets for all regions from R_k.

4.1.5 Get Top-*N* Tuples from Candidate Set (TTC)

Algorithm **TTC** first computes the distance between a top-*N* query (Q, N) and every tuple in the candidate set of the query, and then identifies the *N* tuples with the smallest distances. A *heap* structure can be used to keep track of the *N* tuples with the smallest distances during the computation. Since this algorithm is straightforward, its detail will not be presented here.

If the candidate set has less than *N* tuples, the search region of the query needs to be enlarged, which will be discussed in Section 4.4.2.

4.2 Models of Clustering of Multiple Top-*N* Queries

In this section, we present three models of region clustering and discuss the clustering conditions for each model. Let $\mathcal{T} = \{T_i : i=1,\dots, m\}$ be a set of m regions and $\mathcal{C} = \{H_k : k = 0, 1,\dots, L\}$ be a subset of $\mathcal{T}$. If $\mathcal{C}$ satisfies some conditions, it is called a cluster. Suppose *SCR* is the smallest containing region of $\mathcal{C}$.

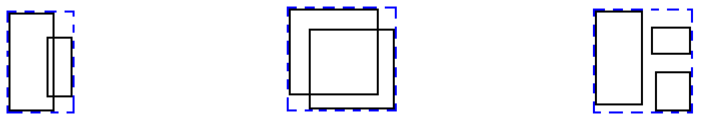

(a) I-clustering (b) C-clustering (c) S-clustering

Fig.8. Models of region clustering

4.2.1. I-Clustering

$\mathcal{C}$ is called an ***I-cluster*** (for **I**ntersection based cluster) if it satisfies the following conditions:

(1) for each $0 < j \le L$, there is $0 \le k < j$ such that $H_k \cap H_j \neq \phi$;

(2) $v(\cup_{k=0}^{L} H_k) / v(SCR(\{H_k\})) \ge c_1$, where c_1 is a constant.

The first condition means that each region in $\mathcal{C}$ intersects with at least another region in $\mathcal{C}$. The second condition is to limit the lacuna of the *SCR*, making the regions in a cluster reasonably close to each other. Figure 8(a) shows an example of I-clustering.

The conditions of I-clustering are called ***I-cluster conditions***, and c_1 is called ***I-cluster threshold***. Since $0 < v(\cup_{k=0}^{L} H_k) / v(SCR(\{H_k\})) \le 1$, $c_1 \in [0, 1]$. It can be seen that in general a larger c_1 requires closer regions, which leads to more compact clusters (i.e., the lacuna of the *SCR* is smaller). On the other hand, a larger c_1 reduces the applicability of I-clustering as fewer regions can satisfy the I-cluster conditions. In this paper, we determine an approximate optimal value of c_1 through training (see Section 5.2). Note that volume $v(\cup_{k=0}^{L} H_k)$ can be obtained using algorithm **VUR**.

4.2.2. C-Clustering

ς is called a ***C-cluster*** (for **C**enter based cluster) if it satisfies the following conditions:

(1) for each $0 < j \leq L$, there is $0 \leq k < j$ such that the center of H_j is in H_k;

(2) $v(\cup_{k=0}^{L} H_k) / v(SCR(\{H_k\})) \geq c_2$, where c_2 is a constant.

It is clear that C-cluster is a special case of I-cluster and it requires the regions to have higher degree of overlap because of the condition that the center of one region is contained in another region. Figure 8(b) shows an example of C-clustering. The two conditions above are called ***C-cluster conditions***, and c_2 is called ***C-cluster threshold***. The impact of c_2 on C-clustering is the same as that of c_1 on I-clustering. We also use training to determine an approximate optimal value for c_2.

4.2.3. S-Clustering

The set $\varsigma = \{H_k : k = 0, 1, \ldots, L\}$ is called an ***S-cluster*** (for **S**um based cluster) if it satisfies the condition:

$v(SCR(\{H_k\})) / \sum_{k=0}^{L} v(H_k) < c_3$, where c_3 is a constant.

The condition is called ***S-cluster condition***, and c_3 is called ***S-cluster threshold***. Figure 8(c) shows an example of S-clustering and the three regions form an S-cluster. The value of c_3 reflects the degree of closeness of the *region*s. There is no clear upper bound for c_3. When $c_3 < 1$, according to the discussion about τ in Section 3, some of these regions must overlap and they are already considered by either I-clustering or C-clustering. On the other hand, if $c_3 > 3$, then these regions are likely to be too far apart to form effective clusters. Therefore, it is reasonable to consider $c_3 \in [1, 3]$. Training will again be used to determine an approximate optimal value for c_3.

4.2.4. Comparison of the three clustering models

Based on the above discussions about the three clustering models defined above, the relationships among the three models can be summarized as follows:

1. C-clustering tends to produce the most compact clusters (i.e., the lacuna of the *SCR* tends to be the smallest), followed by I-clustering and then by S-clustering.

2. S-clustering tends to be most applicable, followed by I-clustering and then by C-clustering.

4.3 Cluster Search Regions

Let $\mathbf{Q} = \{(Q_1, N_1), \ldots, (Q_m, N_m)\}$ be a set of m top-N queries and $R_1, \ldots, R_m$ be their respective search regions. In this section we discuss how to cluster the search regions in $\mathbf{R} = \{R_k : k = 1,\ldots,m\}$ based on the basic algorithms and clustering models described in Sections 4.1 and 4.2. For $R_k = \prod_{i=1}^{n} [a_i, b_i]$, we sometimes denote $R_k.a_i = a_i$ and $R_k.b_i = b_i$. Also, we sometimes do not distinguish the top-N query (Q_k, N_k) and its search region R_k in the following discussion. Suppose $\min(A_i)$ and $\max(A_i)$ are the minimum and maximum values of attribute A_i of all tuples in a relation. Without loss of generality, for the set $\mathbf{R}$, if $R_k.a_i < \min(A_i)$, let $R_k.a_i := \min(A_i)$, and if $R_k.b_i > \max(A_i)$, let $R_k.b_i := \max(A_i)$.

Based on the relationships among the three clustering models (see Section 4.2.4), it seems natural to employ the following region clustering strategy. First, C-clustering is employed to the input search regions; next I-clustering is applied to those regions that are not clustered by C-clustering; and finally, S-clustering is used to cluster the remaining regions as much as possible. However, our experiments indicate that for some datasets, C-clustering is not effective because its relatively low applicability and after C-clustering is applied, it becomes more difficult to apply I-clustering to the remaining regions. Further analysis indicates that when the sizes of the search regions of the input top-N queries are large, it is more likely that the regions can satisfy the center-in condition (i.e., the first condition) of the C-cluster conditions; as a result, C-clustering is more effective in such situations. On the other hand, when the search regions are small, C-clustering is not as effective as I-clustering.

Based on the above observation, we adopt the following general clustering strategy in this paper.

1. Apply either I-clustering or C-clustering to cluster the input regions based on the sizes of their search regions. Specifically, C-clustering is applied if the regions are large and I-clustering is applied if the regions are small.
2. Apply S-clustering to cluster the remaining regions.

For convenience, the I-clustering and C-clustering models will be called *primary clustering models* while the S-clustering model will be called a *secondary clustering model*. The size of a region can be estimated based on the distribution of data in a dataset, the query point and the value of N in a top-N query.

More detailed description of our clustering algorithm is given below. Again $\mathbf{R} = \{R_k : k = 1,...,m\}$ is the set of search regions under consideration.

1. Label all the regions in $\mathbf{R}$ that have a super-region and remove them temporarily. For each k, if R_k has a super-region, determine its best-super-region (BSR) (see Section 4.1.4) and save the BSR number, $R_k.bsn$. Since we always place a region and its super regions in the same cluster, it is sufficient to consider how to cluster those that have no super regions.

   Let $\boldsymbol{T} = \{T_i\}$ denote the subset of the regions in $\mathbf{R} = \{R_k\}$ that have no super-regions. Moreover, let $m_T = |\boldsymbol{T}|$. Obviously, $m_T \leq |\mathbf{R}|$.

2. Obtain the Smallest Containing Region (*SCR*) of all $\{T_j\}$. Denote $SCR.a_i = \min\limits_{1 \leq j \leq m_T} \{T_j.a_i\}$ and $SCR.b_i = \max\limits_{1 \leq j \leq m_T} \{T_j.b_i\}$, where $i = 1,\ldots, n$ and n is the number of dimensions.

3. Partition the centers of the search regions in $\boldsymbol{T}$ into K subsets, $1 < K < m_T$, using Algorithm **PPS** (see Section 4.1.3). This effectively partitions the *SCR* into K sub-regions, $S_1,...,S_K$, $1 < K < m_T$, and there are at most m_0 centers in each sub-region. The search regions in $\boldsymbol{T}$ whose centers are in the same sub-region are likely to be grouped into the same cluster.

4. Cluster the regions in $\boldsymbol{T}$ using the clustering models described in Section 4.2. We first apply a primary clustering model (C-clustering or I-clustering) and then apply the secondary clustering model (S-clustering).

   We need a threshold L_0 as the maximum number of regions in a cluster $\boldsymbol{\varsigma}$. Because the **VUR** algorithm has a complexity of $O(2^m)$, where m is the number of regions, and we use **VUR** to calculate the volume of the unions of the search regions in $\boldsymbol{\varsigma}$ (see Section 4.2), L_0 should not be too large. We use $L_0 = 10$ in our experiments, which allows **VUR** to run fast.

   First, for each sub-region S_i, $i = 1,..., K$, find all not-yet clustered T_k's in $\boldsymbol{T}$ whose centers are in S_i. Let $\boldsymbol{T}_i = \{T_k$: the center of T_k is in S_i, $k = 1,..., L$, and $1 \leq L \leq m_0\}$

   Second, arbitrarily select a region from $\boldsymbol{T}_i$, denote as H_0, and use it as the seed of a new cluster. Then find the next region H_1 from $\boldsymbol{T} - \{H_0\}$ that satisfies the *clustering conditions* of the clustering model being used. Repeat this process to find H_h from $\boldsymbol{T} - \{H_0, H_1,..., H_{h-1}\}$ until no remaining regions satisfy the *clustering conditions* of the clustering model or $h = L_0$ is reached. Thus we get a cluster $\varsigma = \{H_j : j = 0, 1, ..., h-1, h \leq L_0\}$. If $\boldsymbol{T_i}$ is not empty, select another un-clustered region from $\boldsymbol{T_i}$ as the new seed and get another cluster by repeating the above process.

Let $\{\varsigma_i, i = 1, ..., M\}$ denote the clusters constructed above and build the smallest containing region $SCR_i = SCR(\varsigma_i)$ for each cluster ς_i, $i = 1, ..., M$. It is possible that some of the clusters may contain a single region, for example, when the selected seed cannot be clustered with the remaining regions.

5. Check if some of the un-clustered regions (i.e., clusters with a single region) can be added into an existing cluster. If there is such a region $T_j \subset SCR_h$, add T_j into ς_h (note that due to the constraint of the threshold L_0, it is possible that T_j could not be added to a ς_i in step 4). If there are multiple SCR_i's that contain such a T_j, place it in the cluster whose SCR_i has the smallest volume. Therefore, it is possible that the size of the final cluster is more than L_0. The remaining un-clustered regions will be processed individually.
6. For each smallest containing region $SCR_i = SCR(\varsigma_i)$, find its best-super-region (BSR) SCR_k and save its BSR number $SCR_i.bsn = k$ (see Section 4.1.4).

Example 2. In Figure 9, $\{R_i : i=1, ..., 25\}$ is the set of search regions in 2-dimensinal space. Suppose these search regions are small. By applying I-clustering as the primary clustering model first, three clusters $\varsigma_1 = \{R_1, R_2\}$, $\varsigma_2 = \{R_3, R_4, R_5\}$ and $\varsigma_3 = \{R_{11}, R_{12}, R_{13}, R_{14}, R_{15}, R_{16}, R_{17}, R_{18}, R_{19}, R_{20}\}$ are formed. By applying S-clustering to the remaining regions, $\varsigma_4 = \{R_6, R_7, R_8\}$ is formed. The smallest containing regions of ς_1, ς_2, ς_3 and ς_4 are SCR_1, SCR_2, SCR_3 and SCR_4, respectively. R_9 is added into ς_1 since it is in the same cluster with its super-region R_1. By step 5, R_{10} is added to ς_2 although it does not intersect with any of other regions in ς_2. Due to the constraint of L_0 (L_0=10), R_{24} was not added into ς_3 initially but added into ς_3 later because it is contained in SCR_3. Four regions R_{21}, R_{22}, R_{23} and R_{25} are single-region clusters; thus their corresponding queries will be evaluated independently. Although $S_{21} \cap S_{22} \neq \phi$, S_{21} and S_{22} do not form a cluster because they do not satisfy the second condition of I-clustering (see Section 4.2.2). Thus, for this example, the total number of I/O requests to the database is 8, one for each of SCR_1, SCR_2, SCR_3, SCR_4, R_{21}, R_{22}, R_{23} and R_{25}. □

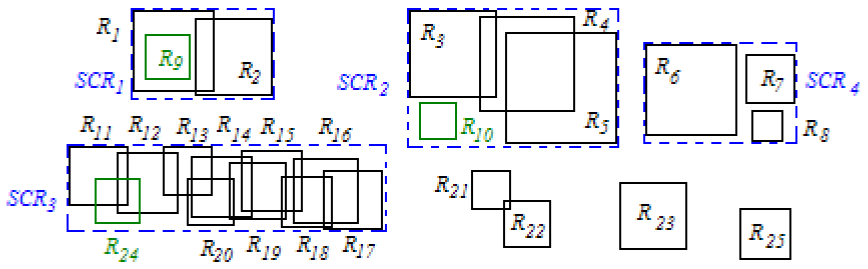


Fig.9. An example of clusters for {R_i : i=1,…,25}

4.4 Top-*N* Tuple Retrieval

4.4.1 Search top-*N* tuples

For obtaining the sets of top-*N* tuples for *m* queries, we consider two cases, one is for the clustered queries, and the other is for individual queries.

Case 1: clustered queries. There are three steps.

Step 1. From $\{SCR_i\}$, find the first *SCR* that has no super region, denote it as $SCR_{i_0} = \prod_{i=1}^{n} [a_i, b_i]$. Retrieve all tuples $\{t^{(i)} : i=1, \ldots, M\}$ from it using a simple selection range query:

SELECT * FROM R WHERE $(a_1 \le A_1 \le b_1)$ AND … AND $(a_n \le A_n \le b_n)$

For each search region R in ς_{i_0} (therefore $R \subset SCR_{i_0}$),

(a) if R has no super region, search the candidate set of R from $\{t^{(i)} : i=1, \ldots, M\}$, and get the top-*N* tuples using algorithm **TTC**.

(b) else (i.e. R has a super region), find out its best-super-region BSR, search the candidate set from the BSR and get the top-*N* tuples by using algorithm **TTC**. Note that getting the best-super-region may be a recursive process if BSR has its own super region.

Step 2. For each SCR_i, $i > i_0$, that has no super region,

(a) if there is no SCR_j $(i_0 \le j < i)$ that intersects with SCR_i, follow the same procedure as Step 1.

(b) else, among all SCR_j's $(i_0 \le j < i)$ that intersect with SCR_i, find the one, denoted SCR_{j_0}, that has the largest intersection. Then calculate the difference of the two regions using algorithm **DTR**: $SCR_i - SCR_{j_0}$ and obtain all tuples $\{t^{(i)} : i=1, \ldots, M\}$ in SCR_i using algorithm **CST**. Get the top-*N* tuples as Step 1 using **TTC**.

Step 3. For each SCR_i, $i \ge 1$, that has a super region, find its best-super-region SCR_k, and get all the tuples in SCR_i from SCR_k. Then follow the same procedures as Step 1(a) and Step 1(b) for each region SCR_i.

Case 2: individual queries.

Step 1. For each R_i, if it has no super region, find all the clustered R_j's that intersect with R_i, and let R_{j_0} be the one with the largest intersection. All candidates in R_{j_0} have been retrieved in Case 1 above. Calculate $R_i - R_{j_0}$ using algorithm **DTR**, and obtain candidates of R_i using **CST**, then get the top-N tuples using **TTC**.

Step 2. If R_i has a super region, it is sufficient to get all the candidates in R_i from its best-super-region R_k and then obtain the top-N tuples for R_i using **TTC**.

There are two special situations in Case 1 and Case 2. If two queries are the same, it suffices to obtain the top-N tuples only one time; if a search region is the same as its best-super-region, it is sufficient to get the candidates once.

4.4.2 Guarantee exact top-*N* tuples

If there are not at least N tuples in the search region R of a top-N query (Q, N), i.e. $|R| < N$, then the process of Section 4.4.1 cannot guarantee to get the top-N tuples. Two cases are considered as follows.

1. R has a best-super-region R_k or *SCR*, and the size of R_k or *SCR* contains at least N tuples. Suppose $|R| = N'$, take $N - N'$ tuples from $R_k - R$ or $SCR - R$, and calculate the distances between those tuples and the query Q. Now we use the maximum distance to define the new search region for Q and this search region guarantees at least N tuples will be retrieved. Get the top-N *tuples*-using algorithm **TTC**.
2. R has no super region or the number of tuples in its super region is also less than N. Use an existing method such as the learning-based method in [43] or the histogram-based method in [5], to determine a larger search region that contains at least N tuples.

5. EXPERIMENTAL RESULTS

In this section, we report our experimental results and compare our Region Clustering Method (**RCM**) against the Naïve Method (**NM**) for evaluating multiple top-N queries. Our experiments are carried out using Microsoft's SQL Server 2000 and VC++6.0 on a PC with Windows XP and a Pentium 4 processor with 2.8GHz CPU and 768MB memory.

5.1 Data Sets and Preparations

The datasets we used include data of both low dimensionality (2, 3, and 4 dimensions) and high dimensionality (25, 50, and 104 dimensions). For low-dimensional datasets, both synthetic and real datasets used in [5] are used. The real datasets include Census2D and Census3D (both with 210,138 tuples), and Cover4D (581,010 tuples). The synthetic datasets are Gauss3D (500,000 tuples) and Array3D (507,701 tuples). In the names of all datasets, suffix "nD" indicates that the dataset has *n* dimensions. For high-dimensional datasets, real datasets derived from LSI are used in our experiments. They have 20,000 tuples and the same 25, 50 and 104 attributes as used in [11] are used to create datasets of 25, 50 and 104 dimensions, denoted by Lsi25D, Lsi50D and Lsi104D, respectively.

Each dataset uses three training-workloads tw_1, tw_2 and tw_3, and three test workloads w_1, w_2 and w_3. Each training-workload includes 100 queries and each test workload includes 2,000 queries that are the tuples randomly selected from the dataset used respectively. Three training-workloads will be used to determine the clustering models and their threshold values, and three test workloads will be used to report our experimental results, which are the average of the results of the three test workloads.

Two top-*N* query evaluation techniques are used to construct the search region of a top-*N* query:

1. *Optimum (Opt) technique* [5]: As a baseline, we consider the ideal technique that uses the smallest search region containing the actual top-*N* tuples for a given query. The smallest search region is obtained using the *sequential scan technique*. That is, first obtain the top-*N* tuples from scanning the entire dataset, and then construct the search region that just contains the top-*N* tuples for each query.
2. *Learning-based (LB) technique* [43]: For a given dataset D, a set of data points is randomly selected from D and these data points are used as sample query points. For each such query point, a top-N query (Q_i, N_i) is generated and a profile is created for it. The profile contains the search distance r_i of the smallest search region of (Q_i, N_i) (obtained using the *sequential scan technique*) and the number of tuples in this search region. Once the profiles are created and saved, the learning-based technique uses them to estimate the search engine for any new top-*N* query, (Q, N). More specifically, we first identify the profiles whose query points are closest to Q, then we use the information in these profiles to estimate the local density of Q, and finally we construct the search region for (Q, N) (please refer to [43] for more details about this method). The experimental results reported in [43] indicate that using less than 1,000 sample queries to create the profiles is sufficient. For the experiments conducted in this paper, 178, 218, 250, 833, 909 and 954 profiles are created for datasets of 2, 3, 4, 25, 50 and 104 dimensions, respectively. These numbers are determined based on the consideration that the total size of the profiles does not exceed the size of the histogram used in [5] and the size of sampling set used in [11] when these methods are compared using the same dataset [43].

For any given top-N query, the search region generated by the Opt method is usually smaller than that generated by the LB method because the former is the smallest possible. With smaller search regions, the likelihood of overlap among them will be reduced. Consequently, I-Clustering and C-Clustering may become less effective.

In our experiments, we report results based on a default setting. This default setting uses a 2,000-query workload. For low dimensional datasets (2, 3, and 4 dimensions), the default setting has $N = 100$ (i.e., retrieve top 100 tuples for each query) and the distance function is the *maximum distance* (i.e., L_∞-norm distance); for high dimensional datasets (25, 50, and 104 dimensions), the default setting has $N = 20$ and the distance function is *Euclidean distance* (i.e., L_2-norm distance) [43].

The following measures are used in our experiments:

- *The elapsed time (millisecond, ms) used to obtain top-N tuples*: The sum of the times needed to retrieve the top-*N* tuples from the respective dataset for all queries. The time for constructing the search regions is not included because this time is the same for both Naïve and RCM query evaluation strategies. In addition, the time to construct the search regions is relatively small compared to the time needed to retrieve the tuples from the regions. For example, it takes about 3 ms to construct one search region and about 150 ms to retrieve the data from a region for Lsi25D.
- *The number of I/Os*: It is the total number of I/O operations for accessing the database.
- *The extra time used by* **RCM** *(millisecond, ms)*: It is the sum of the times needed to perform the clustering using our algorithms, i.e., it corresponds to the δ in Formula (2) in Section 3.
- *The number of tuples retrieved*: The total number of tuples retrieved in order to obtain the top-*N* tuples from the respective dataset for all queries. A smaller number of tuples retrieved indicates a better efficiency.

The values of the above four measures depend on the following set of factors {*m*, D, T, A, W}, where

m: The number of top-*N* queries to be evaluated concurrently. The following values are considered: 1, 4, 10, 40, 100, 400, 1000 and 2000.

D: The dataset used. Its valid "values" include Census2D, Census3D, Array3D, Gauss3D, Cover4D, Lsi25D, Lsi50D, and Lsi104D.

T: The technique used to obtain the search region of a top-*N* query. Its valid "values" are LB (Learning-based technique) and Opt (Optimum technique).

A: The algorithm used to retrieve the top-*N* tuples for the *m* top-*N* queries. It's either **RCM** or **NM**. The naïve method (**NM**) is frequently used as the baseline to evaluate other methods of processing multiple queries [16] because it is simple, easy to understand, and generally usable.

W: The workloads: three workloads, w_1, w_2 and w_3, are used to get the original values of the performance measures, and then their average is taken to obtain the reported results.

Because *the elapsed time* and *the extra time* do not include the time needed to construct the search regions (the RC( ) in Formulas (1) and (2) in Section 3), from Formulas (1) and (2) in Section 3, we have the following formulas:

$$\text{Elapsed-time}(m, \text{D}, \text{T}, \textbf{NM}, \text{W}) = (\textit{IO-time}(Q_1, N_1) + \textit{S-time}(Q_1, N_1)) + \ldots + (\textit{IO-time}(Q_m, N_m) + \textit{S-time}(Q_m, N_m)) \quad (1')$$

$$\text{Elapsed-time}(m, \text{D}, \text{T}, \textbf{RCM}, \text{W}) = (\textit{IO-time}(SCR_1) + \ldots + \textit{IO-time}(SCR_M)) + (\textit{S-time}(Q_1, N_1) + \ldots + \textit{S-time}(Q_m, N_m)) + \text{Extra-time}(m, \text{D}, \text{T}, \textbf{RCM}, \text{W}) \quad (2')$$

where *IO-time*( ) is *the elapsed time* of the I/O operations for accessing the database and *S-time*( ) is *the elapsed time* of sorting the results. In (2'), $\{SCR_k : k = 1,.., M\}$ includes all single-region clusters.

It is well known that the cost of all the algorithms that we consider depend on the size of the buffer in main memory. When presenting cost estimates we generally assume the worst-case scenario [34] and let the size of the buffer be as small as possible. In Microsoft's SQL Server 2000, we configure the '*max server memory*' to be 5MB for Lsi50D, 8MB for Lsi104D, and 4MB for the other datasets. Note that 4MB is the smallest value for the '*max server memory*' in Microsoft's SQL Server 2000.

5.2 Determining Clustering Model and Thresholds by Training

Each of the three clustering models we introduced in Section 4.2 has a threshold whose value can affect the quality of the produced clusters. From the discussion in Section 4.2, we have $0 \le c_1 \le 1$, $0 \le c_2 \le 1$, and $1 \le c_3 \le 3$, where c_1, c_2 and c_3 are the thresholds for I-clustering, C-clustering and S-clustering, respectively. In this section, we discuss how to obtain the appropriate values for these thresholds for each dataset.

Our idea is to use the three training workloads associated with each dataset to perform this task. Let D be a dataset under consideration and tw_1, tw_2 and tw_3 be the associated training workloads. Our training process is as follows. First, we obtain two sets of search regions for the queries in each training workload, one based on the LB technique and the other based on the Opt technique. Next, for each set of search regions, we run two versions of our **RCM** algorithm, one employs only the I-clustering and the other uses only the C-clustering (S-clustering will not be used in both cases), with different values of c_1 and c_2, respectively. Based on the performances of these experiments, we find the best values for c_1 and c_2 and determine which of the two primary clustering models (i.e., I-clustering and C-clustering) is better based on their best threshold values. Finally, we add the S-clustering to the **RCM** algorithm for each best case and try different values of c_3 between 1 and 3 to determine the best value for c_3.

Example 3. Consider the dataset Gauss3D in Section 5.1. For each training-workload tw_i, $i=1, 2, 3$, use each of the LB and Opt techniques to get 100 search regions, respectively. Then apply I-clustering and C-clustering in the **RCM** algorithm with c_1 and c_2 being 0, 0.05, 0.1, 0.15, …, 0.95, and 1, respectively. Based on the *number of I/Os* we obtained for each run using the I-clustering, we compute the average of the number of I/O requests for the 3 training workloads and then the ratios of the number of I/Os of **RCM** to **NM**:

average-num-IO1(*100*, Gauss3D, T, **RCM**) = ($\sum_{i=1}^{3}$ Num-IO(*100*, Gauss3D, T, **RCM**, tw_i))/3

Ratio-IO1(*100*, Gauss3D, T) = average-num-IO1(*100*, Gauss3D, T, **RCM**) /100

Similarly, based on the *number of tuples retrieved* we obtained for each run using the I-clustering, we compute the average of the number of tuples retrieved for the 3 training workloads and then the ratios of the number of tuples retrieved by **RCM** to that by **NM**:

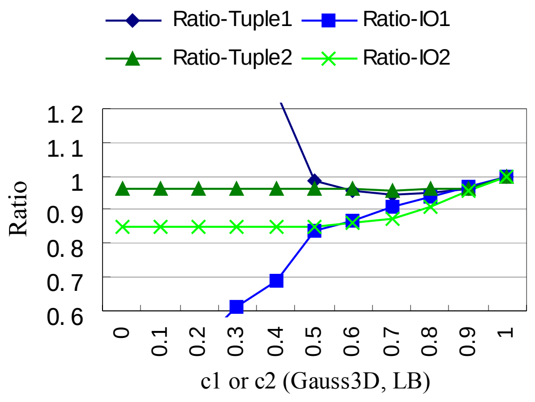
average-num-tuple1(*100*, Gauss3D, T, **RCM**) = ($\sum_{i=1}^{3}$ Num-Tuple(*100*, Gauss3D, T, **RCM**, tw_i))/3

Ratio-Tuple1(*100*, Gauss3D, T) =

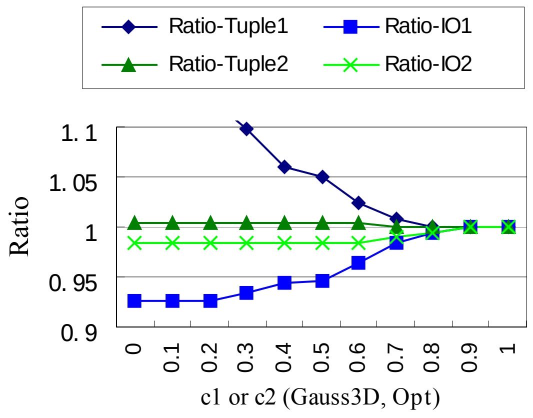
average-num-tuple1(*100*, Gauss3D, T, **RCM**) /average-num-tuple1(*100*, Gauss3D, T, **NM**)

We can repeat the above computations for C-clustering. Let “average-num-IO2”, “Ratio-IO2”, “average-num-tuple2” and “Ratio-Tuple2” denote the corresponding averages and ratios for C-clustering.

Figure 10(a) shows the results for different values of c_1 and c_2 using the regions constructed by the LB technique. Obviously, 0.5 is a pivotal point with I-clustering; when c_1 and c_2 are 0.5, the *number of I/Os* of I-clustering is less than that of C-clustering, and the *numbers of tuples retrieved* are almost the same for both clustering models.



(a) The regions constructed by LB technique



(b) The regions constructed by Opt technique

Fig.10. Ratios of the number of I/Os and ratios of the number of tuples retrieved by training workloads for Gauss3D

Figure 10(b) shows the results where the search regions are constructed using the Opt technique. We can see that when c_1 and c_2 are 0.5, the ratio of tuples retrieved using I-clustering is about 1.05, and the radio of I/Os of I-clustering is less than that of C-clustering. Since the number of I/Os plays a bigger role in efficiency, we choose I-clustering with threshold $c_1 = 0.5$ as the primary clustering model for dataset Gauss3D. □

Table 1 lists the primary clustering model and the approximate optimal thresholds that are obtained for each dataset using the training workloads tw_1, tw_2, and tw_3. All experimental results to be reported in the subsequent sections are based on the thresholds in Table 1.

Table 1: Primary clustering model and thresholds for all datasets

Dataset	Primary Model		Secondary Model (S-clustering)
	I-clustering	C-clustering	
Census2D	$c_1 = 0.25$	-	$c_3 = 1.5$
Census3D	$c_1 = 0.25$	-	$c_3 = 1.5$
Gauss3D	$c_1 = 0.5$	-	$c_3 = 2$
Array3D	-	$c_2 = 0.5$	$c_3 = 2$
Cover4D	$c_1 = 0.25$	-	$c_3 = 1.5$
Lsi25D	-	$c_2 = 0.25$	$c_3 = 1$
Lsi50D	-	$c_2 = 0.1$	$c_3 = 1$
Lsi104D	-	$c_2 = 0.0$	$c_3 = 1$

Table 1: Primary clustering model and thresholds for all datasets

5.3 Performance Comparison

Using the thresholds in Table 1, for each of the 8 datasets D, each of the three test workloads, and one of the two techniques LB and Opt for generating search regions, we first obtain 2,000 search regions corresponding to the 2,000 queries, then we use *m* regions(s) to carry out one group of experiments, where *m* takes one of the following 8 values: 1, 4, 10, 40, 100, 400, 1000 and 2000. Thus, there are 8∗3∗2∗8 = 384 groups of experimental results.

5.3.1 Comparison of the Elapsed Time

We use the ratio of **RCM**'s performance to **NM**'s performance to compare **RCM** and **NM**. Denote

average-elapsed-time(m, D, T, A) = ($\sum_{i=1}^{3}$ Elapsed-time(m, D, T, A, w_i))/3

ratio-elapsed-time(m, D, T) = average-elapsed-time(m, D, T, **RCM**) / average-elapsed-time(m, D, T, **NM**)

Figure 11(a) shows that, using the regions constructed by the LB-technique, ratio-elapsed-time < 1.0 for all datasets when $m \geq 10$, except for Census3D which has the ratio = 1.0 when $m = 10$, and ratio-elapsed-time ≤ 0.82 for all datasets when $m = 100$, i.e., **RCM** is more efficient than **NM** when $m \geq 10$. Even when $m < 10$, we have Ratio-time ≈ 1.0. This means that **RCM** is generally usable for all *m*.

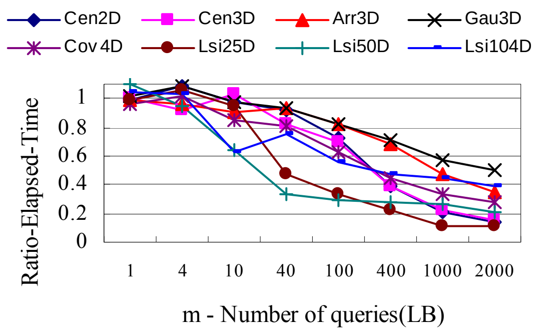
When $m = 100$, Figure 11(a) shows that there are four datasets (Cover4D, Lsi25D, Lsi50D, and Lsi104D) with ratio-elapsed-time < 0.62 (for Lsi25D and Lsi50D, the ratios are 0.33 and 0.28 respectively), that is, the elapsed time of **RCM** is less than 62% of that of **NM** for these datasets. Therefore, our **RCM** can save the time by at least 38% for these four datasets, and save the time by 67% and 72% for datasets Lsi25D and Lsi50D, respectively. The ratios for the other four datasets (Census2D, Census3D, Array3D and Gauss3D) are between 0.62 and 0.82.

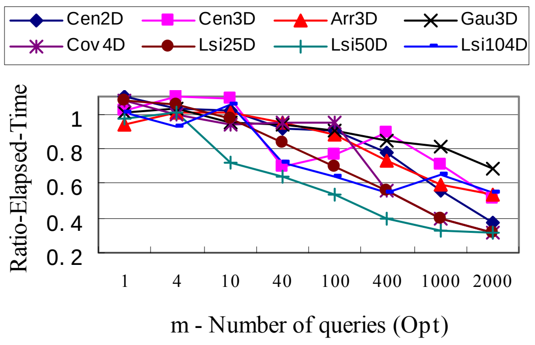
We can see that more savings on elapsed time can be achieved with the increase of the number of queries from 400 to 2000. When $m = 2000$, in the worst case, **RCM** can save the time by 50% for Gauss3D; but in the best case, **RCM** can save the time by nearly 90% for Census2D, Census3D and Lsi25D.

We should point out that when the number of queries *m* becomes sufficiently large, the I/O cost for evaluating most additional queries would become zero. This occurs when the SCRs of the formed clusters cover almost the entire space, and as a result, the search region of any additional query considered will likely be contained in one of the existing SCRs, leading to no additional I/O operation. In other words, when *m* becomes sufficiently large, the total I/O cost approaches a constant when **RCM** is used. For example, for Lsi25D, the number of I/O operations by **RCM** is 70 for both $m = 1000$ and $m = 2000$. Moreover, if we assume that every query incurs the same I/O cost, *IO-*

time, and the same sorting cost, *S-time*, based on the naïve method, then the ratio-elapsed-time of **RCM to NM** is bounded by *S-time*/(*IO-time* + *S-time*), which represents the extreme case when **RCM** incurs no I/O cost.

Figure 11(b) shows the results when the search regions are constructed using the Opt technique. When m = 100, all ratios are between 0.52 and 0.9, except for cover4D which has the ratio = 0.95. Note that the Opt technique is the theoretical best for evaluating individual top-N queries (i.e., using **NM**) and it cannot be implemented in practice. The results in Figure 11(b) show that **RCM** can outperform **NM** even when the smallest search region is used for each query; for m = 2000, all ratios are between 0.3 and 0. 7.





(a) The regions constructed by LB technique (b) The regions constructed by Opt technique

Fig.11. The ratios of elapsed times of RCM to NM

In Figures 11(a) and (b), the ratios should be close to 1.0 if no cluster contains two or more regions. However, there are some ratios that are greater than 1.0, say, when m = 1 for Lsi50D in Figure 11(a) and when m = 4 for Census3D in Figure 11(b). The primary reason is that the total elapsed time is very small in these cases so that a small random perturbation can cause the ratio to have a large change. The effect of random perturbation diminishes when m becomes larger. This will also be discussed in Section 5.3.3.

5.3.2 Comparison of the Number of I/Os

The total number of I/O requests for accessing the database plays an important role in the total response time of queries. In fact, I/O cost of accessing the database is usually the dominating cost of processing database queries. We denote

$$\text{average-num-IO}(m, \text{D, T, A}) = (\textstyle\sum_{i=1}^{3} \text{Num-IO}(m, \text{D, T, A}, w_i))/3.$$

Because average-num-IO(m, D, T, **NM**) is always m, we have

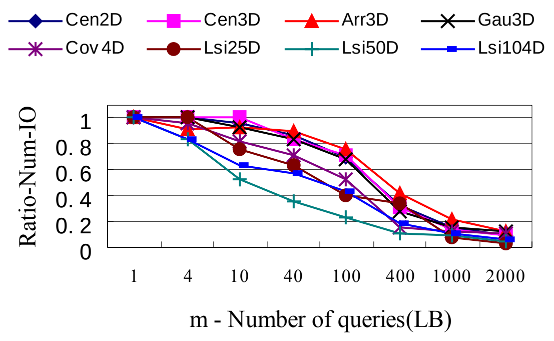
$$\text{ratio-num-IO}(m, \text{D, T}) = \text{average-num-IO}(m, \text{D, T, }\mathbf{RCM}) / \text{average-num-IO}(m, \text{D, T, }\mathbf{NM})$$

= average-num-IO(m, D, T, **RCM**)/m.

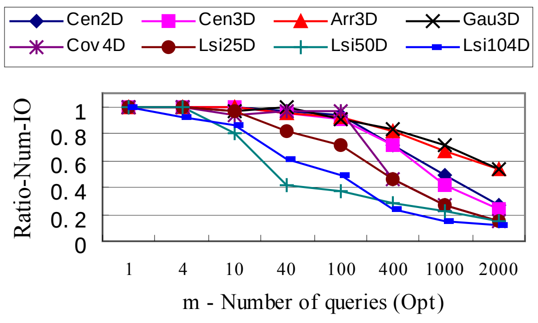
Figure 12(a) shows that, using the search regions constructed by the LB technique, ratio-num-IO < 1.0 for all datasets when $m \geq 10$, except for Census3D which has ratio 1.0 when $m = 10$. For $m = 100$, the ratio is less than 0.6 for four datasets (Cover4D, Lsi25D, Lsi50D and Lsi104D) and between 0.6 and 0.8 for the other four datasets. As mentioned in Section 5.3.1, when the number of queries becomes sufficiently large, the total I/O cost of **RCM** will approach a constant. Consequently, the ratio-num-IO will approach zero when m becomes larger and larger because the total I/O cost of **NM** will keep on increasing as m increases.

Figure 12(b) shows the case when the search regions are constructed using the Opt technique. When $m \geq 40$, the ratios are less than < 1.0 for all datasets. Because the Opt technique yields the smallest search regions, which means there is less chance to form clusters. As a result, the ratio-num-IO has a slower speed of approaching zero compared with the case when the LB technique is used to generate search regions.

Comparing Figures 11(a)-(b) with Figures 12(a)-(b), respectively, we can see that the curves of ratio-elapsed-time and the curves of ratio-num-IO have similar trends. This indicates that the number of I/O requests has a direct and proportional impact on the elapsed time for sufficiently large m.



(a) The regions constructed by LB technique



(b) The regions constructed by Opt technique

Fig.12. Ratios of the number of I/Os of RCM to NM

5.3.3 Comparison Between the Extra Elapsed Time and Total Elapsed Time Using RCM

In this subsection, we discuss the extra elapsed time for **RCM**. From Formula (2') in Section 5.1 and the first formula in Section 5.3.1, we have:

extra-time(m, D, T, **RCM**, W) = elapsed-time(m, D, T, **RCM**, W) –

((*IO-time* (SCR_1) + …+ *IO-time* (SCR_k)) + (*S-time* (Q_1, N_1) + …+ *S-time* (Q_m, N_m)))

average-elapsed-time(m, D, T, **RCM**) = ($\sum_{i=1}^{3}$ elapsed-time(m, D, T, **RCM**, w_i))/3

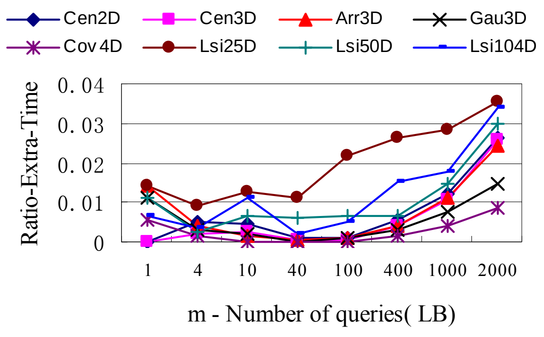
We denote

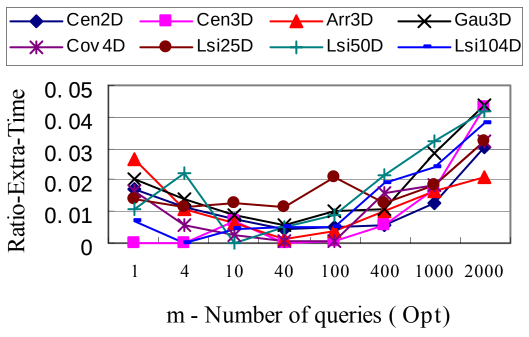
average-extra-time(m, D, T, **RCM**) = ($\sum_{i=1}^{3}$ extra-time(m, D, T, **RCM**, w_i))/3

ratio-extra-time(m, D, T) = average-extra-time(m, D, T, **RCM**)/average-elapsed-time(m, D, T, **RCM**)

From Figures 13(a) and (b), we can see that all the values for ratio-extra-time are very small. In fact, none of the ratios exceed 0.04 and 0.05, respectively.

We can see that the curve of ratio-extra-time rises with the increase of the number of queries when $m \geq 100$ for each dataset because of two factors, i.e., the extra time will become larger and the total elapsed time of **RCM** increases more slowly as the total I/O cost approaches a constant. Nevertheless, the extra time remains very small compared with the total elapsed time of **RCM**. When m is small ($m \leq 10$), the total elapsed time is very small; therefore, a small random perturbation can lead to the curve of ratio-extra-time to have a large change.





(a) The regions constructed by LB technique (b) The regions constructed by Opt technique

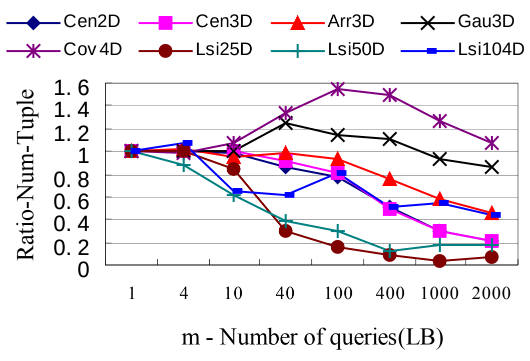
Fig.13. The ratios of extra time to total elapsed time

5.3.4 Comparison of the Number of Tuples Retrieved

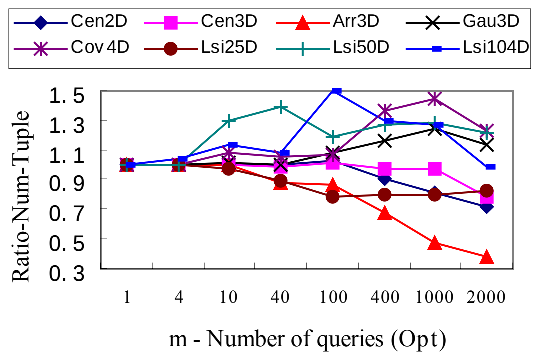
With the requirement that the top-N tuples for each query must be retrieved, an evaluation method is usually more efficient if it retrieves a smaller number of tuples because this means less useless tuples would be retrieved. We denote

average-num-tuple(m, D, T, A) = ($\sum_{i=1}^{3}$ Number-Tuple (m, D, T, A, w_i)) /3

ratio-num-tuple(m, D, T) = average-num-tuple (m, D, T, **RCM**) / average-num-tuple (m, D, T, **NM**)



(a) The regions constructed by LB technique



(b) The regions constructed by Opt technique

Fig.14. The ratios of the number of all tuples retrieved by RCM to that by NM

In Figure 14(a), for regions generated by the LB technique, the curves of ratio-num-tuple are either near 1 or below 1 except for datasets Cover4D and Gauss3D. Actually for most datasets, when $m \geq 40$, the ratios are less than 1. In other words, other than for datasets Cover4D and Gauss3D, the number of tuples retrieved by the **RCM** method is either the same as (usually for small *m*) or less than (usually for larger *m*) that by the Naïve Method. The reason for the exception of Cover4D and Gauss3D is due to the nature of the two datasets (their data distributions) and their parameters (see Table 1 in Section 5.2). The two datasets all use I-clustering as their primary clustering model, which tends to yield larger lacunas in the *SCRs* (smallest containing regions) (see Section 4.2), causing more tuples to be not in the search regions of the queries.

In Figure 14(b), for regions generated by the Opt technique, when $m > 10$, the curves of ratio-num-tuple are either near 1 or below 1 except for datasets Cover4D, Gauss3D, Lsi50D and Lsi104D. So the number of all tuples retrieved by the **RCM** method is almost equal to that by the Naïve Method even when optimal regions are used except for datasets Cover4D, Gauss3D, Lsi50D and Lsi104D. The reasons for the exceptions are that smaller clustering thresholds tend to generate larger lacunas in the *SCRs*, and c_2 is 0.1 and 0.0 for Lsi50D and Lsi104D, respectively (see Section 4.2 and Table 1 in Section 5.2). Note that the number of all tuples retrieved by the Naïve Method is the smallest possible for regions generated by the Opt technique. This shows that **RCM** performs very well.

6. CONCLUSIONS

In this paper, we proposed a new method for evaluating multiple top-*N* queries concurrently over a relational database. The basic idea of this method is region clustering that groups the search regions of individual top-*N* queries into larger regions and retrieves the tuples from the larger regions. This method tends to avoid accessing the same region multiple times and reduce the number of random I/O accesses to the underlying databases. Our region clustering method is based on three basic clustering models, namely I-clustering, C-clustering and S-clustering, with different properties in terms of the compactness of the generated clusters and the applicability to region clustering. We carried out extensive experiments to evaluate the effectiveness of the proposed method based on eight different datasets with dimensionality ranging from 2 to 104, the number of concurrent queries considered ranging from 1 to 2,000, and the initial search regions obtained using two techniques (Learning-based and optimum). Our experimental results showed that our region clustering method yielded significant cost savings over the naïve one-query-at-a-time method for both low-dimensional and high-dimensional data. When 2,000 concurrent queries are considered and when more realistic initial search regions (the ones generated by the learning-based method) are used, the newly proposed method reduced the elapsed-time of the naïve method by 50 to 90% with most saving coming from the saving on I/O cost.

We believe our method is directly applicable to evaluating multiple range queries in multi-dimensional spaces. Since unlike top-*N* queries, traditional range queries do not involve sorting the final results, the region clustering method may have more relative advantage in evaluating multiple range queries than evaluating multiple top-*N* queries. In the future, we plan to apply this method to process multiple traditional range queries and evaluate its effectiveness.

The experiments conducted in this paper assumed that a single server evaluates all the queries. As such, the results are directly applicable to a client/server environment where the server evaluates all the queries sent in from different clients. It would be interesting to consider a distributed environment where clusters of queries may be formed and evaluated at different nodes of the distributed system. Another interesting issue is to consider the case when queries come in as a stream and to devise a strategy for clustering a stream of queries.

ACKNOWLEDGMENT: This work is supported in part by: NSFC (60496322, 60496327) and NSFHEE (2004305). The authors would also like to express their gratitude to the anonymous reviewers for providing some very helpful suggestions.

REFERENCES

[1] H. Andrade, T. Kurç, A. Sussman, J. Saltz. Optimizing the Execution of Multiple Data Analysis Queries on Parallel and Distributed Environments. IEEE Trans. Parallel Distrib. Syst. 15(6) (2004) 520 - 532.

[2] W. Balke, W. Nejdl, W. Siberski, U. Thaden. Progressive Distributed Top-k Retrieval in Peer-to-Peer Networks, in: Proceedings of the 21st International Conference on Data Engineering (ICDE'05), 2005, 174 - 185.

[3] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum. IO-Top-k: Index-access Optimized Top-k Query Processing, in: Proceedings of 29th International Conference on Very Large Data Bases (VLDB'06), 2006, 475 - 486.

[4] S. Berchtold, C. Böhm, D. A. Keim, H.-P. Kriege, X. Xu. Optimal Multidimensional Query Processing Using Tree Striping, in: Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery (DaWaK'00), 2000, 244 - 257.

[5] N. Bruno, S. Chaudhuri, L. Gravano. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. ACM Transactions on Database Systems 27 (2) (2002) 153 -187.

[6] M. Carey, D. Kossmann. On saying "Enough Already!" in SQL, in: Proceedings ACM International Conference on Management of Data (SIGMOD '97), 1997, 219-230.

[7] M. Carey, D. Kossmann. Reducing the braking distance of an SQL query engine, in: Proceedings of 24th International Conference on Very Large Data Bases (VLDB '98), 1998, 158-169.

[8] K. Chakrabarti, M. Ortega-Binderberger, S. Mehrotra, K. Porkaew. Evaluating Refined Queries in Top-k Retrieval Systems. IEEE Trans. Knowl. Data Eng. 16(2) (2004) 256-270.

[9] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S.Li, M.-L. Lo, J. R. Smith. The Onion Technique: Indexing for Linear Optimization Queries, in: Proceedings ACM International Conference on Management of Data (SIGMOD '00), 2000, 391-402.

[10] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. IEEE Trans. Knowl. Data Eng. 16(8) (2004) 992–1009.

[11] C. Chen, Y. Ling. A sampling-based estimator for top-k selection query, in: Proceedings of the 18th International Conference on Data Engineering (ICDE'02), 2002, 617-627.

[12] G. Das, D. Gunopulos, N. Koudas. Answering Top-k Queries Using Views, in: Proceedings of 29th International Conference on Very Large Data Bases (VLDB'06), 2006, 451 - 462.

[13] D. Donjerkovic, R. Ramakrishnan. Probabilistic optimization of top N queries, in: Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), 1999, 411-422.

[14] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe systems, SIGMOD Conference 2001, 115-126.

[15] R. Fagin, A. Lotem, M. Naor. Optimal aggregation algorithms for middleware, in: Proceedings of the Twentieth ACM Symposium on Principles of Database Systems (PODS '01), 2001, 102 – 113.

[16] R. Fenk, V. Markl, R. Bayer. Improving multidimensional range queries of non-rectangular volumes specified by a query box set, in: Proceedings of Databases, Web and Cooperative Systems (DWACOS) 1999.

[17] D. Habich, W. Lehner, A. Hinneburg. Optimizing multiple top-K queries over joins, in: Proceedings of the 17th international conference on Scientific and statistical database management, 2005,195 - 204.

[18] V. Hristidis, N. Koudas, Y. Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries, in: Proceedings of the 2001 ACM International Conference on Management of Data (SIGMOD '01), 2001, 259-270.

[19] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. VLDB Journal, 13(1) (2004) 49–70.

[20] I. Ilyas, W. Aref, A. Elmagarmid. Supporting top-k join queries in relational databases. VLDB J. 13(3) (2004) 207-221.

[21] I. Ilyas, W. Aref, A. Elmagarmid. Joining ranked inputs in practice, in: Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02), 2002, 950–961.

[22] I. Ilyas, R. Shah, W. Aref, J. Vitter, A. Elmagarmid. Rank-aware query optimization, in: Proceedings ACM International Conference on Management of Data (SIGMOD'04) , 2004, 203–214.

[23] J. Lee, D. Kim, C. Chung. Multi-dimensional Selectivity Estimation Using Compressed Histogram Information, in: Proceedings ACM International Conference on Management of Data, (SIGMOD'99), 1999, 205-214.

[24] C. Li, K. Chang, I. Ilyas, S. Song. RankSQL, Query Algebra and Optimization for Relational Top-k Queries, in: Proceedings ACM International Conference on Management of Data (SIGMOD'05), 2005, 131-142.

[25] A. Marian, N. Bruno, L. Gravano. Evaluating top-k queries over web-accessible databases, ACM Trans. Database Syst. 29(2) (2004) 319-362.

[26] W. Meng, C. Yu, W. Wang, N. Rishe. Performance Analysis of Three Text-Join Algorithms, IEEE Trans. Knowl. Data Eng. 10(3) (1998) 477-492.

[27] S. Michel, P. Triantafillou, G. Weikum. KLEE: A Framework for Distributed Top-k Query Algorithms, in: Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05), 2005, 637-648.

[28] A. Motro. VAGUE: A user interface to relational databases that permits vague queries. ACM Trans. Office Inf. Syst. 6 (3) (1988), 187–214.

[29] K. Mouratidis, S. Bakiras, D. Papadias. Continuous Monitoring of Top-k Queries over Sliding Windows. in: Proceedings ACM International Conference on Management of Data (SIGMOD'06), 2006, 635-646.

[30] P. O'Neil and E. O'Neil. Database: Principles, Programming, and Performance, 2nd edition, Morgan Kaufmann Publishers, 2001.

[31] D. Papadias, Q. Shen, Y.Tao, K. Mouratidis. Group Nearest Neighbor Queries, in: Proceedings of the 20th International Conference on Data Engineering (ICDE'04), 2004, 301-312.

[32] T. Sellis, Multiple-Query Optimization. ACM Trans. Database Syst. 13(1)(1988) 23-52.

[33] T. Sellis, S. Ghosh, On the Multiple-Query Optimization Problem, IEEE Trans. Knowl. Data Eng. 2(2) (1990) 262-266.

[34] A. Silberschatz, H. F. Korth, S. Sudarshan. Database System Concepts, 4th Edition. McGraw-Hill, 2002.

[35] Ion Stoica. A Time-Optimal Multiple-Query Nearest-Neighbor Algorithm on Meshes with Multiple Broadcasting, International Journal of Pattern Recognition and Artificial Intelligence 9(4) (1995) 663-677.

[36] I. Taksa. Predicting the Cumulative Effect of Multiple Query Formulations, International Symposium on Information Technology: Coding and Computing (ITCC(2)'05), 2005, 491-496.

[37] M. Theobald, G. Weikum, R. Schenkel. Top-k Query Evaluation with Probabilistic Guarantees, in: Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB'04), 2004, 648-659.

[38] M. Wojciechowski, M. Zakrzewicz. On Multiple Query Optimization in Data Mining, in: Proc. Advances in Knowledge Discovery and Data Mining, 9th Pacific-Asia Conference (PAKDD'05), Lecture Notes in Computer Science, 3518 (Springer 2005), 696-701.

[39] D. Xin, J. Han, H. Cheng, X. Li. Answering Top-k Queries with Multi-Dimensional Selections: The Ranking Cube Approach, in: Proceedings of 29th International Conference on Very Large Data Bases (VLDB'06), 2006, 463 – 474.

[40] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views, in: Proceedings of the 19th International Conference on Data Engineering (ICDE'03), 2003, 189–200.

[41] C. Yu, G. Philip, W. Meng. Distributed Top-N Query Processing with Possibly Uncooperative Local Systems, in: Proceedings of 29th International Conference on Very Large Data Bases (VLDB'03), 2003, 117-128.

[42] C. Yu, P. Sharma, W. Meng, Y. Qin. Database selection for processing k nearest neighbors queries in distributed environments, in: ACM/IEEE Joint Conference on Digital Libraries (JCDL), 2001, 215-222.

[43] L. Zhu, W. Meng. Learning-Based Top-N Selection Query Evaluation over Relational Databases, in: Advances in Web-Age Information Management: 5th International Conference (WAIM'04), Lecture Notes in Computer Science, 3129 (Springer 2004), 197-207.

[44] M. Zhu, D. Papadias, J. Zhang, D. Lee. Top-k Spatial Joins, IEEE Trans. Knowl. Data Eng. 17(4) (2005), 567-579.