An Enhanced Technique for k-Nearest Neighbor Queries with Non-spatial Selection Predicates †

Dong-Joo Park Hyoung-Joo Kim

School of Computer Science and Engineering, Seoul National University
San 56-1, Shillim-dong, Kwanak-gu, Seoul 151-742, KOREA

{djpark, hjk}@oopsla.snu.ac.kr

Abstract

In multimedia databases, k-nearest neighbor queries are popular and frequently contain non-spatial predicates. Among the available techniques for such queries, the incremental nearest neighbor algorithm proposed by Hjaltason and Samet is known as the most useful algorithm[15]. The reason is that if $k' > k$ neighbors are needed, it can provide the next neighbor for the upper operator without restarting the query from scratch. However, the R-tree in their algorithm has no facility capable of partially pruning tuple candidates that will turn out not to satisfy the remaining predicates, leading their algorithm to inefficiency. In this paper, we propose an RS-tree-based incremental nearest neighbor algorithm complementary to their algorithm. The RS-tree used in our algorithm is a hybrid of the R-tree and the S-tree, as its buddy tree, based on the hierarchical signature file. Experimental results show that our RS-tree enhances the performance of Hjaltason and Samet's algorithm.

1 Introduction

Recently, multimedia applications such as geographic information systems(GISs) and image retrieval systems have come to require the efficient processing of k-nearest neighbor queries over a collection of d-dimensional spatial objects. Many available techniques for such queries have been introduced[4, 5, 6, 14, 15, 21]. However, most of them have never focused on efficiently evaluating k-nearest neighbor queries with *non-spatial predicates* such as:

```
SELECT *
FROM DISC
WHERE artist = 'Beatles'
ORDER BY distance(color, red)
STOP AFTER k
```

This query, represented in an extended SQL language, is similar to Fagin's multimedia query `artist='Beatles'` $\wedge$ `color='red'`[11], meaning "choose k albums by the Beatles whose cover colors best match red". It involves the non-spatial predicate(i.e., `artist='Beatles'`) as well as spatial proximity(i.e., `distance(color, red)`). For processing it efficiently, it is necessary to *browse* through a collection of `color` vectors on the basis of their distances from a

†This work was supported in part by University Research Program supported by Ministry of Information & Communication in South Korea, entitled "A Study on Extending the Spatial Databases and Its Application to Spatial Data Warehousing".

given query object `red`[1] until k query answers are obtained. Up to date, Hjaltason and Samet's incremental nearest neighbor algorithm is known as the most useful algorithm for processing the query above[15].

Hjaltason and Samet's algorithm, using a hierarchical index like the R-tree, has the following advantage: when the upper operator(e.g., the *select* operator for `artist='Beatles'`) needs another TID[2] of tuple with the next neighbor, their algorithm can provide it for the upper operator without restarting the query from scratch[15]. However, their algorithm may generate a large number of tuples(or TIDs) that will turn out not to fulfill the remaining non-spatial predicates. Such worthless tuples generated during the query processing will lead to a longer query response time, which is on account of their algorithm behaving independently without using the given non-spatial predicates.

In the study, we propose a new algorithm capable of enhancing Hjaltason and Samet's algorithm when evaluating k-nearest neighbor queries with *non-spatial predicates*. Our algorithm is complementary to their algorithm in the sense that it can partially prune worthless tuples as early as possible. For this purpose, our algorithm takes advantage of an *RS-tree* as a hierarchical index, while their algorithm employs an R-tree without the facility of pruning worthless tuples. The RS-tree is a hybrid of the R-tree and as its buddy tree, the S-tree based on a hierarchical signature file index[9] where a non-spatial attribute(e.g., `artist`) is used as the key. Unlike the R-tree used in Hjaltason and Samet's algorithm, the S-tree offers our algorithm the facility mentioned above.

In this paper, we assume that the query contains only *one equal selection* predicate(i.e, =) like `artist='Beatles'`. The "equal selection" assumption is founded on the fact that the signature file which the S-tree builds on is used for handling *keyword*-based queries in information retrieval systems. We can relax the "one" constraint so that the query involves a boolean predicate(i.e, $\wedge$ or $\vee$) like `artist='Beatles' OR artist='Sting'` or `artist='Beatles' AND artist='Sting'`, or that the given non-spatial predicate is in the form of `artist='B*'`. However, we believe that our techniques in this paper can be applied to these classes of queries, since the S-tree inherently supports such queries. For the remainder of this section, we use the term *a non-spatial predicate* as *an equal selection predicate*. The predicates above often appear in GIS applications(e.g., Paradise[16] and Dedale[13]), or image retrieval systems(e.g., Chabot[20] and QBIC[18]).

The rest of this paper is organized as follows: Section 2 sketches the current available techniques for k-nearest neighbor queries. Section 3 reviews Hjaltason and Samet's algorithm and indicates its problem. Section 4 outlines our key ideas capable of overcoming the problem and presents the

[1] Although it really means `red`'s d-dimensional feature vector, we use this in the same context.

[2] A TID denotes the ID of each of tuples stored into for example, a `DISC` table. It is assumed that each leaf node in the hierarchical index stores a pair of a spatial object and a TID.

RS-tree as an implementation of our strategies followed by the signature chopping technique to enhance the performance of the pure RS-tree. Section 5 describes an RS-tree incremental nearest neighbor algorithm that we propose and proves that our algorithm is theoretically better than Hjaltason and Samet's algorithm. Section 6 shows the experimental results, and finally, Section 7 presents our conclusion.

2 Related Work

In multimedia database systems or geographic information systems, a lot of available techniques for k-nearest neighbor queries were introduced. They include techniques based on multidimensional indexes(e.g., Voronoi cell based algorithm[5], branch & bound algorithm[21], the incremental nearest neighbor algorithms[6, 14, 15], multi-step algorithms[22], and so on), the vector approximation technique for an efficient sequential scan[23], a fast parallel method[3], etc. In many applications, an approximate query result suffices, so that the approximate nearest neighbor algorithms have been developed[1], thereby saving the cost in producing the exact query result.

The branch & bound algorithm using *mindist* and *minmaxdist* of [21] and the vector approximation technique of [23] require that k be known prior to the processing of the query, thus having to restarting the query from scratch. Therefore it is inefficient to process queries with "distance browsing" concept described in Section 1. On the contrary, the incremental nearest neighbor techniques can toss incrementally, i.e., one by one, the next neighbor to an upper operator without exhausting the restart cost. Such techniques as these include [6], [14], and [15], based on the k-d-tree, LSD-tree, and hierarchical data structures like the R-tree, respectively. Among them, Hjaltason and Samet's algorithm is simple, efficient, and general enough to be adapted to any hierarchical indexes[15]. Also, a multi-step k-nearest neighbor algorithm, which is based on the incremental algorithm of [15], improved the performance of [17] by way of finding the minimum number of candidates, *optimal* d_{max}[22].

In relational database systems, there have been several optimizing techniques for processing *top-k* queries over traditional tuples[8, 19, 10]. The *top-k* query refers to the query that selects the first k tuples based on how they match given query values or scoring functions(e.g., *Min, Max, Euclidean, Sum*, etc.).

Fagin[11] proposed a simple way of answering queries like "`artist='Beatles'` $\wedge$ `color='red'`". His approach is under the assumption that sub-systems are involved in evaluating each predicate(e.g., a traditional relational database system and QBIC-like image retrieval system), and that the non-spatial predicate(e.g., `artist='Beatles'`) is *highly* selective. In addition, His approach corresponds to a *sequential scan method* without using any spatial index like the R-tree. On the contrary, we assume in our approach that the non-spatial predicate is so *lowly* selective

that the query optimizer decides to use the spatial index(Note that Hjaltason and Samet's algorithm is based on the spatial index). Moreover we also restrict the constraint to the cardinality of the query result.

3 Incremental Nearest Neighbor Algorithms

In this section, we review Hjaltason and Samet's algorithm known to be a state-of-the-art algorithm that can process distance browsing queries efficiently. Then we will indicate the problem of their algorithm when it is applied to answering such queries.

3.1 State of the Art Algorithm

Hjaltason and Samet[15] presented the R-tree-adapted incremental nearest neighbor algorithm[3](from now on, *RtreeINN*). The *RtreeINN* algorithm first initializes the priority queue. Then the algorithm traverses the root node in the R-tree, and for each child node in the root node, enqueues, as a key, the distance from the query object to its MBR[4] together with *pointer* pointing at it. Then the algorithm fetches the next *target node* (or *object*) from the front of the queue. The next target is one of three kinds: *a non-leaf node, a leaf node*, or *an object*. If it is a non-leaf node, the same process as that of the root node above is performed. If it is a leaf node, each object in this node and its real distance from the query object are inserted into the queue along with additional TID(for evaluating the remaining predicate, e.g., `artist='Beatles'`). Lastly, the case where it is an object indicates that this object has the smallest distance among all objects both existing in the queue and generated afterwards. Therefore it is returned to the upper operator. Note that the queue is always sorted based on the distance. The previous processes are repeated until the upper operator informs the algorithm of "stopping", or until the queue is empty.

3.2 Performance Drawbacks

Betchtold et al.[4] defined the optimality of the k-nearest neighbor algorithm that uses a hierarchical index like the R-tree, and showed that the *RtreeINN* algorithm is optimal. Note that the query result size k is involved in k-nearest neighbor queries, irrelevant to non-spatial predicates. The definition of optimality is presented below:

Definition 1 *Optimality[4]*
An algorithm for the k-nearest neighbor search is optimal if the pages accessed by the algorithm during the k-nearest neighbor search is exactly the pages that intersect $SP(Q,r)$.

[3]The algorithm assumes the case where the R-tree stores d-dimensional *points*. A general algorithm for any spatial object(e.g., polygon, polyline, point, and so on) was described in [15].

[4]Each R-tree node contains an array of (MBR, *pointer*) entries where MBR is an hyper-rectangle that minimally bounds the spatial objects in the subtree pointed at by *pointer*.

In Definition 1, Q and r are a query object and the distance from Q to the object(O^k) which is the k-th farthest from Q(i.e., $r = \|Q - O^k\|$), respectively. $SP(Q,r)$ is a d-dimensional hypersphere with center Q and radius r.

From now on, we consider k-nearest neighbor queries with *a non-spatial predicate*. Let $O^{k'}$ be the last object that $RtreeINN$ must search in order to provide the upper operator with the TID which will be the k-th answer. The problem of $RtreeINN$ is that most of the k' objects(or TIDs) would not satisfy the non-spatial predicate evaluated by the upper operator. For discussing this problem in detail, we derive the rough cost of query processing using $RtreeINN$. The total cost is $C_{all} = C_{rtree} + C_{tuple}$, where C_{rtree} and C_{tuple} denote the R-tree node and tuple I/O cost, respectively[5].

By the optimality of $RtreeINN$ above, the cost C_{rtree} equals the number of R-tree pages intersected by the sphere $SP(Q,r)$, where $r = \|Q - O^{k'}\|$. Before finding the cost C_{tuple}, let us approximate the number(k') of objects(or TIDs) that must be tossed to the upper operator. For the cardinality of the query result k, the approximation of k' is derived as follows: We assume that the non-spatial attribute(e.g., `artist`) is not correlated with the spatial attribute(e.g., `color`). Let S be the selectivity of a given query value(e.g., 'Beatles') and O^i be an object that satisfies the i-th query answer. For finding the $(i+1)$-th answer, $RtreeINN$ must access $\frac{1}{S}$ more objects, which are object O's such that $d_i \le \|Q - O\| \le d_{i+1}$ where $d_i = \|Q - O^i\|$, intuitively. As a result, the approximation of k' is $k'_{approx} = \sum_1^k \frac{1}{S} = \frac{k}{S}$. The approximation k'_{approx} increases with the slope of $\frac{1}{S}$ as k grows. Lastly, we approximate the tuple scan cost C_{tuple} using k'_{approx}. For this goal, we use the Yao function $b(x, y, z)$[24]. The tuple scan cost is $C_{tuple} = b(\frac{N \cdot T}{B}, \frac{B}{T}, \frac{k}{S})$, where N, B, and T are the number of overall tuples, the page size, and the tuple size, respectively. See the Appendix for details.

We conducted experiments to identify the tuple candidate estimate, k'_{approx}. In our experiments, we implemented the $RtreeINN$ algorithm in [15] and ran queries such as "`artist='Beatles'` $\wedge$ `color='red'`". The attribute `artist` values were generated from Uniform Distribution. For the attribute `color`, synthetic or real-life datasets were used(for details, refer to Section 6). Figure 1 shows that the two curves increases linearly as we expected, and the ratio of the query result size k to the size of tuple candidates is too *high*. In this observation, we can estimate that most of the candidates returned to the upper operator are worthless, resulting in expensive tuple scan cost.

In short, $RtreeINN$ is efficient in searching the next neighbor in favor of no restart cost, but is not so in dealing with k-nearest neighbor queries with *a non-spatial predicate* owing to the

[5]Except for two cost factors, the queue I/O and CPU computation cost actually reside in the query processing cost, but are negligible since they are not dominant compared to the two costs above.

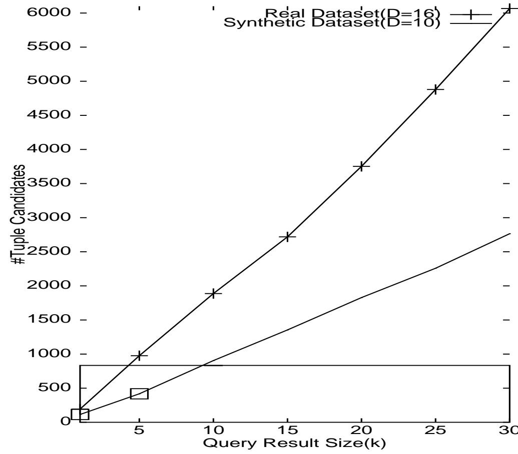


Figure 1: Observation on the Candidate Size(N=100,000)

generation of many worthless candidates. We believe that such a handicap of $RtreeINN$ is due to the fact that the R-tree used in $RtreeINN$ has no device capable of filtering them.

4 Our Approach

In this section, we illustrate our techniques that lessen the deficiencies of Hjaltason and Samet's approach. This section is organized as follows: Section 4.1 outlines our key ideas to provide us a way to get out of shortcomings of their algorithm. Section 4.2 describes the RS-tree as an implementation of our ideas. Section 4.3 presents the pruning effect of the S-tree. Finally, Section 4.4 proposes the signature chopping technique to improve the performance of the pure RS-tree.

4.1 Key Ideas

As indicated in Section 3.2, the disadvantage of the $RtreeINN$ algorithm results from the R-tree with no ability of pruning worthless tuples. For such a ability, the R-tree is required to be equipped with a new data structure, or a buddy that will help the R-tree reduce the number of worthless tuples. In Figure 2, the R-tree is responsible for evaluating the spatial predicate, e.g., `color='Red'`, and the S-tree(to be described in Section 4.2) takes charge of partially handling the non-spatial predicate, e.g., `artist='Beatles'`. Before the R-tree puts each of the child nodes in the current target node on the queue(Section 3.1), the R-tree asks its buddy if the subtree rooted at each child node really includes a given non-spatial value or not. Then the R-tree receives the answer from its buddy. If the answer is "Yes", then the R-tree will insert the corresponding child node into the queue($ST1$ in the figure). Otherwise the R-tree will look upon that child node as a worthless node and thus prune the subtree rooted at it($ST2$ in the figure). Pruning subtrees like $ST2$ will lead to significant additional savings during the query processing, considering that

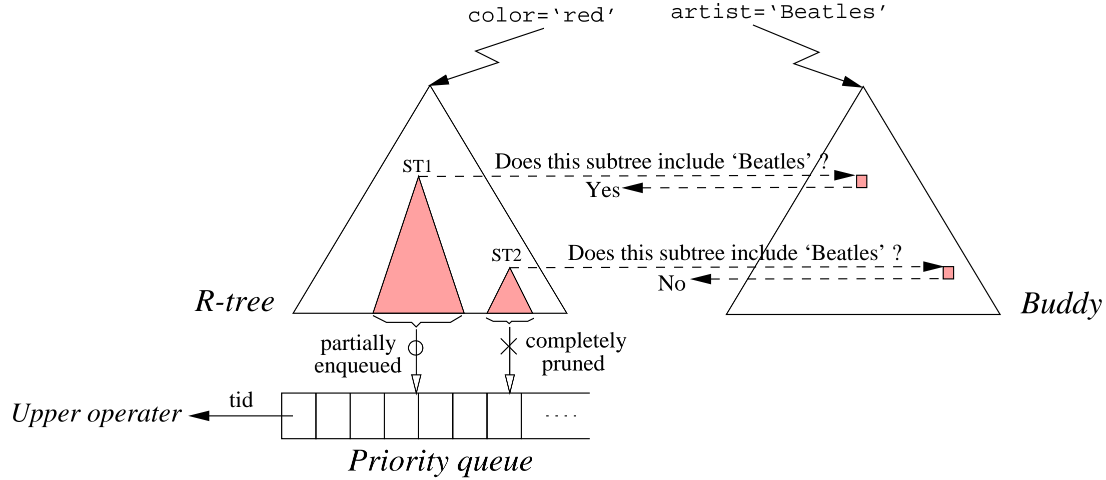


Figure 2: An Example of A Buddy Tree

a lot of worthless tuples below such subtrees might otherwise be returned to the upper operator.

In order for the R-tree's buddy to perform the "inclusion test" well, it is necessary to build a data structure with the following properties:

1. *Hierarchy*: The algorithm traverses the R-tree hierarchically and also sends a question to the R-tree's buddy in the same fashion. Thus it is advantageous that this buddy has a hierarchical structure like the R-tree.

2. *Transitive inclusion relation*: This property is derived from the "hierarchy" property above and the fact that if a certain subtree includes a query value, then its *ancestor* subtree must also include this value.

3. *Small storage* and *inexpensive computation*: The performance improvement by virtue of the R-tree's buddy should be beneficially traded off with overheads resulting from it.

In the following, we propose the *RS-tree* that consists of the R-tree and the S-tree as its buddy.

4.2 The RS-tree

4.2.1 Description

The RS-tree is a hybrid of the R-tree and the S-tree based on the hierarchical signature file, which are used as data structures for the spatial attribute(e.g., `color`) and for the non-spatial attribute(e.g., `artist`), respectively. A tuple generally involves several non-spatial attributes besides the spatial ones, therefore we can extend the RS-tree to the RS^m-tree where the S^m-tree is composed of m S-trees. We describe our new data structure focusing on the RS-tree, i.e., $m = 1$.

Since the R-tree in the RS-tree is not different from the R-tree used in the $RtreeINN$ algorithm,

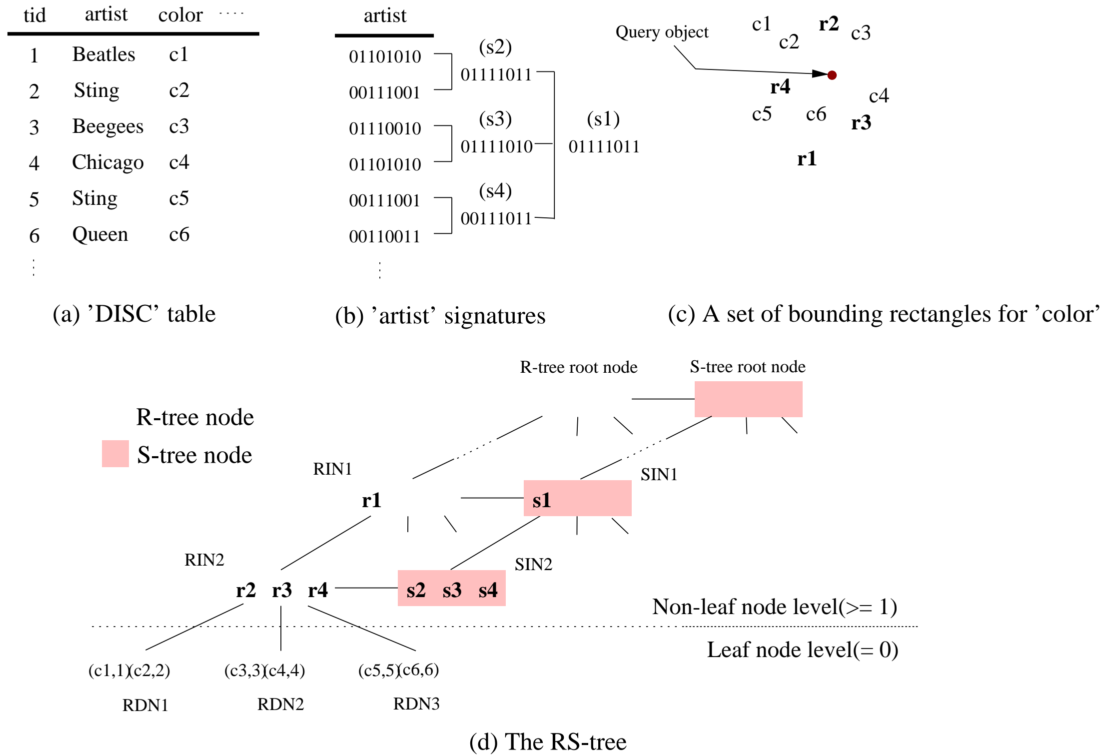


Figure 3: The Construction of the RS-tree

we pay much attention to the S-tree. The S-tree is a *signature hierarchy* whose structure is similar to the R-tree. Each S-tree node contains an array of (*key, pointer*) entries where *key* is a *signature*(representing a set value) which bounds all the non-spatial values in the subtree pointed at by *pointer*. The signatures in the S-tree node are used for *signature checking*(to be described in Section 4.3). The reasons why we used the *signature* technique to represent a set value are from the properties of the S-tree to be presented later.

Not only the S-tree node but also its entries are symmetric with those of the R-tree. Each R-tree node contains an extra pointer(e.g., a page number) that points at the corresponding S-tree node. The R-tree has both non-leaf nodes and leaf nodes, while the S-tree has only non-leaf nodes(i.e., the level $l \geq 1$), since its storage overhead becomes high if it would store even leaf nodes. Note that in the R-tree, the storage ratio of leaf nodes to non-leaf nodes is greatly high. Of course, allowing S-tree leaf nodes can improve the pruning effect of the S-tree, but it may be impractical due to its huge storage overhead.

Assuming that the S-tree is built at the time when the R-tree is bulk-loaded, we can classify the construction of the S-tree into two types according to the level of the S-tree, i.e., $l = 1$ and $l > 1$. In the case where $l = 1$, a signature of each entry in the S-tree node is created as follows: 1) After an RS-tree bulk-loader generates a certain R-tree leaf node, it promptly looks for a set of

non-spatial attribute values through TIDs in that leaf node. 2) Each value in the set is modified into a bit stream of a fixed size, i.e., a signature, using a signature hashing function. 3) All signatures in the hashed set are ORed on a bit-by-bit basis. The ORing result is a signature that we would like to obtain at the bottom level of the S-tree. At the higher level than 1, a signature of each entry in the S-tree node is created by ORing all the signatures in the child node pointed at by *pointer*. In this manner, the last S-tree root node is generated and the complete S-tree is constructed.

Example 1: In Figure 3(a), consider a `DISC` table with both `color` and `artist` attributes that are used as keys of the R-tree and S-tree, respectively. Figure 3(b) shows a signature table for `artist`. Figure 3(d) is the RS-tree for a set of bounding rectangles for `color` in Figure 3(c). In Figure 3(d), a set of dark nodes linked hierarchically and the rest correspond to the S-tree and R-tree, respectively. To generate, for example, the signature $s2$ in $SIN2$ at the bottom level, we first chase the R-tree node $RIN2$ mapped to $SIN2$ and obtain a set of { 'Beatles', 'Sting'} through TIDs(i.e., 1 and 2) in $RDN1$. Then we convert the set of the searched non-spatial values into the set {01101010, 00111001} using a given signature hashing function and finally perform an OR operation for two signatures, resulting in 01111011. In the case where $l > 1$, for example, the signature $s1$ in $SIN1$ is generated by ORing $s2$, $s3$, and $s4$ in the child node $SIN2$ below it, resulting in 01111011.

4.2.2 Properties

The S-tree reflects the property requirements described in Section 4.1. Hence, our algorithm(to be presented in Section 5) can win some advantages over Hjaltason and Samet's algorithm. Detailed properties of the S-tree are as follows:

- The S-tree is a *set of signatures*: A signature is used as a set value, as mentioned above.
- The S-tree is a *hierarchical data structure* like the R-tree.
- The S-tree supports the *good transitive inclusion relation*: The two properties above satisfy this.
- The S-tree has *individual storage* independent of the R-tree: The S-tree is not subject to the R-tree with regard to physical storage. Therefore the performance degradation of the primary R-tree due to associating it with the S-tree never occurs.
- The S-tree has a *small storage overhead*: A signature size is relatively small[12], and also the S-tree has pages of the same size as that of the R-tree *non-leaf* pages.

- The S-tree has *inexpensive computation cost*: A signature is a simple bit stream with '0' or '1', so the computation cost for AND or OR operations is fully saved[12].

4.2.3 Drawbacks

The RS-tree is subject to some deficiencies including the long bulk-loading time or the slightly expensive update cost. When a new spatial object is inserted into or deleted from the R-tree in a dynamic environment, splitting or merging propagation along the hierarchical paths will be done not only to the R-tree pages but also to the corresponding S-tree pages. Thus the update cost will be about two times as high as that of the case where only the R-tree is used in the algorithm. However, it is highly unlikely that multimedia database systems are as dynamic as traditional relational database systems. Hence, we believe that the performance improvement by the RS-tree can sufficiently compensate for the deficiencies resulting from the S-tree.

4.3 Pruning Effect of the S-tree

In the following, we present how worthless R-tree nodes and objects(or TIDs) can be partially pruned by the S-tree in the context of the S-tree-aided $RtreeINN$ algorithm.

Before proceeding any further, we explain the property of the signature hashing function that we used in building the S-tree. Each non-spatial value yields m bit positions(not necessarily distinct) in the range 1-F. The corresponding bits are set to 1, while all the other bits are set to 0. For example, in Figure 3, the value "Beatles" sets to 1 the bits of positions 2, 3, 5 and 7($m = 4$). The signature for each non-spatial value is referred to the "word" signature. The "block" signature is based on superimposed coding[7], that is, ORing a given number of word signatures. The inclusion test of a word into a block signature is performed as follows: The signature($= s_q$) of the word(e.g., "Beatles") is created. Suppose that the signature contains 1 in positions 2, 3, 5 and 7. The block signature($= s_j$) is examined. If the above positions(i.e., 2, 3, 5, 7) of the block signature contain 1, in other words, $s_q \wedge s_j = s_q$ where $\wedge$ denotes an AND operator(a counterpart of the OR operator), the block signature is once considered to include the word. Otherwise, it is discarded.

Let R be the current target node at the front of the queue, and $r_j (1 \leq j \leq C)$ be its child node, where C is a *fanout* of the R-tree node. Before computing the distance for r_j, $RtreeINN$ performs the "inclusion test" for each signature s_j in the S-tree node S corresponding to R. That is, for the query signature s_q(which is generated by hashing the query value, e.g., 'Beatles'), the algorithm checks if $s_q \wedge s_j = s_q$. We refer to this testing as *signature checking*[6]. If s_j passes the signature checking for s_q, it means that there may be some values equivalent to the query value

[6]If the target is an R-tree leaf node or an object, the algorithm does not perform the signature checking since the S-tree has only non-leaf nodes.

in the subtree rooted at r_j. If not, it is guaranteed that there are no such values in the subtree. Therefore, we have only to enqueue the child nodes in R whose corresponding s_j's in S pass the signature checking for s_q.

This strategy, based on the S-tree-supported pruning, offers additional savings for handling k-nearest neighbor queries with a non-spatial predicate, which are as follows:

- Disk I/Os for accessing the pruned R-tree nodes and their child nodes are saved.
- Disk I/Os for accessing the worthless tuples resulting from the pruned R-tree nodes are eliminated.
- Distance computations for the R-tree nodes pruned are removed.
- Queue management costs are reduced.

Undoubtedly, the S-tree node I/Os are required for the signature checking, but the benefit of pruning by the S-tree is greater than this cost(See experimental results in Section 6).

Example 2: In Figure 3, let us assume that $RIN2$ is the current target node. We also assume that $d3$ is smaller than $d1$, $d2$, and any distance in the queue, where d_i denotes the distance from the query object to the child node RDN_i of $RIN2$. For the query signature $s_q = 01101010$, which is hashed for 'Beatles', the signature checking is performed for $s2$, $s3$, and $s4$, and consequently, except for $s4$, $s2$ and $s3$ will pass the test since for $s4$, $s_q \wedge s4 \neq s_q$ where $s4 = 00111011$. Therefore, only $RDN1$ and $RDN2$ will be inserted into the queue. Note that in the *pure RtreeINN* algorithm, $RDN3$ will be enqueued, while in the S-tree-aided *RtreeINN* algorithm, it will be pruned.

"Signature saturation" occurs with the growing level of the S-tree, where each bit in a signature is highly likely to be '1'. Since such phenomena raise the *phantom effect* significantly, it makes the pruning effect of the S-tree *weaker*, resulting in a lot of *false drop*s. A false drop refers to a *worthless tuple* that will turn out not to satisfy the remaining query predicate, e.g., `artist='Beatles'`, although all the signatures along its R-tree hierarchical path will have passed the signature checking. For instance, $s3$ passes the signature checking for $s_q = 01101010$, in Figure 3, but because no tuples identified by TIDs(i.e., 3 and 4) in $RDN2$ satisfy the predicate, `artist='Beatles'`, they result in false drops. This is due to the fact that $s3$ incurred the phantom effect for s_q. At a high level of the S-tree, the phantom effect will be severe, and this will mitigate the pruning effect of the S-tree. In the next subsection, we shall explain how to diminish false drops resulting from signature saturation.

4.4 Signature Chopping

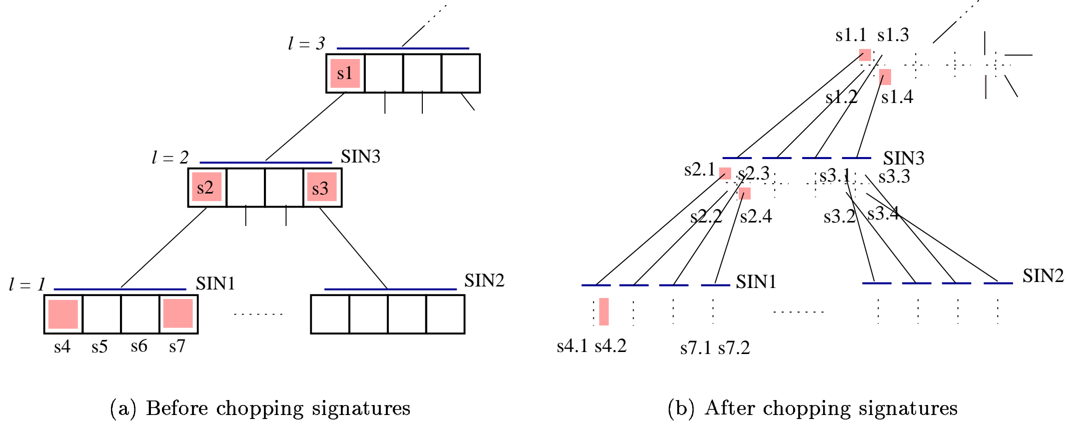
We can identify signature saturation formally: Let α_L be the probability that the i-th bit of the child signature is '1' and N_L be the number of child signatures ORed for a parent signature. The probability that the i-th bit of the parent signature is '1' is represented as $\alpha_P = 1-(1-\alpha_L)^{N_L}$ [9]. Suppose that a relatively small α_L, say 0.3, is used(In order to make α_L smaller, the signature of large size is required.). We can estimate that if N_L becomes larger, α_P is likely to be closer to '1'. As the level of the S-tree grows, N_L will become larger. Therefore, by the above estimation, the speed of signature saturation will be more rapid with the growing level of the S-tree. We can relax the saturating rate by making N_L in the above formula *smaller*. In the following, we focus on how to make N_L smaller, which we call the ***signature chopping*** technique.

The goal of signature chopping is that by partitioning each signature in the S-tree node into several signatures according to a given signature chopping function, $f(l)$(e.g., 2^l), let N_L be as small as possible and thereafter the speed of signature saturation be less[7]. For a signature $s_i (1 \leq i \leq C)$ in the S-tree node on level l where C is the fanout of the R-tree node, s_i is partitioned into the $f(l)$ chopped-signatures $s_{i.j}$'s, $1 \leq j \leq f(l)$. If $f(l) > C$ on a certain level l of the S-tree, $f(l) = C$. The C signatures in the child node below s_i are divided into separate buckets by placing the first $\lceil \frac{C}{f(l)} \rceil$ signatures into bucket #1, the next $\lceil \frac{C}{f(l)} \rceil$ signatures into bucket #2, and so on. Then each $s_{i.j}$ is generated by ORing the $\lceil \frac{C}{f(l)} \rceil$ signatures in the j-th bucket. Note that $s_i = \bigvee_{j=1}^{f(l)} s_{i.j}$, where $\bigvee$ is an OR operator in the context of the signature technique. As a result, the degree of signature saturation, namely the phantom effect is reduced to about $\frac{1}{f(l)}$ times as small as that of the non-signature chopping approach.

Example 3: Figure 4(b) shows the S-tree when applying the signature chopping technique to the S-tree of Figure 4(a), where $C = 4$ and $f(l) = 2^l$. Let us assume that the R-tree node corresponding to the S-tree node SIN_i is RIN_i, and that $RIN1$, $RIN2$, and $RIN3$ will be the next target nodes. The dark regions in the figure denote the (chopped-)signatures that will pass the signature checking. $s2$ on level 2 is divided into four chopped-signatures, $s2.1$, $s2.2$, $s2.3$, and $s2.4$, because $f(2) = 2^2 = 4$. The bucket size of each of the chopped-signatures is 1, since $\lceil \frac{4}{f(2)} \rceil = 1$. In the case of $s1$ on level 3, $f(l)$ is set to 4, since $f(3)(= 2^3 = 8) > C$.

In the signature chopping approach, signature checking is performed for each of the $f(l)$ chopped signatures. Let R be the current target node and S be the S-tree node corresponding to R. Also, let $s_{i.j}(1 \leq j \leq f(l))$ be the j-th chopped-signature of the i-th entry in S. If none of $s_{i.j}$'s passes the signature checking, then the child node($CHILD_i$) of the i-th entry in R will be pruned, not be enqueued. For example, none of $s3.1$, $s3.2$, $s3.3$, and $s3.4$ in Figure 4

[7] If $f(l)$ is larger, overflow pages in the S-tree may occur. Therefore, it is necessary to choose a suitable $f(l)$.



(a) Before chopping signatures

(b) After chopping signatures

Figure 4: Signature Chopping

passes the signature checking. Hence, the R-tree node $RIN2$ corresponding to $SIN2$ will be pruned($CHILD_i = RIN2$). In the other case, $CHILD_i$ will be enqueued along with a hint(e.g., a bitmap). The hint means that when $CHILD_i$ will be the next target node in the future, the subtrees rooted at the $\lceil \frac{C}{f(l)} \rceil$ entries in $CHILD_i$ corresponding to the S-tree child node of the chopped-signature which never passes the signature checking need never be accessed at all. Note that each $s_{i.j}$ is responsible for the $\lceil \frac{C}{f(l)} \rceil$ entries in $CHILD_i$. For example, in Figure 4, $s2.2$ does not pass the signature checking. Therefore, the subtree rooted at the second entry in the R-tree node $RIN1$ corresponding to $SIN1$ need never be accessed at all, when $RIN1$ will be the next target node in the future. Such information is stored into the hint.

By comparing two figures of Figure 4, we find that in the left figure, the signatures s_3, s_4, and s_7 give rise to the phantom effect, while in the right figure, it is alleviated by employing the signature chopping technique. The expensive costs saved by the signature chopping technique include disk I/Os to access the S-tree or R-tree pages caused by for example, $s3$ and $s7$, and moreover disk I/Os to read a large number of potential worthless tuples resulting from for example, $s4.1$, $s7.1$, and $s7.2$.

5 RS-tree Incremental Nearest Neighbor Algorithm

In this section, we present the RS-tree incremental nearest neighbor algorithm that is complementary to the algorithm described in Section 3. Then we make an analysis between the two algorithms.

Algorithm 1 RStreeINN(Q_s, Q_r)

```
    /* Qs and Qr denote a given spatial query object and a non-spatial query value, respectively */
1   Sq := hash(Qr)                                  /* a query signature */
2   Queue := PriorityQueue()
3   NewElm.pointer := RtreeRootNode; NewElm.dist := 0; NewElm.bitmap.setall()
4   Enqueue(Queue, NewElm)
5   while Queue is not empty
6       Elm := Dequeue(Queue)                       /* the next target node is fetched from the queue */
7       if Elm.pointer is a non-leaf node           /* the next target node is a non-leaf node */
8           for each entry(child, mbr) in Elm.pointer
                /* check whether current entry was pruned by the previous signature checking or not */
9               if Elm.bitmap.isset(EntryIndex) is True
                    /* the corresponding S-tree page is read, ChoppedSignArray is filled with the f(l) chopped
                    signatures in that page, and signature checking is performed */
10                  NewBitmap := SignChecking(Sq, ChoppedSignArray)
                    /* check whether at least one chopped signature passed the signature checking or not */
11                  if at least one bit in NewBitmap is not zero
12                      NewElm.pointer:=child; NewElm.dist:=DIST(Qs, mbr)
13                      NewElm.bitmap:=NewBitmap
14                      Enqueue(Queue, NewElm)
15                  end if
16              end if
17          end for
18      else if Elm.pointer is a leaf node          /* the next target node is a leaf node */
19          for each entry(object, tid) in Elm.pointer
20              if Elm.bitmap.isset(EntryIndex) is True
21                  NewElm.pointer := object; NewElm.dist := DIST(Qs, object)
22                  NewElm.tid := tid
23                  Enqueue(Queue, NewElm)
24              end if
25          end for
26      else                                        /* Elm.pointer is an object */
27          return Elm.tid
28      end if
29  end while
```

Algorithm 2 SignChecking(S_q, ChoppedSignArray)

```
1   S := ChoppedSignArray
2   for i := 1 to f(l)                              /* l is the current level of the S-tree */
3       if (Sq AND Si) == Sq
4           for j := i*AllocSize(l) to (i+1)*AllocSize(l)
5               Bitmap.set(j)
6           end for
7       end if
8   end for
9   return Bitmap
```

Algorithms 1 and 2 specify our algorithm and signature checking routine, respectively. Additional parts in our algorithm are mainly steps identifying what child nodes or objects within the the next target node were pruned(Steps 9 and 19 in Algorithm 1), and signature checking steps prior to inserting them into the queue(Step 10 in Algorithm 1). In Algorithm 2, *AllocSize(l)* denotes the number of entries in the R-tree node on level l allocated to each chopped signature(i.e., the size of each bucket in Section 4.4). Our algorithm is not well-suited to queries with range

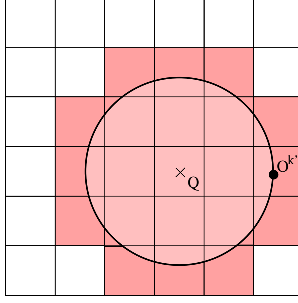


Figure 5: R-tree Node Search Space of the $RtreeINN$ Algorithm

predicates(i.e., $<$ or $>$). However, for handling such queries, we can apply the $RtreeINN$ algorithm using the R-tree of our RS-tree without degrading the performance. This is because in the RS-tree, the R-tree has individual storage independent of the S-tree.

5.1 Analysis

We prove theoretically that the signature-chopping based $RStreeINN$ algorithm is better than the $RtreeINN$ algorithm, and quantify the performance gain of our algorithm over the other. Note that we do not derive the comprehensive cost models of two algorithms. Deriving them are complicated, especially for high dimensional spaces. For details, refer to [4, 15].

In the following, we use terms such as k', $O^{k'}$, and $SP(Q,r)$ and their meanings in the same context as that of Section 3.2. Figure 5 depicts the R-tree node search space after finding the k-th answer using the $RtreeINN$ algorithm, where it is assumed that all grids specify R-tree leaf nodes. As described in Section 3.2, the R-tree node I/O cost($RtreeINN_C_{rtree}$) of the $RtreeINN$ algorithm is equivalent to all R-tree leaf nodes intersecting the hypersphere $SP(Q,r)$(the shaded region in Figure 5)[8]. Here we can divide $RtreeINN_C_{rtree}$ into two classes of R-tree leaf nodes, i.e., *inside*(the dark shaded region in the figure) or *intersecting*(the light shaded region in the figure) the boundary of $SP(Q,r)$, which we denote $PAGE_{inside}$ and $PAGE_{intersect}$, respectively. Thus the R-tree node I/O cost is $RtreeINN_C_{rtree} = PAGE_{inside} + PAGE_{intersect}$. It is reasonable to assume that on the average, half of the C points in each leaf node in $PAGE_{intersect}$ are inside the search region, while half are outside[15], where C denotes the average R-tree leaf node capacity. The total number of the tuples touched during the query processing is $RtreeINN_k' = C(PAGE_{inside} + \frac{1}{2}PAGE_{intersect})$.

We consider the $RStreeINN$ algorithm exploiting the signature chopping technique presented in Section 4.4. The main difference between the $RtreeINN$ and $RStreeINN$ algorithms is whether signature checking is performed or not. As mentioned in Section 4.4, a child node in the target node

[8] If the average leaf and non-leaf node capacities are fairly high, the number of leaf node accesses dominates the number of non-leaf node accesses[15]. Thus for the R-tree node I/O cost, we approximate the total number of node accesses by the number of leaf node accesses.

will be inserted into the queue only when among the $f(l)$ chopped signatures, at least one chopped signature passes the signature checking. Generally, the probability that a chopped signature will pass the signature checking corresponds to a *false drop probability*[9], namely F_{drop}, which is often addressed in the signature file techniques[12]. Thus, assuming that the signature chopping function $f(l)$ is in the form of $(base)^l$, say 2^l, the probability that at the bottom level(i.e., $l = 1$) of the S-tree, at least one chopped signature will pass the signature checking, i.e., a child node in the target node will be enqueued, is $P_{pass} = 1 - (1 - F_{drop})^{base}$. As a result, if we apply P_{pass} to $RtreeINN_C_{rtree}$ and $RtreeINN_k'$ of the $RtreeINN$ algorithm, the R-tree node I/O cost and the total number of the tuples accessed by the $RStreeINN$ algorithm are, respectively, bound by

$$P_{pass}(PAGE_{inside} + PAGE_{intersect}) \text{ and } P_{pass} \cdot C(PAGE_{inside} + \frac{1}{2}PAGE_{intersect}).$$

The tuple I/O cost depends on the total number of the tuples accessed, namely k'(See the Yao function in Section 3.2). Thus we can use k' as the performance measure for the tuple I/O cost. Putting the above formulas together, $RStreeINN$ outperforms $RtreeINN$ by a factor of $\frac{1}{P_{pass}} = \frac{1}{1-(1-F_{drop})^{base}}$ in terms of both the R-tree node and tuple I/O costs. It is always guaranteed that the performance gain is greater than 1, since $0 < F_{drop} < 1$ and thus $\frac{1}{1-(1-F_{drop})^{base}} > 1$.

6 Experiments

In this section, we study the performance of the two algorithms of Section 3 and Section 5 in evaluating k-nearest neighbor queries with a non-spatial predicate. The results presented in this section are based on an extensive experimental study on both a real dataset and synthetic datasets.

In comparing the two algorithms, we used three metrics: execution time, the number of tuples accessed, and the number of pages accessed. The second metric denotes the number of tuples(or TIDs) returned to the upper operator until the k answers are obtained. The third metric specifies the number of pages accessed for both tuples and R(S)-tree pages during the query processing. The above metrics were measured in a system that is equipped with a Pentium 133 MHz processor, 128 MB of main memory, and a 6 GB disk drive and operated by the PowerLinux 6.0 version.

6.1 Experimental Setting

6.1.1 Datasets

We defined a simple `DISC` table that our test queries are performed over and has the following self-explanatory schema:

```
DISC(artist, type, country, color)
```

`color` has a d-dimensional spatial domain where each value is *d*sizeof(float)* bytes long. The remaining attributes defined in the alphanumeric domain are about 30 bytes long, respectively.

[9]The false drop probability is approximated by $F_{drop} = (\frac{1}{2})^{\frac{1}{D_t} F \ln 2}$, where D_t is the cardinality of signatures ORed for a chopped signature, and F is the signature size in bits.

In all experiments, only `color` and `artist` are used as attributes for the test queries. The rest simply pad the tuples with characters to ensure that each `DISC` tuple is about 100 bytes long. A page is 4 KBytes long in storing the `DISC` table. The size of the cache for accessing tuples to evaluate test queries is about 1% of the `DISC` database size, which follows the LRU replacement policy.

For non-spatial attributes, we systematically generated only synthetic datasets from Zipfian Distribution with the Zipfian variable $z = 0.5$, where the number of distinct values of each attribute shows 0.2% of the number of tuples. For the spatial attribute `color`, we used both synthetic datasets and a real dataset: Each synthetic dataset contains d-dimensional points generated using Uniform Distribution, which are distributed in a $[0, 1]^d$ space. The dimension d ranges from 2 to 12 since the R*-tree is efficient in low- or middle-dimensional vector spaces. The number of points within each synthetic dataset is in the range of 100,000 to 500,000. The real dataset contains 16-dimensional Fourier points used in a CAD model[4], of which the size is 200,000.

6.1.2 Test Query Sets

The test query sets consist of about 50 test queries, and each experimental result is the average over about 50 test queries. Like the example query of Section 1, each test query involves three values: a d-dimensional point for `color`, the alphanumeric value of size 30 bytes for `artist`, and the query result size k. The spatial query point was randomly selected from the set of `color` values in the `DISC` database. The criteria for choosing the 50 query values of `artist` from the `DISC` database is how low the selectivity of each of them is. If the query value is lowly selective, the query optimizer is highly likely to use the spatial index rather than a sequential scan, subject to high sorting cost. Note that in this paper, the *RtreeINN* or *RSTreeINN* algorithms are applied to perform the test queries above .

6.1.3 Data Structures

In this paper, the two algorithms for the distance browsing queries use the R(S)-tree. In our experiments we make use of a variant called the R*-tree[2] with the key being `color`. The S-tree, of which the key is `artist`, for *RStreeINN* was constructed in conjunction with the bulk-loading of the R*-tree. When building the S-tree, we applied various signature chopping functions $f(l)$'s(e.g., $2^l, 3^l$, etc.) under the constraint that no overflow pages occur except for particular cases(e.g., $d = 2$). When generating a signature, we used the hash function of [12]. In all experiments, we used the signature of a fixed size, i.e., F=64 bits(= 8 bytes). We believe that using a larger signature will improve the performance under the condition that overflow pages never occur. The size of each page is 4 KBytes in storing spatial indexes used in two algorithms, and the cache of each spatial index accommodates a page in size. The priority queue used in the

two algorithms was implemented to permit a memory-based data structure for it.

6.2 Experimental Results

In this subsection, we present experimental results of the two algorithms on both synthetic and real-life datasets. We study the performance of two techniques with respect to the *query result size*, the *signature chopping function*, the *dataset size* and the *number of the dimension*.

6.2.1 Synthetic Dataset

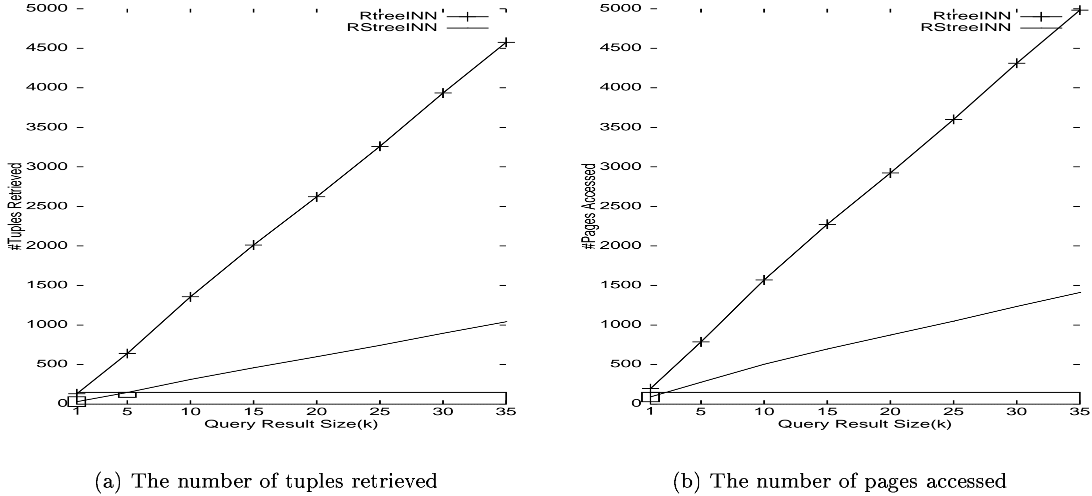
This subsection presents experimental results on synthetic datasets. In reality, the R*-tree is efficient for low- or middle-dimensional datasets. For this reason, we performed experiments for 6-dimensional synthetic datasets containing *color* feature vectors, except the experiment for the effect of different dimensions.

Effect of Different Query Result Sizes(k): This experiment studies the performance of the two algorithms for different query result sizes. Figure 6 shows the execution time, the number of pages accessed, and the number of tuples retrieved with the following experimental parameters: the dimension of `color` $d = 6$, the Zipfian variable of `artist` $z = 0.5$, the number of tuples $N = 100,000$, the signature partition function $f(l) = 6^l$, and the signature size $F = 64$(Bits).

Figure 6(a) shows the number of tuples retrieved, which corresponds to the number of tuple candidates that were returned to the upper operator. The curves of both of the algorithms increase linearly as k becomes larger. However, the curve slope of $RStreeINN$ is less steep than that of $RtreeINN$. The reason for this is that $RStreeINN$ has the effect of pruning the R*-tree nodes or objects(or TIDs) using the S-tree. Figure 6(b) reveals the number of pages accessed(i.e., the R(S)-tree node I/Os plus the tuple access I/Os) until the k answers are obtained. The curves of both of the algorithms, as shown in Figure 6(a), increase as k grows and resemble those of Figure 6(a), because for relatively small datasets, the tuple access I/Os affect the number of pages accessed more than the R(S)-tree node I/Os do.

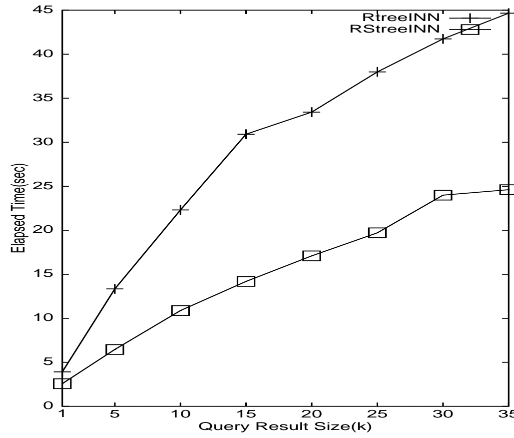
In Figure 6(c), we find that the difference in the number of tuples accessed between the two algorithms is reflected in the execution time of Figure 6(c). However, the two graphs are not so exact as we expected them to be, since $RStreeINN$ accesses the R-tree pages and the S-tree pages alternatively and therefore must pay off the random disk I/O cost. Nonetheless, $RStreeINN$ outperforms $RtreeINN$ by about 100%.

Effect of Different Signature Chopping Functions($f(l)$): In this experiment, we study the impact of the signature chopping function on the performance of the two algorithms, where experimental parameter values, except for the signature chopping function are equivalent to those of the previous experiment.



(a) The number of tuples retrieved

(b) The number of pages accessed

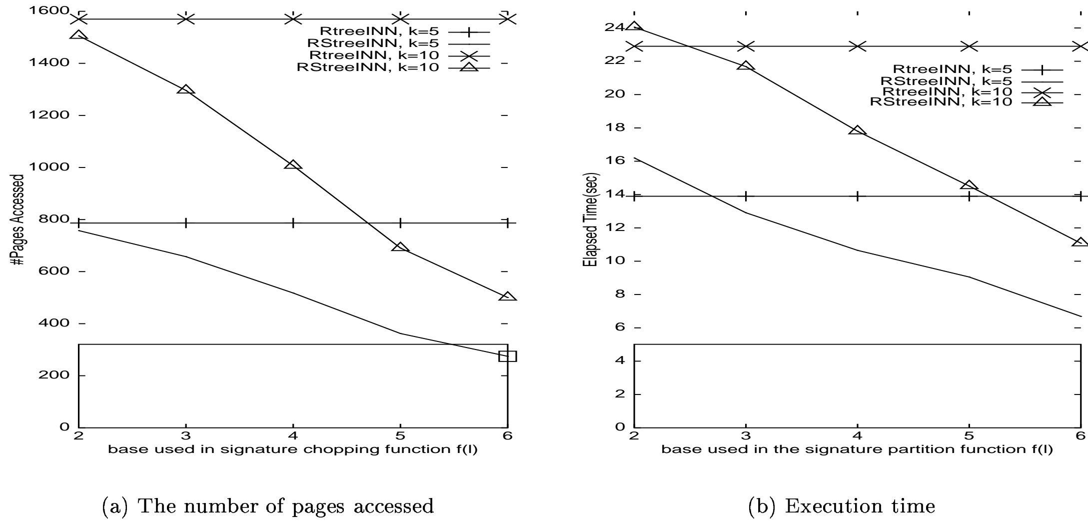


(c) Execution time

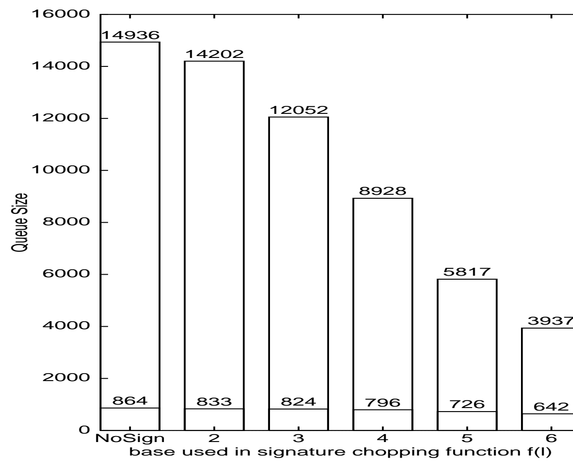
Figure 6: Performance vs. Query Result Size k (d=6, N=100,000, z=0.5, $f(l)$=6^l, F=64)

We can expect that the performance of *RStreeINN* will be affected by the degree of signature chopping(i.e., $f(l)$) and maybe will be in proportion to it. However, we must also consider that overflow pages subject to a large $f(l)$ may degrade the performance of the algorithm. We experimented the signature chopping function $f(l)$ to such an extent that $f(l)$ does not give rise to the overflow pages. The results are like in Figure 7, where the query result size k= 5 or 10.

Figure 7(a) shows that for the number of pages accessed, the curve of $RStreeINN$ declines significantly as the signature partition degree(i.e., the *base* in Figure 7) increases. In Figure 7(a), the curve of $RtreeINN$ is stationary regardless of $f(l)$. The incremental difference of the heights of the two curves as the degree of $f(l)$ grows is reflected in the execution time of Figure 7(b). However, in the case where $f(l) = 2^l$(i.e., $base$ = 2 in Figure 7(b)), the performance of $RStreeINN$ is worse than that of $RtreeINN$ since the benefit of pruning by the S-tree does not compensate for the



(a) The number of pages accessed (b) Execution time



(c) The queue size

Figure 7: Performance vs. Signature Chopping Func. $f(l)$ (d=6, N=100,000, z=0.5, F=64)

damage of the random disk I/Os done for accessing the S-tree nodes. Putting them together, when a somewhat large f(l) is used, *RStreeINN* can obtain a more significant performance improvement compared to *RtreeINN*.

Figure 7(c) shows the entire queue size according to $f(l)$ where $k = 10$. *NoSign* and $base > 2$ correspond to *RtreeINN* and *RStreeINN*, respectively. In the figure, the upper and lower numbers, respectively, denote the entire queue size and the number of R-tree nodes inserted into the queue during the query processing. The difference between the two numbers corresponds to the number of objects(or TIDs) enqueued during the query processing. In Figure 7(c), we find that the amount of both R-tree nodes and the objects inserted into the queue decreases with the growing degree of signature chopping, meaning that in the S-tree, the *signature chopping* lowered

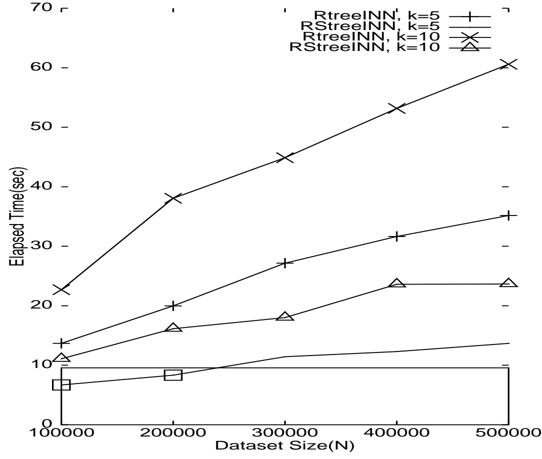


Figure 8: Execution Time with Varying Dataset Sizes

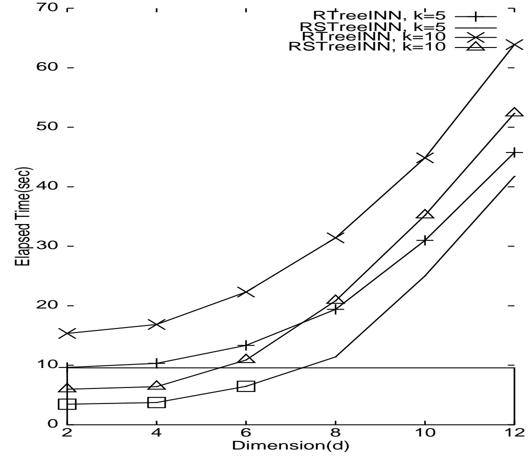


Figure 9: Execution Time with Varying Dimensions

the speed of signature saturation and hence reduced the phantom effect. This fact explains that $RStreeINN$ outperforms $RtreeINN$ as *f(l)* increases(Figure 7(a-b)).

Effect of Different Dataset Sizes(N): In this experiment, we study the performance of the two algorithms for different dataset sizes. The experimental parameter values are $d = 6, z = 0.5$, $f(l) = 6^l$, and $F = 64$. Figure 8 shows that as the dataset size is larger, $RStreeINN$ is significantly better than $RtreeINN$. Also, as the dataset size grows, the curve slope of execution time of $RStreeINN$ increases gradually, while that of $RStreeINN$ increases abruptly. Based on this fact, we know that $RStreeINN$ guarantees good performance irrespective of the dataset size.

Effect of Different Dimensions(d): The goal of this experiment is to measure the difference of the two algorithms as the number of the dimension is varied. We used the experimental parameter values: $N = 10,000$, $z = 0.5$, $f(l) = 6^l$, $F = 64$. In this experiment, in the case where $d = 2$, overflow pages for storing the signatures of the S-tree occurred and the overall storage of the S-tree was twice as large as that of the other cases(i.e., $d > 2$).

Figure 9 displays that $RStreeINN$ is better than $RtreeINN$. We also find from the figure that the difference of the performance between the two algorithms decreases as the number of the dimension grows. In fact, the capacity(or fanout) of the *high*-dimensional R*-tree node is relatively small[10]. Hence, TIDs of tuples with the query value(e.g., 'Beatles') may be evenly distributed over most R-tree leaf nodes, which consequently makes the probability of passing signature checking higher. For this reason, in high-dimensional vector spaces, $RStreeINN$ accesses a lot of R-tree nodes.

[10]The capacity of the 12-dimensional R-tree leaf node is a little under a forth times as large as that of the 2-dimensional one.

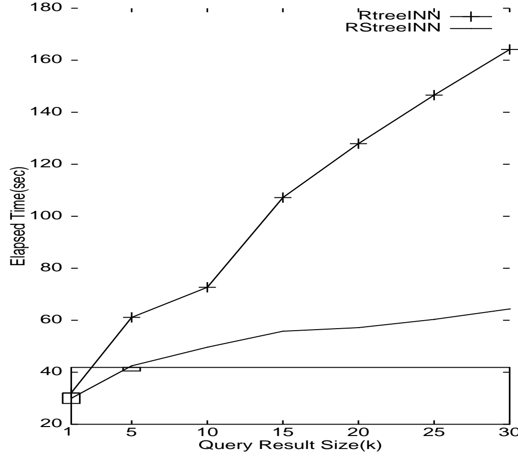


Figure 10: Execution Time for A Real Dataset(N=200,000)

6.2.2 Real Dataset

In this experiment, we study the performance of the two algorithms with respect to the query result size. As a real dataset, we used 16-dimensional Fourier points used in a CAD-model[4], which correspond to feature vectors of `color`. For other non-spatial attributes, synthetic data were generated from Zipfian Distribution with $z = 0.5$. The rest of the experimental parameter values are $N = 200,000$, $f(l) = 6^l$, and $F = 64$. We believe that comparison between the performance of the two algorithms is reasonable, although the performance of the high-dimensional R-tree is not so efficient.

Figure 10 plots the execution time versus the query result size for the two algorithms. The graphs show that $RStreeINN$ is superior to $RtreeINN$, and that the performance gap increases as the query result size grows. Therefore we know that $RStreeINN$ applying the signature chopping technique behaves favorably for the real-world dataset as it does for the synthetic datasets.

7 Conclusion

In this paper, we present the RS-tree-based incremental nearest neighbor algorithm to improve Hjaltason and Samet's algorithm based on the R-tree, when processing k-nearest neighbor queries with a non-spatial predicate. In their algorithm, since the R-tree does not have any facility to prune the tuples that would not satisfy non-spatial predicate, many unnecessary I/Os for accessing worthless tuples and R-tree nodes occur. In contrast, our algorithm can partially prune them using the RS-tree, and its performance is significantly enhanced by applying the signature chopping technique. As a result, our algorithm showed better performance than Hjaltason and Samet's algorithm, which was illustrated through our experimental results.

Acknowledgments We are grateful to the OOPSLA members, Seoul National University, especially a multimedia research team for the helpful discussions during the research. We also wish to thank S. Berchtold and C. Böhm for providing the Fourier point dataset.

References

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. *Journal of the ACM*, 45(6), November 1998.

[2] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient And Robust Access Method for Points and Rectangles. In *Proceedings of the ACM SIGMOD Conference*, June 1990.

[3] S. Berchtold, C. Bohm, B. Braunmuller, D. A. Keim, and H.-P. Kriegel. Fast Parallel Similarity Search in Multimedia Databases. In *Proceedings of the ACM SIGMOD Conference*, June 1997.

[4] S. Berchtold, C. Bohm, D. A. Keim, and H.-P. Kriegel. A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 1997.

[5] S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl. Fast Nearest Neighbor Search in High-dimensional Spaces. In *Proceedings of the 14th Int'l. Conf. on Data Engineering*, February 1998.

[6] A. J. Broder. Strategies for Efficient Incremental Nearest Neighbor Search. *Pattern Recognition*, 23(1-2), January 1990.

[7] C. Mooers. *Apllication of Random Codes to the Gathering of Statistical Information*. Bulletin 31, Zator Co., Cambridge, Mass., 1949.

[8] M. J. Carey and D. Kossmann. Reducing the Braking Distance of an SQL Query Engine. In *Proceedings of 24rd International Conference on Very Large Data Bases*, August 1998.

[9] Walter W. Chang and Hans J. Schek. A Signature Access Method for the Startbust Database System. In *Proc. of the 15th Int'l Conference on Very Large Data Bases*, August 1989.

[10] S. Chaudhuri and L. Gravano. Evaluating Top-K Selection Queries. In *Proceedings of 25rd International Conference on Very Large Data Bases*, September 1999.

[11] Ronald Fagin. Fuzzy Queries in Multimedia Database Systems. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1998.

[12] C. Faloutsos and S. Christodoulakis. Optimal Signature Extraction and Information Loss. *ACM Transactions on Database Systems*, 12(3), September 1987.

[13] S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE System for Complex Spatial Queries. In *Proceedings of the ACM SIGMOD Conference*, June 1998.

[14] A. Henrich. A Distance-scan Algorithm for Spatial Access Structures. In *Proceedings of the Second ACM Workshop on Geographic Information Systems*, December 1994.

[15] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Transactions on Database Systems*, 24(2), June 1999.

[16] J. M. Patel et al. Building a Scaleable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. In *Proceedings of the ACM SIGMOD Conference*, June 1997.

[17] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast Nearest Neighbor Search in Medical Image Databases. In *Proceedings of 22rd International Conference on Very Large Data Bases*, September 1996.

[18] M. Flickner et al. Query by Image and Video Content: The QBIC System. *IEEE Computer*, 28(9), September 1995.

[19] M. J. Carey and D. Kossmann. On Saying "Enough Already!" in SQL. In *Proceedings of the ACM SIGMOD Conference*, June 1997.

[20] V. E. Ogle and M. Stonebraker. Chabot: Retrieval from a Relational Database of Images. *IEEE Computer*, 28(9), September 1995.

[21] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proceedings of the ACM SIGMOD Conference*, May 1995.

[22] T. Seidl and H.-P. Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *Proceedings of the ACM SIGMOD Conference*, June 1998.

[23] R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of 24rd International Conference on Very Large Data Bases*, August 1998.

[24] S.B. Yao. Approximating Block Accesses in Database Organization. *Communications of the ACM*, 20(4), April 1977.

A Brief Description of the Yao Function

In order to estimate the number of block accesses in a database organization where records are grouped into blocks in secondary storage, Yao[24] presented the following theorem[11].

Theorem 1 *Let n records be grouped into m blocks($1 \leq m \leq n$), each containing $p = n/m$ records. If t records are randomly selected from the n records, the expected number of blocks hit(blocks with at least one record selected) is given by*

$$b(m,p,t) = \begin{cases} m[1 - \prod_{i=1}^{t}(n-p-i+1)/(n-i+1)] & t \leq n\text{-}p \\ m & t > n\text{-}p. \end{cases}$$

According to the derivation in Section 3.2, it is satisfied that $m = \frac{N \cdot T}{B}$, $p = \frac{B}{T}$, and $t = \frac{k}{S}$ where N, B, and T are the number of overall tuples, the page size, and the tuple size, respectively. By the above function, the tuple scan cost approximates to $C_{tuple} = b(\frac{N \cdot T}{B}, \frac{B}{T}, \frac{k}{S})$.

[11] We use the notation and some of the conditions in the paper "Kyu-Young Whang, Gio Wiederhold, and Daniel Sagalowicz. Estimating Block Accesses in Database Organizations: A Closed Noniterative Formula, *Communications of the ACM*, 26(11), Nov. 1983." which slightly modified those of [24].