Keyword Search in Spatial Databases: Towards Searching by Document

Dongxiang Zhang [#1], Yeow Meng Chee [†2], Anirban Mondal [*3], Anthony K. H. Tung [#4], Masaru Kitsuregawa [*5]

[#]*School of Computing, National University of Singapore,* {[1]zhangdo,[4]at}@comp.nus.edu.sg

[†]*School of Physical and Mathematical Sciences, Nanyang Technological University,* [2]ymchee@ntu.edu.sg

[*]*Institute of Industrial Science, University of Tokyo,* {[3]anirban@,[5]kitsure}@tkl.iis.u-tokyo.ac.jp

***Abstract*— This work addresses a novel spatial keyword query called the m-closest keywords (mCK) query. Given a database of spatial objects, each tuple is associated with some descriptive information represented in the form of keywords. The mCK query aims to find the spatially closest tuples which match m user-specified keywords. Given a set of keywords from a document, mCK query can be very useful in geotagging the document by comparing the keywords to other geotagged documents in a database.**

To answer mCK queries efficiently, we introduce a new index called the bR*-tree, which is an extension of the R*-tree. Based on bR*-tree, we exploit a priori-based search strategies to effectively reduce the search space. We also propose two monotone constraints, namely the distance mutex and keyword mutex, as our a priori properties to facilitate effective pruning. Our performance study demonstrates that our search strategy is indeed efficient in reducing query response time and demonstrates remarkable scalability in terms of the number of query keywords which is essential for our main application of searching by document.

I. Introduction

With the ever-increasing popularity of services such as Google Earth and Yahoo Maps, as well as other geographic applications, queries in spatial databases have become increasingly important in recent years. Current research on queries goes well beyond pure spatial queries such as nearest neighbor queries [23], range queries [18], and spatial joins [7], [21], [17], [19]. Queries on spatial objects associated with textual information represented by sets of keywords are beginning to receive significant attention from the spatial database research community and the industry.

This paper focuses on a novel type of query called the ***m*-closest keywords** (mCK) query: given m keywords provided by the user, the mCK query aims at finding the closest tuples (in space) that match these keywords. While such a query has various applications, our main interest lies in that of a **search by document**.

As an example, Fig. 1 shows the spatial distribution of three keywords that are obtained from placemarks in some mapping application. Given a blog that contains these three keywords, the user is interested to find a spatial location that the blog is likely to be relevant to [1]. This can be done by issuing an mCK query on the three keywords. The measure of closeness for a set of m tuples is defined as the maximum distance between any two of the tuples:

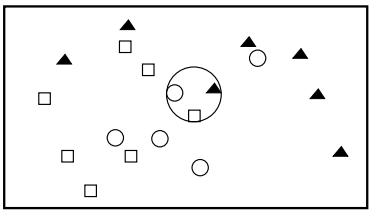


Fig. 1. mCK query for three keywords obtained from placemarks

Definition 1 (Diameter): Let $\mathcal{S}$ be a tuple set endowed with a distance metric $dist(\cdot,\cdot)$. The *diameter* of a subset $\mathcal{T} \subseteq \mathcal{S}$ is defined by

$$diam(\mathcal{T}) = \max_{T,T' \in \mathcal{T}} dist(T, T').$$

Different distance metric will give rise to different geometry of the query response:

- If $dist(\cdot,\cdot)$ is the ℓ_1-distance metric, then the response containing all the keywords of the query is a square oriented at a 45° angle to the coordinate axes.
- If $dist(\cdot,\cdot)$ is the ℓ_2-distance (Euclidean distance) metric, then the response containing all the keywords of the query is a circle of minimum diameter.
- If $dist(\cdot,\cdot)$ is the ℓ_∞-distance metric, then the response containing all the keywords of the query is a minimum bounding square.

In the example (with ℓ_2-distance as the distance metric,) the diameter for the three keywords is precisely the diameter of the circle drawn in Fig. 1. Users can specify their respective mCK queries according to their requirements.

A spatial tuple can be associated with one or multiple keywords. Therefore, the number of response tuples for the mCK query is at most m. To facilitate our statement of the problem, we make the simple assumption that each tuple is associated with only one keyword[2], although our algorithm can be naturally extended without any modification to work efficiently in the case of multiple keywords. The mCK query returns m tuples matching the query keywords: each tuple in the result corresponds to a unique query keyword. Finding m closest keywords is essentially finding m tuples of minimum

[1]This is relevant in an application scenario like the MarcoPolo project (http://langg.com.cn) where blogs need to be geotagged.

[2]Tuples with multiple keywords can be treated as multiple tuples, each with a single keyword and located in the same position.

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.77

diameter matching these keywords. The problem can be formally defined as follows.

Definition 2 (mCK Query Problem): Given a spatial database with d-dimensional tuples represented in the form $(l_1, l_2, \ldots, l_d, w)$ and a set of m query keywords $Q = \{w_{q_1}, w_{q_2}, \ldots, w_{q_m}\}$, the mCK Query Problem is to find m tuples $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$, $T_i.w \in Q$ and $T_i.w \neq T_j.w$ if $i \neq j$, and $diam(\mathcal{T})$ is minimum.

While our initial example involves only three keywords, a search by document is likely to involve many keywords i.e. the value of m is likely to be large. This will give problem to a naive mCK query processing approach which is to exhaustively examine all possible sets of m tuples of objects matching the query keywords. By building m inverted lists for each of the m keywords with each list having only spatial objects that contain the corresponding keyword, the exhaustive algorithm can be implemented in a multiple nested loop fashion. If the number of objects matching keyword i is $D(i)$, then the number of sets of m tuples to be examined is $\prod_{i=1}^{m} D(i)$. This is prohibitively expensive when the number of objects and/or m is large.

Spatial data is almost always indexed to facilitate fast retrieval. We can adopt the idea of Papadias et al. [21] to answer the mCK query. Given N R*-trees, one for each keyword, candidate spatial windows for the mCK query result can be identified by executing multiway spatial joins (MWSJ) among the R*-trees. The join condition here becomes "closest in space" instead of "overlapping in space" [21]. When m is very small, this approach accesses only a small portion of the data and returns the result relatively quickly. However, as m increases, this approach suffers from two serious drawbacks. First, it incurs high disk I/O cost for identifying the candidate windows (due to synchronous multiway traversal of R*-trees) since it does not inherently support effective summarization of keyword locations. Second, it may not be able to identify a "tight" set of candidate windows since it determines candidate windows in an approximate manner based on the leaf-node MBRs of R*-trees without considering the actual objects. To process mCK queries in a more scalable manner, we propose to use one R*-tree to index all the spatial objects as well as their keywords. Integrating all the information in a single R*-tree provides more opportunities for efficient search and pruning.

The main contributions of this work are as follows.

- We propose a novel spatio-keyword query, called the *mCK query*, which has a large number of diverse and important applications in spatial databases.
- We propose a new index, called *bR*-tree*, for query processing. The bR*-tree extends the R*-tree to effectively summarize keywords and their spatial information.
- We incorporate efficient a priori-based search strategies, which significantly reduce the combinatorial search space.
- We define two monotone constraints, namely the distance mutex and keyword mutex, as the a priori properties for pruning. We also provide low-cost implementations for the examination of these constraints.
- We conduct extensive experiments to demonstrate that our algorithm is not only effective in reducing mCK query response time, but also exhibits good scalability in terms of the number of query keywords.

The remainder of this paper is organized as follows. Section II discusses existing work. Section III introduces bR*-tree. Section IV proposes a priori-based mCK query processing strategies and two monotone constraints used as a priori properties to facilitate pruning. Efficient implementations for the examination of these two constraints are also provided. Section V reports our performance study. Finally, we conclude our paper in Section VI.

II. Related Work

Various spatial queries using R-tree [11] and R*-tree [5] have been extensively studied. Besides the popular nearest neighbor query [23] and range query [18], closest-pair queries for spatial data using R-trees have also been investigated [13], [9], [25]. Nonincremental recursive and iterative branch-and-bound algorithms for k-closest pairs queries have been discussed by Corral et al. [9]. An incremental algorithm based on priority queues for the distance join query has been discussed by Hjaltason and Samet [13]. The work of Shin et al. [25] uses adaptive multistage and plane-sweep techniques for the K-distance join query and incremental distance join query. Studies have also been done on extending R-tree to strings [15]. Our problem can be seen as extending the R-tree to handle mixed types; our query being a set of keywords to be matched by combining the keyword sets of spatial objects that are close to each other.

MWSJ queries have been widely researched [21], [17], [19]. Given N R*-trees, one per keyword, the MWSJ technique of Papadias et al. [21] (later extended by Mamoulis and Papadias [17]) draws upon the synchronous R*-tree traversal (SRT) approach [7] and the window reduction (WR) approach [20]. Given two R*-tree-indexed relations, SRT performs two-way spatial join via synchronous traversal of the two R*-trees based on the property that if two intermediate R*-tree nodes do not satisfy the spatial join predicate, then the MBRs below them will not satisfy the spatial join predicate also. WR uses window queries to identify spatial regions which may contribute to MWSJ results. Local and evolutionary search are used by Papadias and Arkoumanis [19] to process MWSJ queries.

The work of Aref and Samet [4] discusses window-based query processing using a variant of the pyramid data structure. The proposal by Aref et al. [3] addresses retrieval of objects that are related by a distance metric (i.e., proximity queries), but it does not consider the "closest" criteria. Papadias et al. [22] examines the problem of finding a nearest neighbour that is spatially close to the center of a group of points. Unlike our work, the points there are not associated with any keywords. Moreover, their queries specify a set of spatial locations, while our queries specify keywords with no specific spatial location.

Various studies have also been done on finding association rules and co-location patterns in spatial databases [16], [24], [29], the aim being to find objects that frequently occur near to each other. Objects are judged to be near to each other if they are within a specified threshold distance of each other. Our study here is a useful alternative which foregoes the distance threshold, but instead allows users to verify their hypothesis through spatial discovery.

Recently, queries on spatial objects which are associated with textual information represented by a set of keywords, have received significant attention. Different spatial keyword queries on spatial databases have been proposed [12], [10]. Hariharan et al. [12] introduced a type of query combining range query and keyword search. The objects returned are required to intersect with the query MBR and contain all the user-specified keywords. A hybrid index of R*-tree and inverted index, called the KR*-tree, is used for query processing. Felipe et al. [10] proposed another similar query combining k-NN query and keyword search, and uses a hybrid index of R-tree and signature file, called the IR^2. Our mCK query differs from these two queries. First, our query specifies keywords with no specific location. Second, all the user-specified keywords do not necessarily appear in one result tuple. They can appear in multiple tuples as long as the tuples are closest in space.

III. bR*-TREE: R*-TREE WITH BITMAPS AND KEYWORD MBRS

To process mCK queries in a more scalable manner, we propose to use one R*-tree to index all the spatial objects and their keywords. In this section, we discuss the proposed index structure called the *bR*-tree*.

The bR*-tree is an extension of the R*-tree. Besides the node MBR, each node is augmented with additional information. A straightforward extension is to summarize the keywords in the node. With this information, it becomes easy to decide whether m query keywords can be found in this node. If there are N keywords in the database, the keywords for each node can be represented using a bitmap of size N, with a "1" indicating its existence in the node and a "0" otherwise. For example, a bitmap $B = 01001$ reveals that there are five keywords in the database and the current node can only be associated with the keywords in the second and fifth positions of the bitmap. This representation incurs little storage overhead. Moreover, it can accelerate the checking process of keyword constraints due to the relatively high speed of binary operations. Given a query $Q = 00110$, if we have B AND $Q = 0$, it implies that the given node does not have any query keywords and thus, this node can be eliminated from the search space.

Besides the keyword bitmap, we also store the *keyword MBR* in the node to set up more powerful pruning rules. The keyword MBR of keyword w_i is the MBR for all the objects in the nodes that are associated with w_i. It summarizes the spatial locations of w_i in the node. Using this information, we know the approximate area in the node which each keyword is distributed. If M is the node MBR and M_i is the keyword MBR for w_i, we have $M_i \subseteq M$.

When N is a large number, the cost for storing the keyword MBR is very high. For example, suppose there are a total of 100 keywords in the database and the objects are three-dimensional data. Spatial coordinates are usually stored in double precision, which occupies eight bytes per coordinate. It would therefore take $100 \times 3 \times 8 \times 2 = 4800$ bytes to store the keyword MBRs in one node. To reduce the storage cost, we split the node MBR into segments along each dimension. Each keyword MBR is represented approximately by the start and end offsets of the segments along each dimension. The range of an offset that occupies n bits is $[0, 2^n - 1]$. In our implementation, we set $n = 8$ (resulting in 256 segments) and found that it provided satisfactory approximation.

After being augmented with the bitmap and keyword MBR, non-leaf nodes of the bR*-tree contain entries of the form (*ptrs, mbr, bmp, kwd_mbr*), where

- *ptrs* are pointers to child nodes;
- *mbr* is the node MBR that covers all the MBRs in the child nodes;
- *bmp* is a keyword bitmap, each bit of which corresponds to a specific keyword, and is marked as "1" if the MBR of the node contains the keyword and "0" otherwise;
- *kwd_mbr* is the vector of keyword MBR for all the keywords contained in the node.

Fig. 2 depicts an example of an internal node containing three keywords w_1, w_2, w_3 represented as 111. It also maintains the keyword MBRs of w_1, w_2 and w_3. The keyword MBR of w_i is a spatial bound of all the objects with keyword w_i. Leaf nodes contain entries of the form (*oid, loc, bmp*), where

- *oid* is a pointer to an object in the database;
- *loc* represents the coordinates of the object;
- *bmp* is the keyword bitmap.

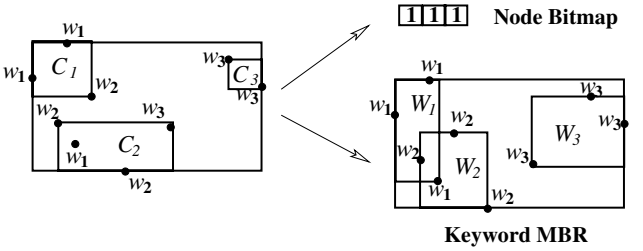


Fig. 2. Node information of the bR*-tree

In R*-tree, insertion works as follows: new tuples are added to leaves, overflowing nodes are split and the changes are propagated upward in the tree. The propagation process is called AdjustTree and the parent node is updated based on the property that its MBR is tightly bound to the MBRs of its child. The bitmap and keyword MBR also have similar properties for convenient information update in the parent node. The set of keywords of the parent node is the union of the sets of keywords in the child nodes. If w_i appears in a child node, it must also appear in the parent node. On the other

hand, the keyword MBR of w_i in the parent node is actually the minimum bound of the corresponding keyword MBRs in the child nodes. If the parent node's MBR does not tightly enclose all its child MBRs, or its keywords or keyword MBRs are not consistent with those in the child nodes, AdjustTree is invoked. Hence, we can construct our bR*-tree by means of the original R*-tree algorithm [5] by adding the operations of updating keywords and keyword MBR when AdjustTree is invoked. In a similar vein, the operations of update and delete in bR*-tree can also be naturally extended from the original implementations.

IV. Search Algorithms

Suppose a hierarchical bR*-tree has been built on all the data objects. The mCK query aims at finding m closest keywords in the leaf entries matching the query keywords. Our search algorithm starts from the root node. The target keywords may be located within one child node or across multiple child nodes of the root. Hence, we need to check all possible subsets of the child nodes. The candidate search space consists of two parts:

- the space within one child node;
- the space across multiple (> 1) child nodes.

If a child node contains all the m query keywords, we treat it as a candidate search space. Similarly, if multiple child nodes together can contribute all the query keywords and they are close to each other, then they are also included in the search space.

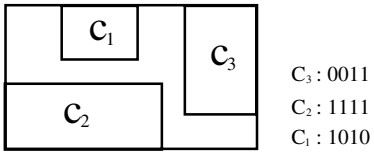


Fig. 3. An illustration of search in one node

To give an intuition of how the search space looks like, let us look at Fig. 3. The node has three child nodes C_1, C_2, and C_3, and they are close to each other. C_1 is associated with w_2 and w_4, C_2 with all the keywords, and C_3 with w_1 and w_2. Their bitmap representations are as shown in the figure. If the query is 1111, our candidate search space includes the subsets $\{C_2\}$, $\{C_1, C_2\}$, $\{C_2, C_3\}$ and $\{C_1, C_2, C_3\}$. The target keywords may be located in these nodes. $\{C_1\}$, $\{C_3\}$ and $\{C_1, C_3\}$ are pruned because they lack certain query keywords.

After exploring the root node, we obtain a list of candidate subsets of its child nodes. In order to find the m closest keywords located at the leaf entries, we need to further explore these candidates and traverse down the bR*-tree. For example, C_2 will be processed in a similar manner to the root node. Subsets of child nodes of C_2 are checked and all those that may possibly contribute a closer result is preserved. The search space for multiple nodes, such as $\{C_1, C_2\}$, is also turned into combinations of subsets of their child nodes. Each combination consists of child nodes from both C_1 and C_2. We can consider this process as node set $\{C_1, C_2\}$ being replaced by subsets of their child nodes and spawn a larger number of new node sets. The number of nodes in the new node set is nondecreasing and their nodes are one level lower in the bR*-tree. If we meet a set of leaf nodes, we retrieve all the combinations of m tuples from the leaf entries and calculate the closest m keywords that match the query keywords to see if a closer result can be found. Note that during the whole search process, the number of nodes in a node set will never exceed m because our target m tuples can only reside in at most m child nodes. This provides an additional constraint to reduce the search space.

Algorithms 1 and 2 summarize our approach for finding m closest keywords. The first step is to find a relatively small diameter for branch-and-bound pruning before we start the search. We start from the root node and choose a child node with the smallest MBR that contains all the query keywords and traverse down that node. The process is repeated until we reach the leaf level or until we are unable to find any child node with all the query keywords. Then we perform exhaustive search within the node we found and use the diameter of the result as our initial diameter for searching. Our experiments show that we can find a result of relatively small diameter in a very short time in this manner. We shall henceforth use δ^* to denote the smallest diameter of a result that has been found so far.

With this initial δ^*, we start our search from the root node. Since we are dealing with search in one node or multiple nodes, for the sake of uniformity, we use $NodeSet$ to denote a set of nodes as candidate search space, regardless of the number of nodes in it. The function `SubsetSearch` traverses the tree in a depth-first manner so as to visit the data objects in leaf entries as soon as possible. This increases the chance of finding a small δ^* at an early stage for better pruning. If $NodeSet$ contains leaf nodes, we retrieve all the objects in the leaf entries and exhaustively search for the closest keywords. Otherwise, we apply search strategies according to the number of nodes contained in $NodeSet$. In the following subsection, we discuss these strategies.

Algorithm 1 — Finding m Closest Keywords

Input: m query keywords, bR*-tree
Output: Distance of m closest keywords
1. Find an initial δ^*
2. return `SubsetSearch`($root$)

A. Searching In One Node

When searching in one node, our task is to enumerate all the subsets of its child nodes in which it is possible to find m closer tuples matching the query keywords. The subsets which contain all the m keywords and whose child nodes are close to each other are considered as candidates. There is also a constraint that the number of nodes in a subset should not exceed m. Therefore, the number of candidate subsets that may get further explored could reach $\sum_{i=1}^{m} \binom{n}{i}$ for a node with n child nodes.

Algorithm 2 — `SubsetSearch`: Searching in a Subset of Nodes

Input: current subset $curSet$
Output: Distance of m closest keywords

```
1.  if curSet contains leaf nodes then
2.    δ = ExhaustiveSearch(curSet)
3.    if δ < δ* then
4.      δ* = δ
5.  else
6.    if curSet has only one node then
7.      setList = SearchInOneNode(curSet)
8.      for each S ∈ setList do
9.        δ* = SubsetSearch(S)
10.   if curSet has multiple nodes then
11.     setList = SearchInMultiNodes(curSet)
12.     for each S ∈ setList do
13.       δ* = SubsetSearch(S)
```

An effective strategy for reducing the number of candidate subsets is of paramount importance as each subset will later spawn an exponential number of new subsets. Incidentally, the a priori algorithm of Agrawal and Srikant [1] has been an influential algorithm for reducing search space for combinatorial problems. It was designed for finding frequent itemsets using candidate generation via a lattice structure and has the following advantages:

1) Each candidate itemset is generated once because the way of generating new candidates is fixed and ordered. The k-itemset is joined by two $(k-1)$-itemsets with the same $(k-2)$-length prefix. Therefore, given a candidate itemset, such as $\{a, b, c, d\}$, we can infer that it is joined by $\{a, b, c\}$ and $\{a, b, d\}$.
2) For a k-itemset, we only need to check whether all its $(k-1)$-itemset subsets are frequent in level $k-1$. The cost is $O(n)$. This is due to the a priori property that all nonempty subsets of a frequent itemset must also be frequent. It is not necessary to check all its subsets at lower levels, the cost of which would be exponential.

In order to take advantage of the a priori algorithm, we define two monotonic constraints called **distance mutex** and **keyword mutex**. If a node set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$ is distance mutex or keyword mutex, then any superset of $\mathcal{N}$ is also distance mutex or keyword mutex and can be pruned.

Definition 3 (Distance Mutex): A node set $\mathcal{N}$ is distance mutex if there exist two nodes $N, N' \in \mathcal{N}$ such that $dist(N, N') > \delta^*$.

The definition of distance mutex is based on the observation that if the minimum distance between two node MBRs of N and N' is larger than δ^*, then the node set $\{N, N'\}$ does not give a result with diameter better than δ^*. This is obvious because the distance between any two tuples from N and N' must be larger than δ^*. Hence, we have the following lemma.

Lemma 4.1: If a node set $\mathcal{N}$ is distance mutex, then it can be pruned.

Proof: If $\mathcal{N}$ is distance mutex, then there exist two nodes $N, N' \in \mathcal{N}$ with $dist(N, N') > \delta^*$. For any m tuples $T_1, T_2, \ldots, T_m$ found in this node set that match the m query keywords , we can find at least one T_u from N and T_v from N' because each node has to contribute at least one tuple for the result. Since the distance between T_u and T_v must be larger than δ^*, any candidate set of m tuples has diameter larger than δ^*. ■

Lemma 4.2: Distance mutex is a monotone property.

Proof: Suppose $\mathcal{N}$ is distance mutex. Then there exist two nodes $N, N' \in \mathcal{N}$ with $dist(N, N') > \delta^*$. Any superset of $\mathcal{N}$ must also contain N and N' and hence must have diameter exceeding δ^*. ■

If all the nodes in node set $\mathcal{N}$ are close to each other, we can still take advantage of the stored keyword MBR for pruning. Here, we consider the problem from the perspective of contribution of keywords. Each node in the set must contribute a distinct subset of query keywords and all the contributed keywords constitute a complete set of query keywords. For example, given a set of two nodes N and N' and a query of three keywords 0111, if the closest keywords exist in this set, there are six cases of different contributions of query keywords by N and N'. N contributes one of the query keywords and N' contributes the other two. This generates three cases: $(w_1, w_2w_3), (w_2, w_1w_3), (w_3, w_1w_2)$. If N contributes two and N' contributes one, there are another three cases: $(w_1w_2, w_3), (w_1w_3, w_2), (w_2w_3, w_1)$. If the distance of any two different keywords (w_i, w_j) is larger than δ^*, where w_i is from N and w_j is from N', then the diameters of the six cases above are all larger than δ^*. We say that the node set is keyword mutex. The distance of (w_i, w_j) can be measured by the minimum distance of the two corresponding keyword MBRs. More generally, the concept of keyword mutex is defined as follows:

Definition 4 (Keyword Mutex): Given a node set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$, for any n different query keywords $(w_{q_1}, w_{q_2}, \ldots, w_{q_n})$ in which w_{q_i} is uniquely contributed by node N_i, there always exist two different keywords w_{q_i} and w_{q_j} such that $dist(w_{q_i}, w_{q_j}) > \delta^*$, then $\mathcal{N}$ is called keyword mutex.

Keyword mutex has properties similar to distance mutex.

Lemma 4.3: If a node set $\{N_1, N_2, \ldots, N_n\}$ is keyword mutex, then it can be pruned.

Proof: For any candidate of m tuples $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$ matching the query keywords, we want to prove $diam(\mathcal{T}) > \delta^*$. Since each node is required to contribute at least one tuple and $m \geq n$, we can extract n different keywords $\{w_{s_1}, w_{s_2}, \ldots, w_{s_n}\}$, each w_{s_j} coming from node N_j. According to our definition of keyword mutex, there exist two keywords w_{s_i} and w_{s_j} whose distance is larger

than δ^*. Two tuples T_u and T_v in candidate $\mathcal{T}$, associated with w_{s_i} and w_{s_j} respectively, can be found to be located within the two corresponding keyword MBRs with distance larger than δ^*. Therefore, $diam(\mathcal{T}) > \delta^*$ and the node set can be pruned. ■

Lemma 4.4: Keyword mutex is a monotone property.

Proof: Suppose $\mathcal{N}$ is keyword mutex and $\mathcal{N}'$ is its superset with t nodes. For any t different keywords $\{w_{s_1}, w_{s_2}, \ldots, w_{s_t}\}$ where w_{s_i} is contributed by node N_i, we can find two keywords w_{s_j} and w_{s_k} from nodes N_j and $N_k (N_j, N_k \in \mathcal{N})$, such that $dist(w_{s_j}, w_{s_k}) > \delta^*$. Hence $\mathcal{N}'$ is also keyword mutex.

■

Algorithm 3 — SearchInOneNode: Searching in One Node

Input: A node N in bR*-tree
Output: A list of new NodeSets

```
1.  L_1 = all the child nodes in N
2.  for i from 2 to m do
3.    for each NodeSet C_1 ∈ L_{i-1} do
4.      for each NodeSet C_2 ∈ L_{i-1} do
5.        if C_1 and C_2 share the first i − 1 nodes then
6.          C = NodeSet(C_1, C_2)
7.          if C has subset not appear in L_{i-1} then
8.            continue
9.          if C is not distance mutex then
10.           if C is not keyword mutex then
11.             L_i = L_i ∪ C
12. for each NodeSet S ∈ ∪_{i=1}^{m} L_i do
13.   if S contains all the query keywords then
14.     add S to cList
15. return cList
```

The method for searching in one node is shown in Algorithm 3. First (in line 1), we put all the child nodes in the bottom level of the lattice. The lattice is built level by level with increasing number of child nodes in the $NodeSet$. In level i, each $NodeSet$ contains exactly i child nodes. For a query with m keywords, we only need to check $NodeSet$ with at most m nodes, leading to a lattice with at most m levels. Lines 5–6 show two sets C_1 and C_2 in level $i-1$ being joined, they must have $i-2$ nodes in common. Lines 7–14 check if any of its subsets in level $i-1$ is pruned due to distance mutex or keyword mutex. If all the subsets are legal, we check whether this new candidate itself is distance mutex or keyword mutex for pruning. If it is not pruned, we add it to level i. In lines 19–22, after all the candidates have been generated, we check each one to see if it contains all the query keywords. Those missing any keywords are eliminated. We do not check this constraint while building the lattice because if a node does not contain all the query keywords, it can still combine with other nodes to cover the missing keywords. As long as it is neither distance mutex nor keyword mutex, we keep it in the lattice.

B. Searching In Multiple Nodes

Given a node set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$, the search in $\mathcal{N}$ needs to check all the possible combinations of child nodes from each N_i to explore the search space in the lower level. The number of child nodes in the newly derived sets should not exceed m. For example, given a node set $\{A, B, C\}$ where $A = \{A_1, A_2\}$, $B = \{B_1, B_2\}$ and $C = \{C_1\}$. A_i, B_i and C_i are child nodes in A, B, and C, respectively. Assume all the pair distances of child nodes are less than δ^*. All the candidate combinations of child nodes are shown in Fig. 4. Every new node set contains child nodes from all the three nodes. If $m = 3$, the candidates are those in the first column. Each query keyword is contributed by exactly one of the child nodes. If $m = 5$, the search space includes all the node sets listed in the figure.

3 nodes	4 nodes	5 nodes
$A_1B_1C_1$	$A_1A_2B_1C_1$	$A_1A_2B_1B_2C_1$
$A_1B_2C_1$	$A_1A_2B_2C_1$	
$A_2B_1C_1$	$A_1B_1B_2C_1$	
$A_2B_2C_1$	$A_2B_1B_2C_1$	

Fig. 4. Possible sets of $\{A_1, A_2\}$, $\{B_1, B_2\}$, and $\{C_1\}$

The a priori algorithm can still be applied to this situation. Fig. 5 shows the lattice to generate candidates for the above node set $\{A, B, C\}$. The sets with child nodes from all three nodes are marked with bold lines. The nonbold nodes cannot be candidates. Given m query keywords, only the bottom m levels of the lattice is built. The properties of distance mutex and keyword mutex are also applicable during generation of the new candidates. The algorithm returns those candidates in the bold nodes, which are neither distance mutex nor keyword mutex. However, this approach creates many unnecessary candidates and incurs additional cost in checking these candidates. For example, if $m = 3$, we know from Fig. 4 that there are only four candidate sets that need to be generated. But the a priori algorithm will create a whole level for candidates with three nodes, thereby resulting in ten candidates.

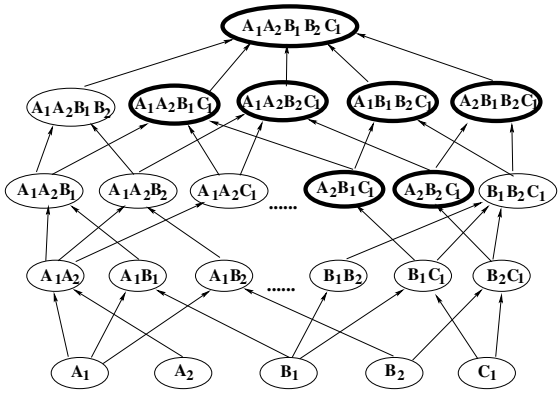


Fig. 5. a priori algorithm applied to search in multiple nodes

Alternatively, we propose a new algorithm which does not generate any unnecessary candidates, but still keeps the

advantages of the a priori algorithm. For a node set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$, we reuse the n lists of candidate node sets generated by applying the a priori algorithm to search in each node. The i^{th} list contains the sets of child nodes in N_i. The sets are ordered from lower levels in the lattice to higher levels. For example, if N_i has three child nodes $\{C_1, C_2, C_3\}$, the sets of child nodes in the corresponding list may be ordered in the following way: $\{C_1\}, \{C_2\}, \ldots, \{C_1, C_2, C_3\}$. An initial filtering is done on N_i's list by only considering the child nodes that are close to all the other N_j. If C_k in N_i is far away from any other node N_j, all the sets in the i^{th} list containing C_k is pruned.

To generate new candidates, we enumerate all the possible combinations of child node subsets from these n lists. Fig. 6 illustrates our approach. At the bottom level, we have three lists of child node subsets from nodes A, B, and C. Combinations of the subsets from these three lists are enumerated to retrieve new candidate sets. As shown in Fig. 6, all the nine candidate sets are directly retrieved from the subsets in the bottom level. In this manner, our algorithm does not generate unnecessary candidates. Moreover, the enumeration process is ordered, as shown by the dashed arrows. A new candidate is enumerated only after all of its subsets have been generated. For example, $A_1A_2B_1C_1$ must be generated after $A_1B_1C_1$ and $A_2B_1C_1$ because the subsets of child nodes in each list are ordered by the node number. As a consequence, we can efficiently generate the candidates and still preserve the advantages of the a priori algorithm:

1) Each candidate item is generated once. For example, given a candidate item $\{A_1A_2B_1C_1\}$, we know that it is combined by $\{A_1, A_2\}$, $\{B_1\}$ and $\{C_1\}$. No duplicate candidates will appear in the results.
2) For a k-item, we only need to check its $(k-1)$-item subsets. Since the candidates in each N_i generated by the a priori algorithm are ordered, all its subsets must have been examined when we are processing the current k-item.

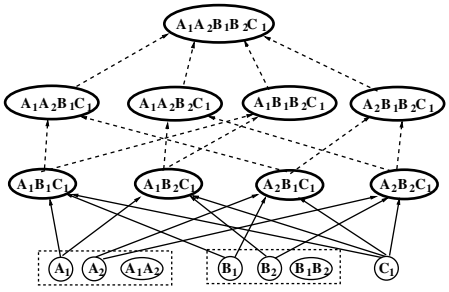


Fig. 6. Extended a priori algorithm

Algorithm 4 shows how a set of n nodes $\{N_1, \ldots, N_n\}$ is explored. First, n lists of ordered subsets of child nodes are obtained. Then Algorithm 5 is invoked to enumerate all the candidate sets. It is implemented in a recursive manner. Each time an enumerated candidate is generated, we check if it contains all the query keywords to decide whether to prune it or to put it in the candidate list(see Lines 1–4). Lines 5–8 indicate the beginning of the recursion process. It starts from each child node subset in list L_n and makes it as our current partial node set $curSet$. $curSet$ recursively combines with other child node subsets until it finally contains child nodes from $\{N_1, \ldots, N_n\}$. In each recursion, we iterate the child node sets in list L_i to combine with $curSet$ and generate a new set denoted as $newSet$. Lines 12–13 show that if $newSet$ already has more than m child nodes, we stop the iteration because the list is ordered. The child node subsets which are not checked could only have more child nodes and will result in even more nodes in $newSet$. Otherwise, we check if any subsets of $newSet$ have been pruned due to distance mutex or keyword mutex. If not, we go on checking whether this new $NodeSet$ itself is distance mutex or keyword mutex. All these checking processes are shown in Lines 14–17. If $newSet$ is not pruned, we set it as $curSet$ and continue the recursion. Finally, the algorithm returns all the candidates that were not pruned away. In the following subsections, we propose two novel methods to efficiently check whether a set is distance mutex or keyword mutex.

Algorithm 4 — SearchInMultiNodes: Search In Multiple Nodes

Input: A set of $\{N_1, \ldots, N_n\}$ in bR*-tree
Output: A list of new NodeSets
1. **for** each node N_i **do**
2. L_i = SearchInOneNode(N_i)
3. perform an initial filtering on L_i
4. return Enumerate($L_1, \ldots, L_n$, n, NULL)

C. Pruning via Distance Mutex

The diameter of a candidate of m tuples matching the query keywords is determined by the maximum distance between any two tuples. The candidate can be discarded if we found two tuples in it with distance larger than δ^*. Similarly, as we are traversing down the tree, we can eliminate the node sets in which the minimum distance between two nodes is larger than δ^*. A candidate which is not distance mutex requires each pair of nodes to be close. It takes $O(n^2)$ time to check the distance between all pairs of a set of n nodes.

To facilitate more efficient checking, we introduce a concept called **active MBR**. Fig. 7(a) illustrates this concept with a set of two nodes $\{N_1, N_2\}$. First, we enlarge these two MBRs by a distance of δ^*, and their intersection is marked by the shaded area M in the figure. We can restrict our search area within area M because any tuple outside M cannot possibly combine with tuples of the other node to achieve a smaller diameter than δ^*. In this example, the child node C_1 does not participate because it does not intersect with M. The objects in C_2 but outside M need not be taken into account as well. We call M the active MBR of N_1 and N_2 because a candidate of m tuples can only reside within the area covered by M. However, we should also check for false intersections, which is shown in Fig. 7(b). The intersection actually lies outside both

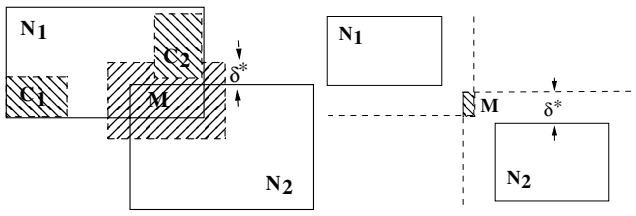
Algorithm 5 — Enumerate: Enumerate All Possible Candidates

Input: n lists of sets of child nodes $L_1, \ldots, L_n$, count and $curSet$

Output: A list of new NodeSets

```
1. if count = 0 then
2.    if curSet contains all the query keywords then
3.       push curSet into the candidate list cList
4.       return
5. if count = n then
6.    for each NodeSet S ∈ L_n do
7.       curSet = S
8.       Enumerate(L_1, ..., L_n, count−1, curSet)
9. else
10.   for each NodeSet S ∈ L_n do
11.      newSet = NodeSet(curSet, S)
12.      if newSet contains more than m nodes then
13.         break
14.      if newSet has any illegal subset candidate then
15.         continue
16.      if newSet is not distance mutex then
17.         if newSet is not keyword mutex then
18.            Enumerate(L_1, ..., L_n, count−1, newSet)
19. return cList
```

N_1 and N_2. If this happens, the set does not have an active MBR and becomes distance mutex. Hence, we can prune it away.



(a) True intersection (b) False intersection

Fig. 7. Example of active MBR

When a third node N_3 combines with N_1 and N_2, we only need to check whether N_3 intersects with M, without having to calculate the distance from N_3 to N_2 and N_1. Any tuple outside M is either far away from N_1 or far away from N_2. Therefore, if N_3 does not intersect with M, we can conclude that the set $\{N_1, N_2, N_3\}$ is distance mutex. Otherwise, we update the active MBR for this new set to be its intersection with the enlarged N_3. This property greatly facilitates the checking of distance mutex. When we are checking a new candidate "joined" by two sets C_1 and C_2 in the a priori algorithm, we only need to check whether the active MBR of C_1 intersects with that of C_2. Moreover, as more nodes participate in the set, the active MBR becomes smaller and smaller, and is likely to be pruned. This helps to reduce the cost of search by avoiding the enumeration of large number of nodes.

D. Pruning via Keyword Mutex

A set of n nodes is said to be keyword mutex if for any n different keywords, each from one node, we can always find two keywords whose distance is larger than δ^*. We use the keyword MBR stored in each node to check for keyword mutex. We present a simple example by considering a set of two nodes $\{A, B\}$. Given four query keywords, we construct a 4×4 matrix $\mathsf{M}(A, B) = (m)_{ij}$ to describe the keyword relationship between A and B: m_{ij} indicates whether tuples with keyword w_i in A can be combined with tuples with keyword w_j in B. If the minimum distance between these two keyword MBRs is smaller than δ^*, then $m_{ij} = 1$; otherwise, $m_{ij} = 0$. If w_i does not appear in A, or w_j does not appear in B, then also $m_{ij} = 0$. Moreover, $m_{ii} = 0$ since each keyword in the mCK result can only be contributed by one node. If $\mathsf{M}(A, B)$ is the zero matrix, we can conclude that the set is keyword mutex. For any two different keywords w_i and w_j from A and B, its distance must be larger than δ^*.

Generally, for a set of $n \geq 3$ nodes $\{N_1, N_2, \ldots, N_n\}$, we define $\mathsf{M}(N_1, \ldots, N_n)$ recursively as follows: for $n \geq 3$,

$$\begin{aligned}\mathsf{M}(N_1, \ldots, N_n) =& (\mathsf{M}(N_1, N_2) \times \mathsf{M}(N_2, \ldots, N_n)) \otimes \\ & (\mathsf{M}(N_1, \ldots, N_{n-1}) \times \mathsf{M}(N_{n-1}, N_n)) \otimes \\ & \mathsf{M}(N_1, N_n),\end{aligned}$$

where $\times$ is the ordinary matrix multiplication, and $\otimes$ is elementwise multiplication. The base case when $n = 2$ has already been defined in the paragraph above.

As the lemma below shows, we need only check whether $\mathsf{M}(N_1, \ldots, N_n) = 0$ to determine if $\{N_1, \ldots, N_n\}$ is keyword mutex.

Lemma 4.5: If $\mathsf{M}(N_1, \ldots, N_n) = 0$, then the set of nodes $\{N_1, N_2, \ldots, N_n\}$ is keyword mutex.

Proof: Suppose $\mathsf{M}(N_1, \ldots, N_n) = 0$ but $\{N_1, \ldots, N_n\}$ is not keyword mutex. Then there must exist n different keywords $k_1, \ldots, k_n$ from nodes $N_1, \ldots, N_n$, respectively, such that all pairs of keywords are at distance less than δ^*. We have $\mathsf{M}(N_i, N_j)_{k_i k_j} = 1$ for $1 \leq i < j \leq n$. First, we prove

$$\mathsf{M}(N_u, \ldots, N_v)_{k_u k_v} \geq \prod_{u \leq i < j \leq v} \mathsf{M}(N_i, N_j)_{k_i k_j} \qquad (1)$$

by induction on $v - u$

When $v - u = 1$, (1) clearly holds. For $v - u > 1$, consider the inequalities:

$$\mathsf{M}(N_u, \ldots, N_{v-1})_{k_u k_{v-1}} \geq \prod_{u \leq i < j \leq v-1} \mathsf{M}(N_i, N_j)_{k_i k_j}$$

and

$$\mathsf{M}(N_{u+1}, \ldots, N_v)_{k_{u+1} k_v} \geq \prod_{u+1 \leq i < j \leq v} \mathsf{M}(N_i, N_j)_{k_i k_j},$$

which hold by the induction hypothesis. Since the matrix

entries are all nonnegative, we have

$$\begin{aligned}
&\mathsf{M}(N_u,\ldots,N_v)_{k_uk_v}\\
&\geq \left(\mathsf{M}(N_u,N_{u+1})_{k_uk_{u+1}}\cdot\left(\prod_{u+1\leq i<j\leq v}\mathsf{M}(N_i,N_j)_{k_ik_j}\right)\right)\cdot\\
&\left(\left(\prod_{u\leq i<j\leq v-1}\mathsf{M}(N_i,N_j)_{k_ik_j}\right)\cdot\mathsf{M}(N_{v-1}N_v)_{k_{v-1}k_v}\right)\\
&\cdot\mathsf{M}(N_u,N_v)_{k_uk_v}\\
&\geq \prod_{u\leq i<j\leq v-1}\mathsf{M}(N_i,N_j)_{k_ik_j}.
\end{aligned}$$

Therefore,

$$\mathsf{M}(N_1,\ldots,N_n)_{k_1k_n}\geq\prod_{1\leq i<j\leq n}\mathsf{M}(N_i,N_j)_{k_ik_j}=1,$$

which is nonzero. This is a contradiction. ■

The advantage of the matrix implementation is that it can be naturally integrated into our a priori-based search strategy. When dealing with set $\{N_1,\ldots,N_n\}$, the matrices involved in the above formula will already have materialized in most cases. Therefore, checking for keyword mutex requires only two matrix multiplications and two matrix elementwise products, which can be achieved at low cost.

V. Empirical Study

This section provides an extensive performance study of our query strategy using one bR*-tree to integrate all the spatial and keyword information. We use the MWSJ approach [21] as reference. If there are N keywords existing in the spatial database, N separate R*-trees are built. Given m query keywords, we pick m corresponding R*-trees $T_1, T_2, \ldots, T_m$. The trees are ordered by the number of objects in the tree. The search process starts from the smallest R*-tree T_1 with the fewest objects. For any leaf MBR M_1 in T_1, we search in T_2 the leaf MBRs that are close enough to M_1. The idea of active MBR can be applied to speed up the search. In T_3, the search space has been shrunk to the active MBR of M_1 and M_2. Only the MBRs intersecting with this active MBR will be taken into account. This process lasts until all the leaf MBRs near M_1 in the other R*-trees have been explored. Then, we move to other leaf MBRs in T_1 until all the combinations of objects in each R*-tree have been explored completely. We found that such an implementation outperforms the traditional top-down strategy used in answering closest-pair queries [9].

We implemented both algorithms in C++ using its standard template library. The bR*-tree is implemented by extending the R*-tree code from *http://research.att.com/~marioh/spatialindex/*. All the experiments are conducted on a server with Intel Xeon 2.6GHz CPU, 8GB memory, running Ubuntu 7.10. Both synthetic and real life data sets are used for performance testing. We use average response time (ART) as our performance metric: ART = $(1/N_Q)\sum_{i=1}^{N_Q}(T_f - T_i)$, where T_i is the time of query issuing, T_f is time of query completion, and N_Q is the total number of times the given mCK query was issued. Note that ART is equivalent to the elapsed time including disk I/O and CPU-time.

A. Experiments on Synthetic Data Sets

The synthetic data generator generates spatial data points in a random manner. Each point is randomly distributed in d-dimensional space $[0,1]^d$ and assigned with a fixed number of random keywords. We fix the number of keywords on each data point so that it is more convenient to analyze the performance when a data point is associated with multiple keywords. In our implementation of the bR*-tree, the page size is set as 4K bytes and the maximum number of entries in internal nodes is set at 30. However, the number of entries in leaf nodes is set to be the same with the total number of keywords to allow flexibility in handling different number of keywords. In the implementation of MWSJ, we also use a page size of 4K and set the maximum number of entries in all nodes at 30. The bR*-tree takes more time than MWSJ in building the index for two reasons. First, in MWSJ, each tuple is inserted into a small R*-tree with the same keyword as the tuple. In bR*-tree, each tuple is inserted into the whole tree. This results in much higher cost for each insertion, including choosing a leaf, invoking more split and AdjustTree operations. Second, bR*-tree maintains additional information, such as keywords and keyword MBR, which need to be updated during insertions.

In the following experiments, we adjust four parameters to generate different data sets. The parameters are

- TK, the total number of keywords in the database;
- DS, the data size;
- DM, the dimension;
- KD, the number of keywords associated with each data.

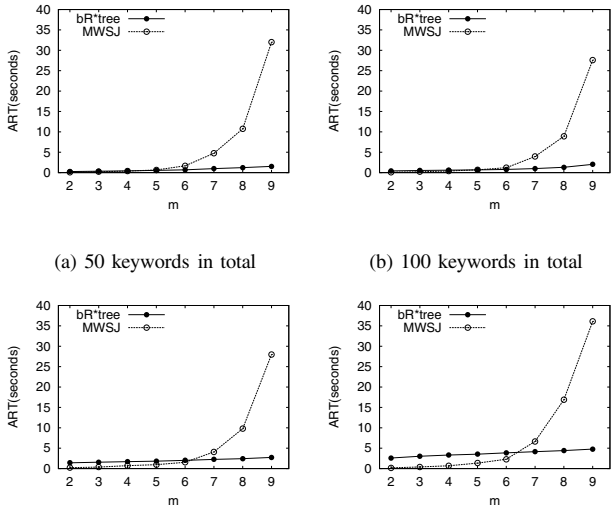
In each experiment, we compare the performance of bR*-tree with MWSJ on different synthetic data sets using ART as the performance metric.

1) **Effect of TK:** We ran the first experiment on four data sets to test scalability in terms of the number m of query keywords. We generated the data sets with total number of keywords 50, 100, 200 and 400, respectively. Each data is two-dimensional and associated with one keyword. In each data set, there are 3,000 data points associated with the same keyword.

Fig. 8 shows the ART of two algorithms with respect to the number of query keywords. When m is small, we can see that MWSJ outperforms the bR*-tree and this advantage becomes clearer as the total number of keywords increases. It only accesses m of the total N R*-trees that occupy a small portion of the whole data set. The query can be processed relatively quickly. However, our search process needs to access all the nodes in the entire bR*-tree because the data with different keywords are randomly distributed in the leaf nodes. This results in relatively poor performance as compared to that of MWSJ.

As m increases to large values, the performance of MWSJ starts to degrade dramatically. The search space is expanded exponentially and MWSJ incurs high disk I/O cost for identifying the candidate windows since it does not inherently support effective summarization of keyword locations. However, our algorithm demonstrates remarkable scalability as m increases[3]. The bR*-tree summarizes the keywords and their locations in each node, and this plays an important role in effectively pruning the search space. The a priori-based search strategy also restricts the candidate search space from growing too quickly.

Note that the overall performance trend of MWSJ across the four data sets is similar. The reason is that the data sets have the same number of data points associated with each keyword and the size of R*-tree is the same. However, the performance of bR*-tree degrades slightly with the increase of data size and the total number of keywords. It integrates all the data points in one tree, leading to higher access cost.



(a) 50 keywords in total (b) 100 keywords in total

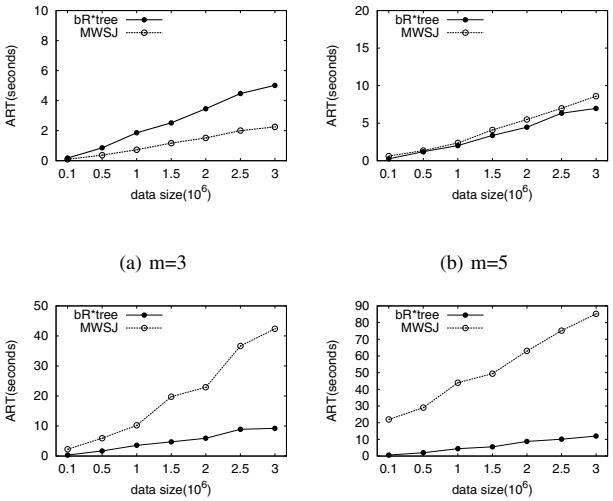
(c) 200 keywords in total (d) 400 keywords in total

Fig. 8. Performance on increasing TK

2) **Effect of DS:** In the above experiment, the number of data points associated with each keyword is fixed. In this experiment, we fix the total number of keywords at 100 and increase the data size from 100,000 to 3,000,000 to examine the performance of bR*-tree and MWSJ.

Fig. 9 shows how ART increases with the data size in answering the same number of query keywords. When m is small, e.g. $m = 3$ and $m = 5$, both algorithms demonstrate similar rate of increase in ART. The spatial index and the pruning using active MBR did take effect to suppress the expansion of search space caused by the increase of data size. However, when m becomes large, e.g. $m = 7$ and $m = 8$, a small amount of increase in the size of the R*-tree in MWSJ can lead to a remarkable increase in the search space. We can observe from Fig. 9 that MWSJ becomes sensitive to the increase of data size and the performance declines dramatically especially when the data size is large. In contrast, bR*-tree scales smoothly in a stable manner, thereby validating the effectiveness of our search strategy.

[3]Note that this make our algorithm particularly useful for purpose like geotagging of documents where a mCK query with large number of keywords are issued by an automatic search algorithm. In addition, for systems in which the number of keywords in a submitted query can varies greatly, our approach will provide very stable performance compared to MWSJ.



(a) m=3 (b) m=5

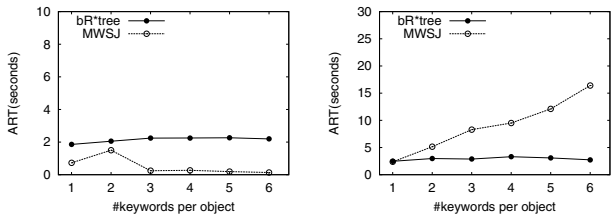
(c) m=7 (d) m=8

Fig. 9. Performance on increasing DS

3) **Effect of KD:** In many applications, a spatial object is associated with a set of keywords rather than only one keyword. Under a fixed data size, if we increase the number of keywords associated with each data point, the search space increases as well. For each keyword, there are more data points associated with it, and hence, a larger R*-tree is needed for indexing in the case of MWSJ. However, the size of bR*-tree is not affected because the bitmap in the node only gets more bits set, but still incurs a fixed storage cost.

In this experiment, we generate 1,000,000 two-dimensional data points. There are a total of 100 keywords in the data set. We increase the value of KD from one to six. The results, depicted in Fig. 10, show that bR*-tree always demonstrates good stability when KD increases. However, MWSJ suffers from serious performance degradation as KD increases because it does not inherently support effective summarization of keyword locations. Note that when $m = 3$, the performance of MWSJ has a sudden improvement when $KD = 3$, i.e. each object is associated with three keywords. An object with all three query keywords is very likely to be found in the data set giving $\delta^* = 0$. This greatly facilitates the pruning in the unexplored search space. When $m = 5$, this improvement is not shown clearly because the probability of finding an object with all the query keywords in the early search stage is low.

4) **Effect of DM:** In the above experiments, we only handle two-dimensional data. In some applications, the data may have multiple attributes and are mapped to higher-dimensional space. For example, a notebook may be mapped to a five-



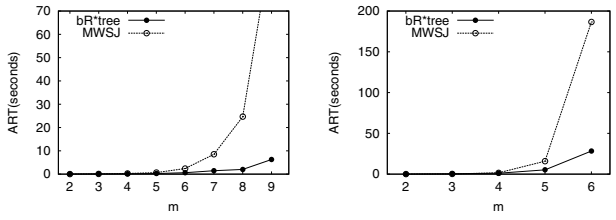
(a) m = 3 (b) m = 5

(c) m = 7 (d) m = 8

Fig. 10. Performance on increasing KD

dimensional value arising from attributes such as CPU, memory, hard disk, weight and price. The closest notebooks from different manufacturers may be serious competitors in the market. Therefore, it is meaningful to test how the algorithms perform on higher dimensional spaces.

We test the performance on three- and four-dimensional data with a small data size of 50,000. There are 100 keywords in total and each data is associated with one keyword. The ART results are shown in Fig. 11. It is clear that MWSJ performs poorly on higher dimensional data because its pruning is based on only the distance constraint. Our bR*-tree takes advantage of both distance and keyword constraints of the mCK query for pruning and shows much better scalability. As m increases, the performance of MWSJ rapidly declines and can be orders of magnitude worse than bR*-tree.



(a) dimension = 3 (b) dimension = 4

Fig. 11. Performance on increasing DM

B. Experiments on Real Data Set

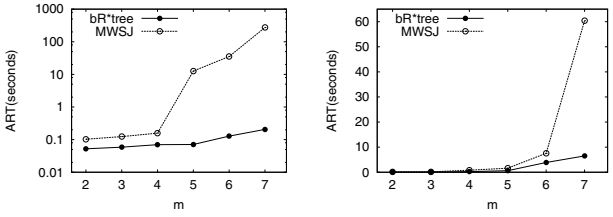
We use TIGER (Topologically Integrated Geographic Encoding and Referencing system) (downloadable from *http://www.census.gov/geo/www/tiger*) as our real data set. The database consists of numerous complicated geographic and cartographic information of the entire United States. Since we are concerned with point data in our mCK query, we simply extract the landmark data, which can be custodial facility (hospitals, orphanages, federal penitentiaries, etc.), educational, cultural or religious institutions, etc. Each point in the data set is associated with a census feature class code to identify its noticeable characteristic. For example, *D85* is the class code for keyword *Park*.

D1	4	D28	206	D43	1956	D71	75
D10	3	D29	1	D44	6092	D73	1
D20	23	D31	266	D51	364	D81	177
D21	872	D32	3	D53	2	D82	3291
D22	2	D33	34	D61	929	D83	21
D23	167	D35	6	D62	21	D84	1
D24	2	D36	17	D63	21	D85	295
D25	20	D37	5	D64	21	D90	78
D26	26	D41	2	D65	120		
D27	44	D42	2	D66	9		

Fig. 12. Keyword distribution on Texas data set

After cleaning and format transformation on the raw data, we extracted two data sets, Texas and California, with 15,179 and 13,863 data points, respectively. Both data sets have dozens of keywords. The distribution for each keyword is highly skewed. Fig. 12 shows the keyword distribution in Texas. Some landmark may get thousands of points while others may have only one data point. For example, *D43* represents educational institutions, including academy, school, college and university. These institutions are widely distributed and are well recorded in the raw data set. However, landmarks like water tower (*D71*) are a rarity and only one such landmark appears in our extracted data set.

In our experiments, we ignore infrequent keywords and submit queries with the most frequent keywords. Fig. 13 shows the ART with respect to the number of query keywords in both data sets. We can see that bR*-tree outperforms MWSJ even when m is small. The reason is that the number data associated with each keyword is highly skewed. When a query has frequent keywords, MWSJ loses the advantage of having to access only a small portion of the data set. When m increases to large values, its performance still degrades dramatically. Our bR*-tree not only answers the frequent keywords query in a shorter time, but also exhibits good scalability. Therefore, the bR*-tree performs significantly better than MWSJ in answering queries with frequent keywords.



(a) Texas (b) California

Fig. 13. Performance on two real data sets

VI. Conclusions and Future Work

In this paper, we take a step towards searching by document by addressing the mCK query. We use the bR*-tree to

effectively summarize keyword locations, thereby facilitating pruning. We propose effective a priori-based search strategies for mCK query processing. Two monotone constraints and their efficient implementations are also discussed. Our performance study on both synthetic and real data sets demonstrates that the proposed bR*-tree answers mCK queries efficiently within relatively short query response time. Furthermore, it demonstrates remarkable scalability in terms of the number of query keywords and significantly outperforms the existing MWSJ approach when m is large.

While handling large number of keywords is an important step towards searching by document, there is still much room for future research.

First, we plan to investigate the use of our method for graph based keyword search [6], [2], [28], [26] by embedding a graph into a multi-dimensional space so that path distance between nodes in the graph are still approximately captured. This can be done by applying the graph mapping method that was adopted in [27]. With such a mapping, keywords that are close on the graph will also be near to each other in the multi-dimensional space and thus our method can be use to process graph-based keyword search as well. Such an approach will however require the processing of spatial data with high dimensionality and provide new technical challenges.

Second, we plan to scale up our method to the total number of keywords by investigating various alternatives. A possible solution is to use a single bR*-tree for most frequent keywords and multiple R*-trees for infrequent keywords. Alternatively, the strategy, one for each group of highly correlated keywords within queries, is also feasible. A look at compressing the keyword bitmaps [8], [14] will be interesting as well.

Finally, given that the number of query keywords from a search by document operation may be large, it may not be possible to match all the keywords in a search result. A look at partial or fuzzy keyword search may be necessary to overcome this problem.

As a first piece of work on this topic, we believe there are possibly many other directions for future research as well.

ACKNOWLEDGMENT

The research of Dongxiang Zhang and Anthony K. H. Tung is supported in part by a grant from the Singapore National Research Foundation under the project entitled "Structure-Aware Data and Query Modeling for Effective Information Retrieval over Heterogeneous Data Sources in Co-Space".

The research of Y. M. Chee is supported in part by the National Research Foundation of Singapore under Research Grant NRF-CRP2-2007-03, and by the Nanyang Technological University under Research Grant M58110040.

REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proceedings of ICDE*, 2002.

[3] W. Aref, D. Barbara, S. Johnson, and S. Mehrotra. Efficient processing of proximity queries for large databases. *Proc. ICDE*, pages 147–154, 1995.

[4] W. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. *Proc. PODS*, pages 265–272, 1990.

[5] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proc. SIGMOD*, pages 322–331, 1990.

[6] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings of ICDE*, 2002.

[7] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. *Proc. SIGMOD*, pages 237–246, 1993.

[8] G. Cong, B. Ooi, K. Tan, and A. Tung. Go green: recycle and reuse frequent patterns. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 128–139.

[9] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. *Proc. SIGMOD*, pages 189–200, 2000.

[10] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. *In Proc. ICDE International Conference on Data Engineering*, 2008.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. SIGMOD*, pages 47–57, 1984.

[12] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In *SSDBM*, page 16, 2007.

[13] G. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. *Proc. SIGMOD*, pages 237–248, 1998.

[14] H. Jagadish, R. Ng, B. Ooi, and A. Tung. ItCompress: An Iterative Semantic Compression Algorithm. In *Proceedings of the 20th International Conference on Data Engineering (ICDE04)*, volume 1063, pages 20–00.

[15] H. V. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. *Proc. SIGMOD*, pages 403–414, 2000.

[16] K. Koperski and J. Han. Discovery of spatial association rules in geographic information databases. *Proc. SSD*, pages 47–66, 1995.

[17] N. Mamoulis and D. Papadias. Multiway spatial joins. *Proc. TODS*, 26(4):424–475, 2001.

[18] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *PODS*, pages 214–221, New York, NY, USA, 1993. ACM.

[19] D. Papadias and D. Arkoumanis. Approximate processing of multiway spatial joins in very large databases. *Proc. EDBT*, pages 179–196, 2002.

[20] D. Papadias, N. Mamoulis, and B. Delis. Algorithms for querying by spatial structure. *Proc. VLDB*, pages 546–557, 1998.

[21] D. Papadias, N. Mamoulis, and Y. Theodoridis. Processing and optimization of multiway spatial joins using R-trees. *Proc. PODS*, pages 44–55, 1999.

[22] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. *Proc. ICDE*, pages 301–312, 2004.

[23] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *Proc. SIGMOD*, pages 71–79, 1995.

[24] S. Shekhar and Y. Huang. Discovering spatial co-location patterns: A summary of results. *Proc. SSTD*, pages 236–256, 2001.

[25] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. *Proc. SIGMOD*, pages 343–354, 2000.

[26] Q. Vu, B. Ooi, D. Papadias, and A. Tung. A graph method for keyword-based selection of the top-K databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 915–926. ACM New York, NY, USA, 2008.

[27] N. Wang, S. Parthasarathy, K. Tan, and A. Tung. CSV: visualizing and mining cohesive subgraphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 445–458. ACM New York, NY, USA, 2008.

[28] B. Yu, G. Li, K. Sollins, and A. K. H. Tung. Effective keyword-based selection of relational databases. In *Proceedings of SIGMOD*, 2007.

[29] X. Zhang, N. Mamoulis, D. W. Cheung, and Y. Shou. Fast mining of spatial collocations. *Proc. KDD*, pages 384–393, 2004.