Monitoring k-Nearest Neighbor Queries Over Moving Objects

Technical Report CSRG-493

Xiaohui Yu Ken Q. Pu Nick Koudas
Department of Computer Science
University of Toronto
Toronto, ON, Canada M5S 3G4
$\{$*xhyu, kenpu, koudas*$\}$@*cs.toronto.edu*

June 2004

Abstract

Many location-based applications require constant monitoring of k-nearest neighbor (k-NN) queries over moving objects within a geographic area. Existing approaches to this problem have focused on predictive queries, and relied on the assumption that the trajectories of the objects are fully predictable at query processing time.

We relax this assumption, and propose two efficient and scalable algorithms using grid indices. One is based on indexing objects, and the other on queries. For each approach, a cost model is developed, and a detailed analysis along with the respective applicability are presented. The Object-Indexing approach is further extended to multi-levels to handle skewed data.

We show by experiments that our grid-based algorithms significantly outperform R-tree-based solutions. Extensive experiments are also carried out to study the properties and evaluate the performance of the proposed approaches under a variety of settings.

1 Introduction

The prevalence of inexpensive mobile devices, such as PDA and mobile phones, and the wide availability of wireless networks give rise to a new family of spatio-temporal database applications. For example, in the location-based mixed-reality games (e.g., *BotFighters*, *BattleMachine*), players with mobile devices locate and "shoot" other participants on city streets. In the location-based commerce setting, retail stores distribute e-flyers to potential customers' mobile devices based on their location.

In this paper we consider the problem of continuously monitoring multiple k-NN queries over moving objects within a two dimensional region of interest. Such queries are important in many applications. In the gaming example, a player naturally wishes to keep track of the k near-by players in order to make a combat plan. In location-based advertising, in order to make the best use of available bandwidth, the stores may wish to send the e-flyers only to the customers who are currently closest to the store.

Given a set of objects, let $P(t) = \{p_1(t), p_2(t), \ldots, p_{N_p}(t)\}$ be their positions on a 2D region as a function of time. The set of objects is highly dynamic: each object can move in an unrestricted fashion, i.e. we do not assume any pattern of motion. For a query set $Q = \{q_1, q_2, \ldots, q_{N_Q}\}$, with each query being a static or moving point in the same region, we are interested in monitoring the *exact* k nearest neighbors (k-NNs) of each query point over time. Since we do not make any assumptions on the trajectories of the objects, exact query answering can only be done at the expense of some time-delay; i.e. at time t, the reported query answer is valid for the past snapshot of the objects and queries at time $t - \Delta t$.

The problem of continously monitoring k-NN of moving objects has been investigated in the past few years[1, 19, 13], with the emphasis on predicative or approximate query-answering by making assumptions on the motion patterns of the objects. This assumption is often violated in real applications where the objects' movements are non-predictable. Our work departs from the existing literature for that we make no assumptions on the motion of the objects, and provide the exact answers with a time delay (Δt).

In a continuous query-answering setting, the performance is gauged by the rate at which query answers are generated, and this is dictated by the time delay (Δt). In order to improve the performance (i.e. reducing Δt), we focus on designing index structures residing in main-memory.

We propose two methods to monitor k-NN queries over moving objects. The first one is based on indexing the objects themselves and we refer to it as *Object-Indexing* and the second is based on indexing the queries and thus we refer to it as *Query-Indexing*. In both methods, the index takes the form of a grid structure, which represents a canonical partition of the 2D space. The grid structure is preferable to other types of indices such as R-trees because its simple structure lends itself to fast maintenance, which is a desired property in the presence of highly dynamic data.

In the Object-Indexing method, two solutions are provided for different scenarios. When the objects are known to be uniformly distributed, an *overhaul* algorithm is used to compute the k-NNs from scratch at each cycle. However, when there is no prior knowledge of the object distribution, or when the distribution is skewed, we show that an *incremental* approach can be taken. When there are only a small number of queries, we find the Query-Indexing approach more attractive than Object-Indexing. Analysis of all indices and algorithms are provided and cost models are presented. We further extend the Object-Indexing method by using hierarchical grids to avoid performance degradation when the distribution of moving objects is skewed.

Our contributions are summarized as follows:

- Computing k-NN, in two ways, by using a dynamically updated grid-based index structure to index objects or queries.

- Generalizing the object indices to hierarchical structures. This is shown to improve query performance and robustness for skewed object distributions.
- Extending the query processing to incrementally maintain the query answers. This further improves the performance when the objects' movement is *localized*, i.e. the velocity is in general bounded.
- A theoretical analysis of the two types of indexing (object-index and query index) w.r.t. uniform and skewed object distributions. A somewhat surprising result is that the k-NN query answering can be done with constant-time (i.e. independent on the number of objects) when the objects are uniformly distributed.
- Experimental validation of the theoretical analysis and evaluation of the hierarchical and incremental extensions. We simulated on both synthetic data as well as data generated based on the Chicago area roadmap. We also performed a comparative study of our approach with R-tree and its variants, and conclude that in the highly dynamic setting, our approach out-performs tree based indexes by many folds.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 describes how to solve the k-NN search problem through object-indexing and query indexing, while Section 4 presents their hierarchical extensions. Section 5 shows the experimental results, and Section 6 concludes this paper.

2 Related Work

Answering k-nearest neighbor queries is a classical database problem. Traditionally, research on this problem has focused on static objects. Most methods use indices built on the data to assist the k-NN search. Perhaps the most widely used algorithm is the branch-and-bound algorithm[14] based on R-trees[5], which traverses the R-tree while maintaining a list of k potential nearest neighbors in a priority queue.

Besides the branch-and-bound algorithm, there have also been attempts to use range queries to solve the k-NN search problem, such as the one proposed by Korn *et al.* [10]. The basic idea is to first find a region that guarantees to contain all k-NNs, and then use a range query to retrieve the potential k-NNs. This algorithm is further extended by improving the region estimation [2], and by a better search technique of the k-NN in the region [16].

More recently, motivated by location-based applications, there has been much work on k-NN query processing for moving objects.

Lee *et al.* propose a bottom-up approach suitable for frequent updates of R-trees [11]. Most of other approaches [9, 1, 19, 13, 15, 7, 18] of answering k-NN queries for moving objects have focused on predictive queries (e.g., "*What will be the query's k-NNs 5 minutes from now?*"). A basic assumption is that the velocity of an object is known and will remain constant until it is invalidated by an update received. This implies that the predicted future trajectory of an object, computed based on its current position and velocity, is used for query answering.

Kollios *et al.* [9] were the first to consider answering k-NN queries for moving objects in 1D space. The algorithm can only be extended to 1.5-dimensions. Other existing methods that work in two or higher dimensional spaces utilize the TPR-tree (time-parameterized R-tree)[15] or its variants as the supporting index structure [1, 19, 13, 7]. Designed to support predictive queries, the TPR-tree and its variants (e.g., the TPR*-tree [21]) extend R*-tree by augmenting the indexed objects and the MBRs with velocity vectors. The k-NN queries are time-parameterized by an interval $[t_1, t_2]$ and can be answered by a depth-first traversal of the TPR-tree [1]. The problem is also studied by Tao *et al.*[19] and a repetitive approach is proposed, where conventional NN algorithms (with appropriate transformations) are applied to keep track of the next set of objects that will influence the current result. This approach requires multiple traversals of the TPR-tree, and is thus very CPU and I/O intensive. Raptopoulou *et al.*[13] propose a method that combines the merits of the previous two approaches, in which the I/O and CPU costs are significantly reduced.

All the above methods have assumed that the future trajectories of objects are known at query time (expressed by either a linear function or some recursive motion functions[18]). When this assumption does not hold, TPR-tree becomes too expensive to maintain, and is no longer a viable solution, as pointed out by Sun *et al.* [17].

Perhaps the piece of previous work most related to our research in terms of methodology is that by Kalashnikov *et al.*[8]. Their problem is to monitor range queries over moving objects. The approach is to index the queries instead of the objects by a grid structure in the memory. Our focus is on the answering of k-NN queries; for range queries, the range to be scanned for a given query is fixed, so the query index can be used without update as long as the query remains the same. However, when monitoring k-NN queries, the range that contains the k NNs cannot be pre-determined, and it constantly changes as the objects move. Therefore, new techniques must be developed to address this challenge. Choi *et al.* also utilize a 3D cell based index structure[3], but their focus is on historical queries. Therefore the design goal of the index structure is to support efficient storage and retrieval of objects' past trajectories. Their work did not consider continuous processing of k-NN queries.

3 Object and Query-Indexing

The general setup of our problem consists of a square region of interest; without loss of generality, we assume that it is the unit square $[0, 1)^2$. Mobile objects freely move in and out of the region. A set of query points in the region is given, and for each query, we periodically monitor its k nearest neighbors among the objects in the region of interest. The k-NNs of each query is recomputed repeatedly with time interval τ. We use a buffer $\mathbf{OBJ}_{\text{curr}}$ to model the current positions of the objects. This buffer is updated by the objects continuously and asynchronously, and at fixed time intervals τ, a snapshot $\mathbf{OBJ}_{\text{snapshot}}$ is taken, and the k-NNs of the queries are recomputed based on $\mathbf{OBJ}_{\text{snapshot}}$. Each query answer has a timestamp of the snapshot indicating the moment at which the answer is exact. In order to reduce the response time τ, we need to index the moving objects with some kind of indexing structure.

Since the objects are continuously moving, the index structure must be amenable to efficient maintenance. We choose to index the objects by a grid structure. Without loss of generality, we assume that all objects exist in the $[0, 1)^2$ unit square, based on some mapping of the region of interest. We introduce a grid-based index structure to facilitate k-NN queries.

Let $P(t)$ be the collection of object positions within the region of interest at time t. We assume that each object has a unique identifier $p(t) \in P(t)$, and we denote the coordinates of the object $p(t)$ by $\langle p(t)_x, p(t)_y \rangle$. The plane is partitioned into a regular grid of cells of equal size $\delta \times \delta$, and each cell is designated by its row and column indices (i, j). Very often, we need to approximate circles with rectangles in the grid, and it is convenient to define the rectangle centered at the cell $c_0 = (i_0, j_0)$ with the size l, written $R(c_0, l)$ as a rectangle with the lower left cell at $(i_0 - l, j_0 - l)$ and the upper right cell at $(i_0 + l, j_0 + l)$. The set of objects in $P(t)$ enclosed by the rectangle R is denoted by $R \cap P(t)$.

3.1 Query Answering with Object-Indexing

At any time instance t the index structure consists of each cell (i, j) having an object list, denoted by $\text{PL}(i, j)$ containing identifiers (IDs) of objects enclosed by the cell (i, j), namely $\text{PL}(i, j) = \{p(t) \in P(t) : \langle p(t)_x, p(t)_y \rangle \in [i\delta, (i + 1)\delta) \times [j\delta, (j + 1)\delta)\}$. Building the index requires scanning through the objects and inserting each object $p(t)$ into the corresponding cell $(\lfloor \frac{p(t)_x}{\delta} \rfloor, \lfloor \frac{p(t)_y}{\delta} \rfloor)$. The index structure is shown in Figure 1.

The objects' coordinates are stored in an array of length N_P. The object identifier (ID) is then the position in the array. The grid is a two dimensional array of size $\lceil \frac{1}{\delta} \rceil \times \lceil \frac{1}{\delta} \rceil$. Each cell (i, j) has a linked-list to implement $\text{PL}(i, j)$. Each query maintains an ordered list of k-objects sorted from the nearest neighbor to the furthest. For the sake of brevity, we simplify the problem by considering only a single k-NN query q positioned at $\langle q_x, q_y \rangle$ (without loss of generality and for simplicity we assume that the query is not moving). We will use the notation $k\text{NN}_q$ to refer to the k-NN of a query q and $k\text{NN}_q(t)$ to refer to the k-NN of q at a specific time instance t. It is easy to generalize the algorithm to multiple queries by repeating the same routine for each of the queries. In order to answer the query q initially we locate a rectangle R_0 centered at the cell $c_q = (\lfloor \frac{q_x}{\delta} \rfloor, \lfloor \frac{q_y}{\delta} \rfloor)$, with some size l such that R_0 encloses at least k objects from $P(t)$: $|R_0 \cap P(t)| \geq k$. Such R_0 can be found by iteratively enlarging its size l until it contains at least k objects.

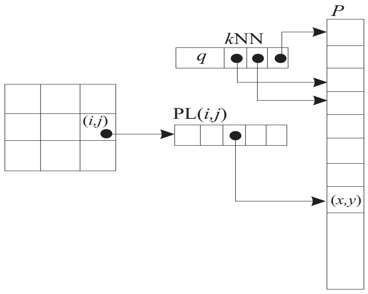


Figure 1: Data Structure of the Object-Index

The idea is that R_0 helps us to locate the k-NNs of q at time t, however, it is not guaranteed that all $k\mathrm{NN}_q(t)$ are located in R_0 as illustrated in Figure 2.

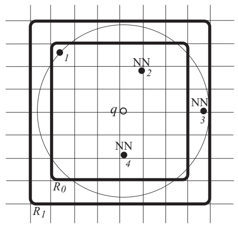


Figure 2: Identifying 3-NN for query q

Given any non-empty set of objects $P'(t) \subseteq P(t)$, define $\mathrm{far}_q(P(t)')$ to be the furthest object in $P(t)'$ from the query q. One can be certain that $k\mathrm{NN}_q(t)$ are located in the circle centered around q with radius $\|q - \mathrm{far}_q(R_0 \cap P(t))\|$ where $\| \cdot \|$ is the Euclidean norm. We approximate the circle by the larger rectangle $R_1 = R\left(c_q, \left\lceil \frac{\|q-\mathrm{far}_q(R_0 \cap P(t))\|}{\delta} \right\rceil\right)$ which is also guaranteed to contain $k\mathrm{NN}_q(t)$.

The procedure to find the k-NNs of q at time t is shown in Figure 3. Since this algorithm effectively recomputes the k-NNs at each cycle, we call it the *overhaul* algorithm.

PERFORMANCE ANALYSIS Recall (as defined in the algorithm in Figure 3), R_0 is the smallest rectangle of cells enclosing at least k neighbours of q, $l_{\text{crit}} = ||q - \mathrm{far}_q(R_0 \cap P(t))||$ is the distance from q to the farthest neighbour in R_0, and $R_{\text{crit}} = R(c_q, \lceil l_{\text{crit}}/\delta \rceil)$ is the bounding rectangle of the k-NN of q with its width and height being $2l_{\text{crit}}$.

The run-time T of computing new query answers consists of two parts $T = T_{\text{index}} + T_{\text{query}}$ where T_{index} is the time required to build the Object-Index, and T_{query} is the time of computing the k-NN using the algorithm in Figure 3.

Lemma 1. *Let N_Q be the number of queries, and N_P the number of objects. Suppose that the objects are uniformly distributed, then for the cell size $\delta \times \delta$, the run-time T is given by, $T = T_{\mathit{index}} + T_{\mathit{query}}$, where*

```
$l_0 := 0$
$R_0 := R(c_q, l_0)$
while( $|R_0 \cap P(t)| < k$ ){
   $l_0 := l_0 + 1$
   $R_0 := R(c_q, l_0)$
}
$l_{\text{crit}} := \|q - \text{far}_q(R_0 \cap P(t))\|$
$R_{\text{crit}} := R(c_q, \lceil l_{\text{crit}}/\delta \rceil)$
foreach cell $c \in R_{\text{crit}}$ {
   foreach object $p(t) \in \text{PL}(c)$ {
    insert $p(t)$ into the answer list
    of $q$ based on distance comparison
   }
}
```

Figure 3: Identifying k-NNs using Object-Indexing

$T_{index} = a_0 N_P$, and

$$T_{query} = \left(a_1 \frac{(l_{crit} + \delta)^2}{\delta^2} + a_2 (l_{crit} + \delta)^2 N_P \right) N_Q$$

for some constants a_0, a_1 and a_2.

Proof. The index is updated by a straight-forward sequential scan of the objects, so the time required is simply $T_{\text{index}} = a_0 N_P$. Let $T_{\text{1-query}}$ be the time required to compute the k-NN of a single query. Then, $T_{\text{1-query}} = a_1 \cdot (\text{ num. of cells in } R_{\text{crit}}) + a_2 \cdot (\text{ num. of objects in } R_{\text{crit}})$. The number of cells in R_{crit} is given by $4 \left\lceil \frac{l_{\text{crit}}}{\delta} \right\rceil^2 \leq 4 \frac{(l_{\text{crit}} + \delta)^2}{\delta^2}$. Under the assumption of uniform distribution of the objects, we have that the number of objects in R_{crit} is $4(l_{\text{crit}} + \delta)^2 N_P$. Substitute into the expression for $T_{\text{1-query}}$ and we arrive at the desired result. □

Theorem 1. *Suppose that the objects are uniformly distributed, the optimal cell size is $\delta^* = \frac{1}{\sqrt{N_P}}$, with the expected query time T_{query} is constant w.r.t. the number of objects N_P, and linear to the number of queries.*

Proof. The uniform distribution implies that l_{crit} is approximately the distance from the query to its k-th nearest neighbour, so $\pi l_{\text{crit}}^2 N_P \approx k$, or $l_{\text{crit}} \approx \sqrt{\frac{k}{\pi N_P}}$. Substitute l_{crit} into the expression of T in Lemma 1, and solve for the optimal cell size with $\frac{\partial T_{\text{1-query}}}{\partial \delta} = 0$, and get $\delta^* = \frac{1}{\sqrt{N_P}}$, and $T^*_{\text{1-query}} = a_3$. Therefore $T^*_{\text{query}} = a_3 \cdot N_Q$. □

Therefore, in the case of uniformly distributed objects, the total time $T = a_0 N_P + a_3 N_Q$ for some constants a_0, a_3.

Theorem 2. *With the cell size $\delta^* = \frac{1}{\sqrt{N_P}}$, and a non-uniform distribution of objects, the query time is given by*

$$T_{query} = (b_1 \mu \sqrt{N_P} + b_2 \mu^2 N_P + b_0) \cdot N_Q,$$

where $0 \leq \mu \ll 1$ is positively related to the non-uniformity of the distribution of the objects.

Sketch of proof. With non-uniform distribution, the rectangle R_{crit} no longer covers only k-NN, hence l_{crit} will be greater than the average distance ϵ from the query points to their k-th nearest neighbors. We write $l_{\text{crit}} = \epsilon + \mu$ where $\epsilon \propto \frac{1}{\sqrt{N_P}}$. Substitute l_{crit} into Lemma 1, we get $T_{\text{query}} = (b_0 + b_1 \mu \sqrt{N_P} + b2\mu^2 N_P) \cdot N_Q$. $\square$

Observe then, for reasonably uniform distributions, the query time is dominated by the $\sqrt{N_P}$, while in the worst case (highly skewed objects), it is linear to the number of objects.

3.2 Incremental Query Answering Using Object-Indexing

Let $P(t)$ be the positions of objects as a function of t and $p(t) \in P(t)$ the position of an object in $P(t)$ at time t. We assume that the objects' locations are updated simultaneously with a regular interval Δt. Let $t' = t + \Delta t$, therefore, $P(t)$ are the previous locations and $P(t')$ are the newly updated locations. One can certainly re-compute the k-NNs of the queries using the Object-Indexing technique in Section 3.1, and for uniformly distributed data, this is quite adequate.

In this section, we present an alternative of incrementally maintaining the index and the k-NNs of q.

In order to incrementally maintain the object list PL(i, j) for each cell (i, j) at time t' we scan through $p(t') \in P(t')$ and if $\lfloor p(t') \rfloor = \lfloor p(t) \rfloor$ then we do nothing, otherwise we move p from PL$(\lfloor p(t) \rfloor)$ to PL$(\lfloor p(t') \rfloor)$. While insertion of objects to PL takes constant time, deletion time is linear to the average length L of the lists. Each move of objects is done in $\mathcal{O}(L)$ time where $L \simeq N_P \delta^2$. The number of moves required after each update is Pr(exit)N_P. Here Pr(exit) is the probability of an object exiting its bounding box between the time t and $t + \Delta t$, and is determined by cell size and the mobility of the objects. If Pr(exit) is too high, incremental maintenance of Object-Index can be more expensive than rebuilding the index from scratch.

As for computing $k\text{NN}_q$ for query q, the previous answer set is used to compute the new. If we are to proceed as outlined in Section 3.1, we would require to construct R_0 iteratively. This can be avoided by constructing R_{crit} directly from the answer set of q at time t. Let $k\text{NN}_q(t)$ be the previous answer set at time t and $k\text{NN}_q(t')$ the new answer set at time t', then:

$l_{\text{crit}} := \max\{\|q - p(t')\| : p \in k\text{NN}_q(t)\}$

$R_{\text{crit}} := R(c_q, l_{\text{crit}})$

`find top $k$-NN in` $p(t') \in R_{\text{crit}} \cap P(t')$.

That is, we first find how far the previous k-NNs have moved from (l_{crit}) and draw the rectangle R_{crit} which is guaranteed to enclose the new answer set $k\text{NN}_q(t')$.

MOBILITY AND INDEX-BUILDING We assume that the motion of each object can be described as a pair (u, v) representing the horizontal and vertical displacements from time t to $t + \Delta t$, i.e. $(u, v) = p(t + \Delta t) - p(t)$. For the sake of analysis, we assume that u and v are distributed identically and independently uniformly between $[-v_{\max}, v_{\max}]$.

By mobility, we mean the probability that an object is to exit its bounding cell between time t and $t + \Delta t$. The probability that the object p stays in its bounding cell is: $\text{Pr(stays)} = \int_0^\delta \int_0^\delta \left[\int_{-x}^{\delta-x} \int_{-y}^{\delta-y} f(u)f(v)dudv\right] \cdot f(p|p \text{ in cell})dxdy$ where,

$$f(p|p \text{ in cell}) = \begin{cases} \frac{1}{\delta^2} & \text{for } p \in [0,\delta) \times [0,\delta) \\ 0 & \text{otherwise,} \end{cases} \quad ,$$

and $f(u) = \begin{cases} \frac{1}{2v_{\max}} & \text{for } u \in [-v_{\max}, v_{\max}] \\ 0 & \text{otherwise.} \end{cases}$

Carrying out the integration and setting $\text{Pr(exits)} = 1 - \text{Pr(stays)}$, we get

$$\text{Pr(exits)} = \begin{cases} 1 - \left(\frac{\delta}{2v_{\max}}\right)^2 & \text{if } \delta \le v_{\max}, \\ \frac{v_{\max}}{\delta}\left(1 - \frac{v_{\max}}{4\delta}\right) & \text{if } \delta > v_{\max}. \end{cases}$$

The probability of exiting the cell is critical in deciding how to incrementally maintain the index. In order to incrementally maintain the moving objects (from the previous cells to new cells) we require the object lists to be implemented with a sorted container such as a binary tree. The run-time is for some constant c, $T_{\text{index,incr}} = cN_P \cdot \text{Pr(exits)}(N_P\delta^2)$; this is of $\mathcal{O}(N_P{}^2)$. But building the index from scratch can be done in $T_{\text{index,init}} = c'N_P(t)$ for some constant c'. Therefore in cases of large N_P and high probability of mobility Pr(exits), it is in fact more economical to rebuild the index from scratch. Of course, in practice, depending on $v_{\max}$ and δ, it is possible for Pr(exits) to be small enough for incremental maintenance.

MOBILITY AND INCREMENTAL QUERY ANSWERING

The expression for T_{query} as shown in Lemma 1 still holds for incremental query answering. However, due to mobility, each time R_{crit} will cover more than k-NN as is in the case of non-uniform distribution of objects.

Following the same arguments found in Theorem 2, we obtain the following result.

Theorem 3. *The incremental query answering time T_{query} with the presence of mobility and cell size δ^* is*

$$T_{query} = \left(b_0 + b_1\mu\sqrt{N_P} + b_2\mu^2 N_P\right) \cdot N_Q, \tag{1}$$

where μ is positively related to mobility v_{max}.

Therefore, under imperfect estimation of l_{crit} due to mobility, the cost of query answering is of $\mathcal{O}(\sqrt{N_P})$ for small $v_{\max}$ and therefore μ, and $\mathcal{O}(N_{P(t)})$ in the worst case.

3.3 Query Answering Using Query-Indexing

Using Object-Indexing, for a sufficiently large enough number of queries, the run-time is dominated by the query-answering T_{query}, and the index maintenance T_{index} becomes insignificant. However, with very few queries and large number of objects, T_{index} becomes the dominant term. In this case, it is more advantageous to index the queries using the grid, and scan through the objects to answer the queries. It has been shown [8] that indexing queries using a grid performs well for monitoring range queries over moving objects. We adopt this approach for k-NN queries.

Each query q has a critical region $R_{\text{crit}}(q)$ which is a rectangle centered at the cell c_q that contains the query object as well as all k-NNs of q. Each cell (i, j) in the grid has a query list QL(i, j) storing all the queries whose critical region includes the cell (i, j), i.e. QL$(i, j) = \{q \in Q : (i, j) \in R_{\text{crit}}(q)\}$. The index structure is shown in Figure 4.

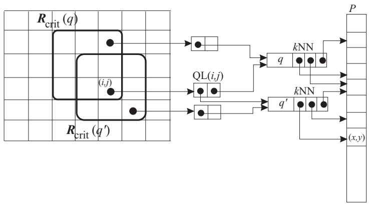


Figure 4: The index structure for Query-Indexing

Unlike the Object-Indexing approach, the Query-Index cannot be efficiently constructed from scratch; it needs a bootstrap from known k-NNs of queries. We therefore initially use Object-Indexing, to first compute the k-NNs $k\text{NN}_q(t_0)$ of all the queries at time t_0, and initialize $l_{\text{crit}}(t_0) = \|q - \text{far}_q(k\text{NN}_q(t_0))\|$, and $R_{\text{crit}}(q)(t_0) = R(c_q, \lceil l_{\text{crit}}/\delta \rceil)$. The critical region coincides with R_0 in Figure 2. At $t + \Delta t$, the critical region $R_{\text{crit}}(q)(t + \Delta t)$ can be computed from the previous answer set $k\text{NN}_q(t)$ as shown in Section 3.2. The query q is then inserted into every cell in $R_{\text{crit}}(q)(t + \Delta t)$. Again we are faced with the choice of rebuilding the query-index from scratch or incrementally updating the index. Incremental update of the index involves comparing $R_{\text{crit}}(q)(t + \Delta t)$. The query q is deleted from cells in $R_{\text{crit}}(q)(t) - R_{\text{crit}}(q)(t + \Delta t)$ and inserted to cells in $R_{\text{crit}}(q)(t + \Delta t) - R_{\text{crit}}(q)(t)$.

Once the Query-Index is updated, we compute the new k-NNs of the queries by scanning through the objects, as shown in Figure 5:

```
foreach( p(t) ∈ P(t) ) {
  (i,j) := ⌊p(t + Δt)⌋
  foreach(q ∈ QL(i,j)) {
      insert p(t) into the answer list
      of q based on distance comparison
  }
}
```

Figure 5: Identifying the k-NNs using Query-Indexing

PERFORMANCE ANALYSIS The cost of building the Query-Index is largely determined by the number of cells of the critical regions. Denote the average number of cells in $R_{\text{crit}}(q)$ by $\overline{|R_{\text{crit}}|}$. Therefore indexing all the queries takes $T_{\text{index}} = c_1\overline{|R_{\text{crit}}|}N_Q$. Query answering involves scanning through the objects and for each object $p(t)$, scanning the queries in the query list QL of the cell containing $p(t)$. Denote the average length of the query lists by $\overline{|\text{QL}|}$. This requires $T_{\text{query}} = c_2 N_{P(t)}\overline{|\text{QL}|}$. Together, we have

$$T = T_{\text{index}} + T_{\text{query}} = c_1\overline{|R_{\text{crit}}|}N_Q + c_2\overline{|\text{QL}|}N_{P(t)}.$$

Observe,

$$\overline{|\text{QL}|}/\delta^2 = \overline{|R_{\text{crit}}|}N_Q.$$

Therefore, we get $T = (c_1 + c_2\delta^2 N_{P(t)})\overline{|R_{\text{crit}}|}N_Q$, where $\overline{|R_{\text{crit}}|} = 4\left(\frac{l_{\text{crit}}+\delta}{\delta}\right)^2$, and therefore

$$T = \left(c_1\frac{(l_{\text{crit}}+\delta)^2}{\delta^2} + c_2(l_{\text{crit}}+\delta)^2 N_P\right)N_Q.$$

This expression has the same form as the query answer time for Object-Indexing in Lemma 1, though one should keep in mind the difference in the constant coefficients. One can immediately draw the conclusion that if l_{crit} can be estimated to be ϵ, the distance to the k-th nearest neighbor, one can achieve constant run-time with respect to N_P by partitioning the plane with cell size $\delta^* \propto 1/\sqrt{N_P}$. However any constant deviation μ in the estimation of l_{crit} from ϵ due to the mobility of the objects results in a complexity of $T = \mathcal{O}(\sqrt{N_P}N_Q)$.

4 Hierarchical Extension of Object-Index

Under a uniform distribution, the simple object or query-index allows very efficient processing of k-NN queries (Section 3.1). The expected time to answer N_Q queries is in fact linearly dependent on N_Q but not at all dependent on the number of objects N_P in the area of interest. This result is valid only on uniform distribution of the objects, and unsurprisingly, one cannot expect such excellent performance in practice where data sets are typically non uniform as demonstrated by Theorem 2.

To better understand the increase in the query-answering time due to non-uniformity, consider the objects and queries shown in Figure 6.

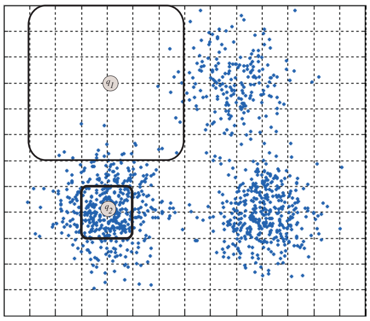


Figure 6: Some objects and two queries

Observe that query q_1 is located at a sparsely populated region, hence has a large but sparse critical region, $R_{\text{crit}1}$, while q_2 resides in a densely populated region and is with a small but dense critical region, $R_{\text{crit}2}$. Answering query q_1 requires scanning all cells in $R_{\text{crit}1}$, and in contrast, answering query q_2 requires testing all of the many objects in $R_{\text{crit}2}$. In both cases, the non-uniformity of the distribution adds cost to query answering using object-index.

Similar effects hold for query answering using the query-index. In addition, unlike object-indexing, building the query-index is also more costly for skewed data due to the increased coverage of the critical region.

HIERARCHICAL OBJECT-INDEXING To alleviate performance degradation due to non uniformity, we introduce a hierarchical object-indexing structure. Our objective is to design an index structure that can be easily built and maintained, and be adaptive to a time-varying distribution. It should also be compact in its memory footprint, having a memory requirement that is minimal and adapts to the changing object distribution. Finally, it must perform well for a wide range of distributions. The hierarchical object-index is a robust indexing structure which exhibits all of the above properties.

The hierarchical object-index is based on the simple idea that a densely populated cell should be split into sub-cells of finer size. This is a recurring theme found in Quad-trees [4] and Multi-level grid file [22]. To build the hierarchical object-index, we proceed with building a one-level object-index at some initial cell-size δ_0. Note that δ_0 is no longer dependent on N_P, and in fact should be much greater than δ^*. Given a *maximal cell load* parameter N_c and a *split factor* m, for all cells (i, j) in the one-level index, whenever the number of objects in (i, j) exceeds N_c, we split (i, j) into $m \times m$ sub-cells and push the objects into the respective lower level cells. This process is repeated iteratively until no cells contain more than N_c objects.

The resulting hierarchical object-index contains two types of cells: the leaf cells and the indexing cells. Indexing cells point to sub-grids while leaf cells store the object IDs as shown in Figure 7.

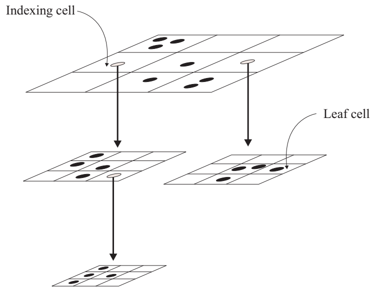


Figure 7: A three-level hierarchical object-index has the split fact $m = 3$ and maximal cell load $N_c = 3$.

The hierarchical object-index is more expensive to maintain incrementally than the one-level version. Suppose that an object p moves from $p(t)$ to $p(t')$. We first locate the leaf cells c and c' corresponding to $p(t)$ and $p(t')$. No update is necessary if $c = c'$; otherwise we need to issue a delete of p from c and an insert into c'. If c' overflows, it needs to be further divided. Also, a check is performed weather the sub-cell that c belongs to, can be collapsed back into a leaf node at the higher level. Highly skewed distributions create more levels, incurring additional cost for searching the cells c and c', and an overhaul index rebuilding may in fact be more economical.

Query-indexing also admits to a hierarchical structure. We create multiple grid levels with varying cell-size, and separate the queries into different levels based on the size of their critical regions. There are many drawbacks with this. A major issue is memory: every cell of each grid must be allocated in order to hold the query list. This can be extremely costly for the granular levels. Secondly, inherent to the query-index, query-answering must be performed incrementally, so to bootstrap the process, it must make use of an object-index initially. For these reasons, we do not consider hierarchical query-indexing further.

QUERY ANSWERING USING THE HIERARCHICAL INDEX For query answering using a hierarchical object-index, we no longer use one rectangle to represent the critical region; instead the critical region, as shown in Figure 8, consists of the largest cells enclosed by, and the smallest cells that partially overlap with, the circle centered at the query q with radius r containing at least k objects. It is not required to store in memory the cells in the critical region as, given the circle, they can be found in a top-down fashion. Initially, for each query, we repeatedly enlarge the radius r and compute the critical region until k-NN are found. Then at time t, we can incrementally maintain the query-answers by first setting $r = \|q - \text{far}_q(P(t))\|$ which guarantees that the circle contains at least the previous k NN of query q, and then proceed with a top-down computation of the critical region.

The hierarchical structure offers greater spatial resolution using few cells. In the case of queries q_1 and q_2 in Figure 6, the large critical region $R_{\text{crit}1}$ is approximated by larger cells reducing the number of cell

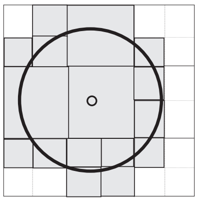


Figure 8: Representation of the critical region using two levels of a hierarchical grid

accesses, while the smaller critical region R_{crit2} is approximated by smaller cells, hence contains fewer objects.

5 Experimental Evaluation

We conducted two sets of experiments in order to (i) verify the analytical models developed in Section 3, and (ii) compare the performance of the proposed Object-Indexing (both one-level and hierarchical) and Query-Indexing algorithms with that of the R-tree-based methods, and study the properties and performance of the proposed approaches.

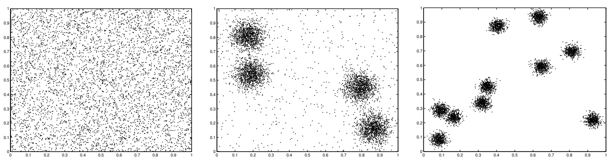
5.1 Experimental Setting

All experiments presented are carried out on a Sun SunFire V440 server running Solaris 8 with SPECInt2k rated at 703 and SPECFP2k 1054. All algorithms are implemented in C++ and compiled by gcc 3.2.

Each cycle consists of two steps: (i) updating the index structure, and (2) query evaluation. We assume that in step (i), the current positions of all objects are already available in main memory. In all experiments, the query points are assumed to be static, i.e., the same set of queries are posed to the objects at each cycle. However, we expect our methods to achieve comparable performance when query points are moving. For the overhaul algorithm, the motion of objects and queries do not have an effect on the performance, because we recompute the query result from scratch at each cycle. For incremental algorithms, the performance depends on the *relative* velocities of query points and objects. When both query points and objects move randomly, only the critical region of the queries is effected, and we expect a slight degradation in performance of the incremental query-answering.

5.2 Data Sets

We used three different synthetically generated data sets in our experiments. They contain the same number of objects but with increasing degree of skewness as shown in Figure 9. 1% of the objects in the skewed data set (Figure 9(b)) are distributed uniformly over the whole region, and the other 99% of the objects form



(a) uniform (b) skewed (c) hi-skewed

Figure 9: Data of different degrees of skewness

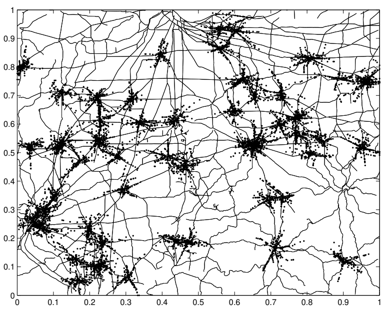


Figure 10: A snap shot of the simulation using Illinois road network data

four Gaussian clusters with randomly selected centers and standard deviation 0.05. The highly-skewed data set (Figure 9(c)) contains ten Gaussian clusters with randomly selected centers and standard deviation 0.02.

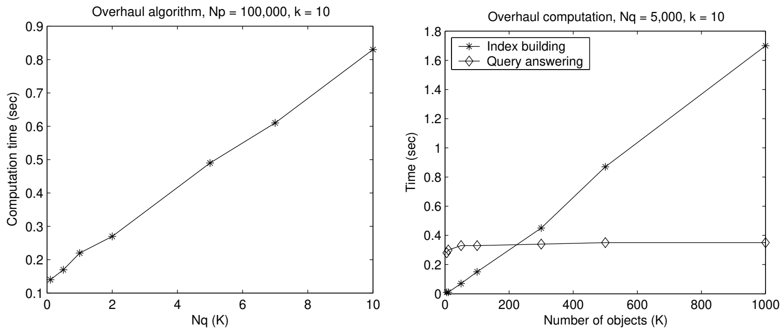
We also used a data set generated based on the road networks of the state of Illinois. Objects start near the major intersections, and then randomly move along the roads. A snap shot of the simulation is shown in Figure 10. Only 1,000 objects are shown in the figure, but up to 100,000 objects are used in the actual experiments.

5.3 Verification of Analytical Results

To verify the analytical results obtained in Section 3, we conduct experiments on data sets with the uniform data set (Figure 9(a)) and uniformly distributed queries. In all the experiments, we assume that $v_{\max}$ is 0.005 unless otherwise specified. This means that an object's displacement in both x and y directions in each cycle is uniformly distributed within the range $[-0.005, 0.005]$.

For the overhaul one-level Object-Indexing algorithm, Figure 11(a) shows that indeed the computation time is linear with respect to the number of queries N_Q, verifying our analysis. The index-building and query-answering times shown in Figure 11(b) verify our analytical expectation: building the index requires linear time with respect to N_P while query answering time is nearly constant.

In Section 3, we described an incremental index maintenance algorithm, and showed that its performance



(a) Performance of overhaul computation w.r.t N_Q

(b) Performance of overhaul computation w.r.t N_P

Figure 11: Performance of overhaul computation w.r.t N_Q and N_P

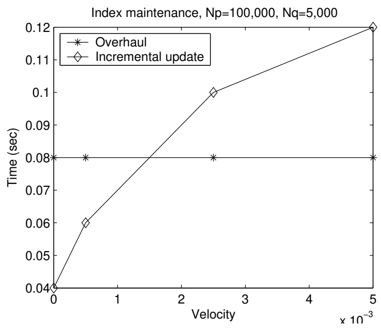


Figure 12: Overhaul v.s. incremental index maintenance

depends on v_{max} and δ. Figure 12 compares the performance of the incremental maintenance algorithm with that of the overhaul rebuild method as v_{max} varies. Naturally, the cost of a complete rebuild of the index does not depend on v_{max}, while the cost of incremental update goes up as $v_{\max}$ increases. In this experimental setting ($N_P = 100,000, N_Q = 5,000$), the cross-over point of these two methods is at around $v_{\text{max}} = 0.0015$. This implies that when the displacement of objects between consecutive cycles is relatively small, it is worthwhile to use the incremental update algorithm, while when the objects move rapidly, a complete rebuild is preferred.

Figure 13 shows the performance of incremental query answering with object-index. Note that in Equation 1, when N_P is small, the second term ($b_1\mu\sqrt{N_P}$) dominates, and therefore the cost of query answering is of $\mathcal{O}(\sqrt{N_P})$. However, as N_P increases, the third term ($b_2\mu^2 N_P$) becomes more significant, which propels the query answering cost into the order of $\mathcal{O}(N_P)$. This is confirmed in Figure 13, where the cost of query answering is of $\mathcal{O}(\sqrt{N_P})$ when the number of objects (N_P) is less than 50,000, while the cost

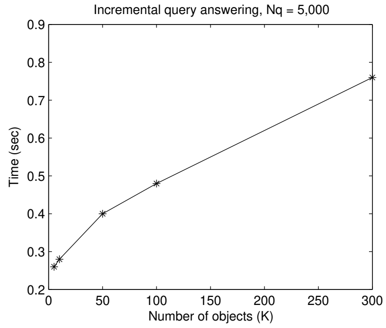


Figure 13: Performance of incremental query answering using object-index

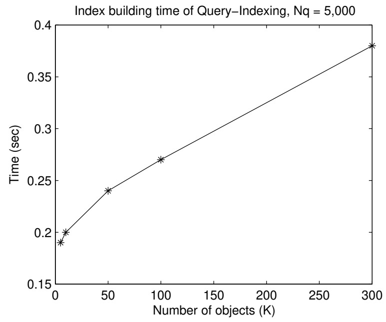


Figure 14: Index building time of Query-Indexing

increases linearly with respect to N_P for N_P greater than 100,000.

Figure 14 confirms our analysis for the index building time in Query-Indexing. The index building time shows a similar trend with respect to N_P as the query answering time does in Object-Indexing, as pointed out in Section 3.

The performance of Object-Indexing and Query-Indexing algorithms is compared in Figure 15. Clearly, for a small number of queries ($\leq 1,000$) and a large number of objects (100,000), the Query-Indexing method is advantageous because it avoids the expensive index construction involved in Object-Indexing. However, the performance of Query-Indexing degrades much faster than Object-Indexing as N_Q increases, with a cross-over point at around $N_Q = 1,000$. The reason is that Object-Indexing enjoys better locality when accessing the grid structure during query-answering. In the Object-Indexing query answering algorithm, the cells and their associated objects within a query's critical region are accessed consecutively, while in the Query-Indexing query answering algorithm, the cells have to be randomly accessed because two objects consecutively examined may fall in distant cells.

Next, we study how the performance of all methods is affected by the choice of grid cell size. As shown in Figure 16, the one-level structures reach their optimal performance when the cell size δ is chosen to be approximately $\sqrt{N_P}$, as predicted by our analysis in Section 3.1, with the constant close to 1. The

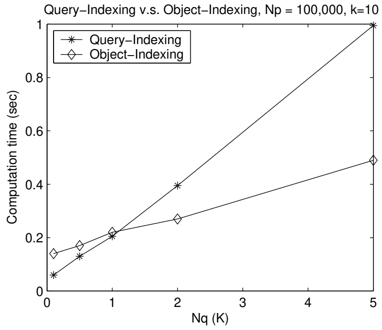


Figure 15: Query-Indexing v.s. Object-Indexing w.r.t. N_Q

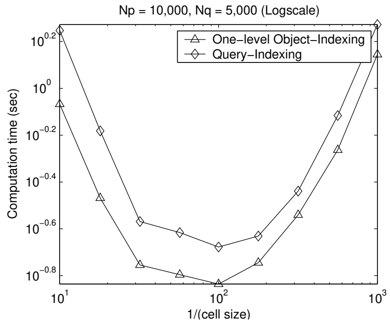


Figure 16: The effect of cell size on the performance of indices

hierarchical object-index, on the other hand, is robust with respect to the specified initial cell size.

Having verified the analytical results via experiments, we proceed to evaluate additional parameter combinations of our techniques on non-uniform data sets.

Figure 17 shows the effects of non-uniformity on the index structures ($N_P = 100,000, N_Q = 5,000, k = 10$). It is evident that one-level Object-Indexing and Query-Indexing both suffer from performance degradation on the non uniform data sets, while hierarchical Object-Indexing consistently performs well. The bottom-up R-tree algorithm is also sensitive to the data distribution.

5.4 Comparison with the R-tree and Variants

We now conduct experiments to compare the performance of our proposed algorithms with that of algorithms based on R-trees and variants. When the velocities of the objects are constantly changing, the horizon in the TPR-tree approaches 0, and the TPR-tree thus degenerates to the R-tree. Therefore, the TPR-tree is not explictly tested in the experiments, but results similar to that of R-trees are expected.

Unless otherwise specified, the following experiments are carried out using the skewed (Figure 9(b)) data set. Similar behavior was observed in all other data sets, and thus we omit these graphs for brevity. We first

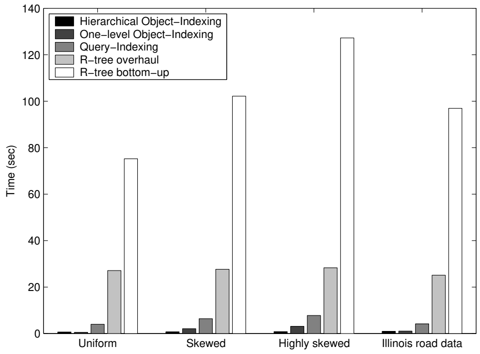


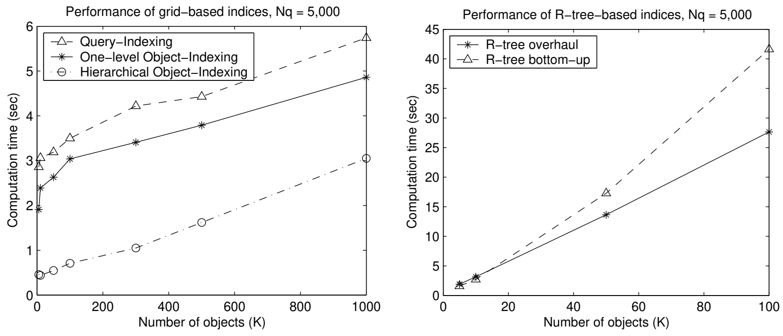
Figure 17: The effect of data skewness on the index structures

compare the performance of our proposed methods with that of the R-tree based approaches. We use the main memory R-tree implementation provided in the Spatial Index Library [6] developed at the University of California, Riverside. Two R-tree maintenance methods are utilized. The first one is the bottom-up update method proposed by Lee *et al.* [11] to support frequent updates of R-trees. The other is to simply re-construct the R-tree entirely; the motivation for this is that it is more economical to re-construct the R-tree entirely, when updating an R-tree at each cycle involves inserting and deleting a large number of objects.

PERFORMANCE W.R.T. N_p The performance of all approaches with respect to the number of objects is presented in Figure 18. For hierarchical Object-Indexing, the top-level cell size is set to be 0.1, with a maximum cell load 10 and a split factor 3.

In all settings, the proposed methods significantly outperform the R-tree, especially as the number of objects increases (a typical performance chart on a uniformly distributed data set is shown in Figure 18(b)). Due to its relatively high construction cost and low per-query cost, the R-tree is more suitable to cases where data remain constant for a certain period of time and queried many times. Our methods, on the other hand, are more applicable to situations where the data are constantly changing, because the grid structures can be cheaply constructed and updated. Similar arguments also hold for TPR-trees, except that TPR-trees are more suited to cases in which the velocities of objects remain constant over a relatively long period of time.

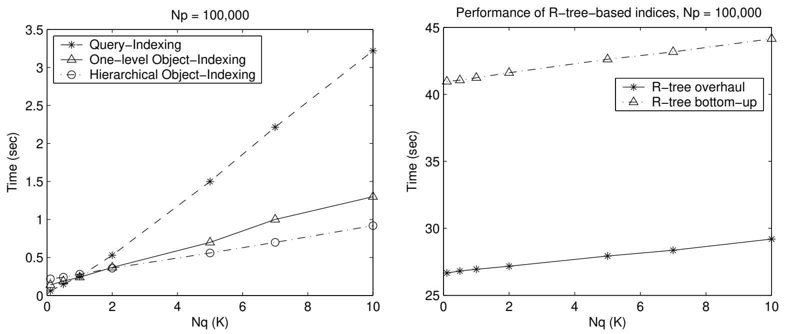
Among our proposed methods, the hierarchical Object-Indexing method performs best, with a nearly linear scalability with respect to N_P. The Query-Indexing method does not perform as well as the Object-Indexing method; this is expected because the number of queries is not small. Observe that the time-complexity of one-level Object-Indexing shifts from $\mathcal{O}(\sqrt{N_p})$ to $\mathcal{O}(N_P)$ as N_P increases, as predicted by the analysis in Section 4. For R-tree-based approaches, notice that the bottom-up update of the R-tree index outperforms a complete rebuild of the R-tree index for relatively small populations only. The reason is that in our study, the average displacement of the moving objects is independent of the population size. Therefore with larger population, there is a greater degree of volatility which results in an increasing number of out-of-bounding-rectangle updates by the bottom-up R-tree update algorithm.



(a) Grid-based approaches (b) R-tree-based approaches

Figure 18: Performance of index structures w.r.t N_P

Performance w.r.t. N_q Figure 19 shows the performance of the index structures with respect to in-



(a) Grid-based approaches (b) R-tree-based approaches

Figure 19: Performance of index structures w.r.t N_Q

creasing number of queries. Non-uniformity again makes the hierarchical approach the best choice for large number of queries. However, notice that Query-Indexing gives the best performance for small query workloads, which is expected since in these cases, the query-index is easier to build, and as shown in Section 3.3, its query-answering cost is similar to that of Object-Indexing. With small workloads, the one-level Object-Indexing outperforms its hierarchical counter-part because it has a smaller index building overhead. But in the case of larger query workloads, it is well worth the extra effort to build a hierarchical object-index. Figure 19(b) shows the performance of the R-tree-based approaches. The bottom-up update approach does not perform as well as the overhaul rebuild approach, as it has greater index maintenance cost, as shown above, and query answering cost, due to increased overlapping of the MBRs.

SCALABILITY W.R.T. k The scalability of the proposed methods with respect to k, the number of nearest neighbors, is also tested (on the skewed data set), and a typical result is presented in Figure 20. All methods scale approximately linearly with respect to k, and again, the hierarchical Object-Indexing method shows best performance for all values of k. The R-tree-based approaches are not shown in the graph because they are an order of magnitude slower than the grid-based methods.

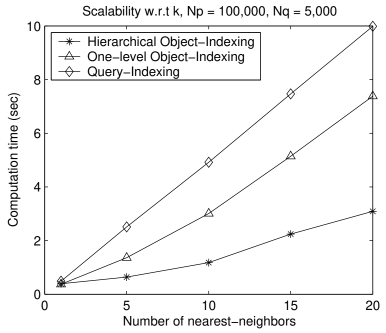
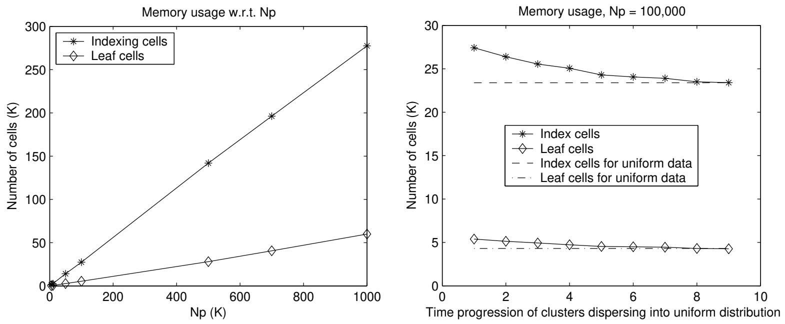


Figure 20: Performance of the grid-based index structures w.r.t k

MEMORY USAGE Since all the data structures reside in main memory, it is very important to keep the memory footprint of the index structure as small as possible. The hierarchical object-index is designed to provide robust performance while occupying a relatively small amount of memory. Figure 21(a) shows the memory requirements of the hierarchical object-index as a function of the size of the population in the region of interest. The number of index-cells and leaf-cells are both shown to be linear to the number of objects. The memory footprint of the hierarchical index is also affected by data distribution. To demonstrate this, we simulate a dispersion of four clusters into uniformly distributed objects while all the objects remain in the region. Figure 21(b) shows the memory profile of the size of the adaptively maintained hierarchical index in terms of the number of indexing cells and leaf cells. Note that the number of both index cells and leaf cells decreases as the data becomes more uniform, and converge to the same value as when uniform data is indexed.

EFFECT OF OBJECT VELOCITY Object velocity is a key factor in deciding whether one should use overhaul or incremental approaches. As one can see in Figure 22(a), the incremental maintenance of the one-level Object-Indexing outperforms the overhaul rebuild method only for very small velocities. For the hierarchical case, incremental maintenance is never the preferred choice due to the relatively high cost of index look-up for object deletion. Note that for Query-Indexing, incremental maintenance outperforms overhaul rebuild over a wide range of velocities (Figure 22(b)).

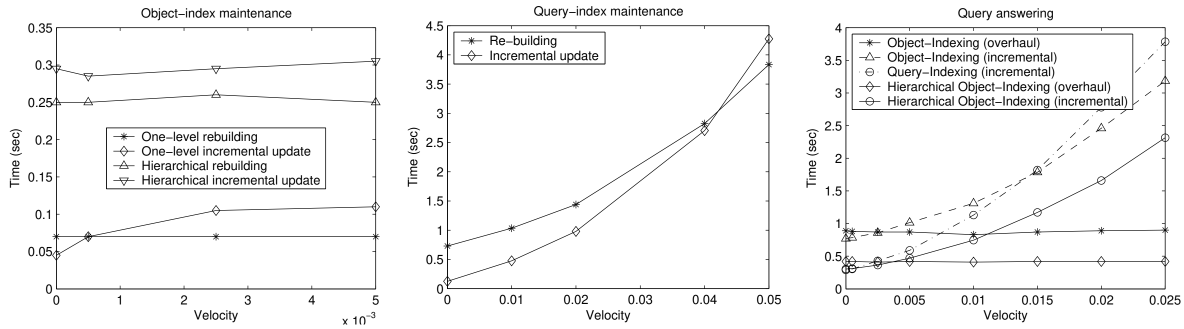
Figure 22(c) shows the effect of velocities of objects on the performance of query answering for the grid-based structures. Observe it is always better to commit to an overhaul re-computation when the velocity is



(a) Memory usage w.r.t N_P

(b) The effect of data skewness on the memory usage

Figure 21: Memory usage of the hierarchical object-index



(a) Object-index maintenance

(b) Query-index maintenance

(c) Query answering

Figure 22: The effect of velocity on index maintenance and query answering

relatively large.

6 Conclusion and Future Work

We have studied the problem of monitoring k-NN queries over moving objects. We presented an analysis of two proposed approaches namely, query-indexing and object indexing. We have validated the results of our analysis, presented extensions of the basic methods to handle non-uniform data efficiently and conducted a variety of experiments to explore the benefits of our approach in a variety of parameter settings.

For future work, we would like to extend the (hierarchical) Object-Indexing and Query-Indexing methods to handle other variations of k-NN queries, such as reverse k-NN queries[1, 20], and group NN queries[12].

We are also considering applying our algorithms in the context of spatial joins of moving objects.

Acknowledgment We would like to thank A.O. Mendelzon and K.C. Sevcik for their valuable feedback.

References

[1] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, 2002.

[2] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB*, 1999.

[3] W. Choi, B. Moon, and S. Lee. Adaptive cell-based index for moving objects. *Data Knowl. Eng.*, 48(1), 2004.

[4] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[5] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD Conference*, 1984.

[6] M. Hadjieleftheriou. Spatial Index Library version 0.62b (C++ Version), 2003. University of California, Riverside.

[7] G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, 2003.

[8] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distrib. Parallel Databases*, 15(2):117–135, 2004.

[9] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *STDM*, 1999.

[10] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In *VLDB*, 1996.

[11] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *VLDB*, 2003.

[12] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *ICDE*, 2004.

[13] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. *GeoInformatica*, 7(2):113–137, 2003.

[14] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, 1995.

[15] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, 2000.

[16] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD Conference*, 1998.

[17] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the past, the present and the future in spatio-temporal databases. In *ICDE*, 2004.

[18] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *SIGMOD Conference*, 2004.

[19] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD Conference*, 2002.

[20] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, 2004.

[21] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, 2003.

[22] K.-Y. Whang and R. Krishnamurthy. The Multilevel Grid File - a dynamic hierarchical multidimensional file structure. In *DASFAA*, 1991.