SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases[*]

Xiaopeng Xiong   Mohamed F. Mokbel   Walid G. Aref

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398
{xxiong,mokbel,aref}@cs.purdue.edu

Abstract

Location-aware environments are characterized by a large number of objects and a large number of continuous queries. Both the objects and continuous queries may change their locations over time. In this paper, we focus on continuous k-nearest neighbor queries (CKNN, for short). We present a new algorithm, termed SEA-CNN, *for answering continuously a collection of concurrent CKNN queries. SEA-CNN has two important features: incremental evaluation and shared execution. SEA-CNN achieves both efficiency and scalability in the presence of a set of concurrent queries. Furthermore, SEA-CNN does not make any assumptions about the movement of objects, e.g., the objects velocities and shapes of trajectories, or about the mutability of the objects and/or the queries, i.e., moving or stationary queries issued on moving or stationary objects. We provide theoretical analysis of SEA-CNN with respect to the execution costs, memory requirements and effects of tunable parameters. Comprehensive experimentation shows that SEA-CNN is highly scalable and is more efficient in terms of both I/O and CPU costs in comparison to other R-tree-based CKNN techniques.*

1. Introduction

The integration of position locators and mobile devices enables new location-aware environments where all objects of interest can determine their locations. In such environments, moving objects are continuously changing locations and the location information is sent periodically to spatio-temporal databases. Emerging location-dependent services call for new query processing algorithms in spatio-temporal databases. Examples of these new services include traffic monitoring, nearby information accessing and enhanced 911 services.

The continuous k-nearest neighbor (CKNN) query is an important type of query that finds continuously the k nearest objects to a query point. Unlike a snapshot KNN query, a CKNN query requires that its answer set be updated timely to reflect the motion of either the objects and/or the queries.

In a spatio-temporal location-aware database server, with the ubiquity and pervasiveness of location-aware devices and services, a large number of CKNN queries will be executing simultaneously. The performance of the server is apt to degrade and queries will suffer long response time. Because of the real-timeliness of the location-aware applications, long delays make the query answers obsolete. Thus new query processing algorithms addressing both efficiency and scalability are required for answering a set of concurrent CKNN queries.

In this paper, we propose, SEA-CNN, a *Shared Execution Algorithm* for evaluating a large set of CKNN queries continuously. SEA-CNN is designed with two distinguishing features: (1) Incremental evaluation based on former query answers. (2) Scalability in terms of the number of moving objects and the number of CKNN queries. Incremental evaluation entails that only queries whose answers are affected by the motion of objects or queries are reevaluated. SEA-CNN associates a searching region with each CKNN query. The searching region narrows the scope of a CKNN's reevaluation. The scalability of SEA-CNN is achieved by employing a *shared execution* paradigm on concurrently running queries. Shared execution entails that all the concurrent CKNNs along with their associated searching regions are grouped into a common query table. Thus, the problem of evaluating numerous CKNN queries reduces to performing a spatial join operation between the query table and the set of moving objects (the object table). By combining incremental evaluation and shared execution, SEA-CNN achieves both efficiency and scalability.

Unlike traditional snapshot queries, the most important

[*] This work was supported in part by the National Science Foundation under Grants IIS-0093116, EIA-9972883, IIS-9974255, IIS-0209120, and EIA-9983249.

issue in processing continuous queries is to maintain the query answer continuously rather than to obtain the initial answer. The cost of evaluating an initial query answer is amortized by the long running time of continuous queries. Thus, our objective in SEA-CNN is not to propose another kNN algorithm. In fact, any existing algorithm for KNN queries can be utilized by SEA-CNN to initialize the answer of a CKNN query. In contrast, SEA-CNN focuses on maintaining the query answer continuously during the motion of objects/queries.

SEA-CNN introduces a general framework for processing large numbers of simultaneous CKNN queries. SEA-CNN is applicable to all mutability combinations of objects and queries, namely, SEA-CNN can deal with: (1) Stationary queries issued on moving objects (e.g., "Continuously find the three nearest taxis to my hotel"). (2) Moving queries issued on stationary objects (e.g., "Continuously report the 5 nearest gas stations while I am driving"). (3) Moving queries issued on moving objects (e.g., "Continuously find the nearest tank in the battlefield until I reach my destination"). In contrast to former work, SEA-CNN does not make any assumptions about the movement of objects, e.g., the objects' velocities and shapes of trajectories.

The contributions of this paper are summarized as follows:

1. We propose SEA-CNN; a new scalable algorithm that maintains incrementally the query answers for a large number of CKNN queries. By combining incremental evaluation with shared execution, SEA-CNN minimizes both I/O and CPU costs while maintaining continuously the query answers.

2. We provide theoretical analysis of SEA-CNN in terms of its execution cost and memory requirements, and the effects of its tunable parameters.

3. We conduct a comprehensive set of experiments that demonstrate that SEA-CNN is highly scalable and is more efficient in terms of I/O and CPU costs in comparison to other R-tree-based CKNN techniques.

The rest of the paper is organized as follows. In Section 2, we highlight related work for KNN and CKNN query processing. In Section 3, as a preliminary to SEA-CNN, we present an algorithm for processing one single CKNN query. In Section 4, we present the general SEA-CNN algorithm to deal with a large number of CKNN queries. We present theoretical analysis of SEA-CNN algorithm in Section 5. Section 6 provides an extensive set of experiments to study the performance of SEA-CNN. Finally, Section 7 concludes the paper.

2. Related Work

k-nearest-neighbor queries are well studied in traditional databases (e.g., see [10, 14, 20, 24]). The main idea is to traverse a static R-tree-like structure [9] using *"branch and bound"* algorithms. For spatio-temporal databases, a direct extension of traditional techniques is to use branch and bound techniques for TPR-tree-like structures [1, 16]. The TPR-tree family (e.g., [25, 26, 30]) indexes moving objects given their future trajectory movements. Although this idea works well for snapshot spatio-temporal queries, it cannot cope with continuous queries. Continuous queries need continuous maintenance and update of the query answer.

Continuous k-nearest-neighbor queries (CKNN) are first addressed in [27] from the modeling and query language perspectives. Recently, three approaches are proposed to address spatio-temporal continuous k-nearest-neighbor queries [11, 28, 29]. Mainly, these approaches are based on: (1) Sampling [28]. Snapshot queries are reevaluated with each location change of the moving query. At each evaluation time, the query may get benefit from the previous result of the last evaluation. (2) Trajectory [11, 29]. Snapshot queries are evaluated based on the knowledge of the future trajectory. Once the trajectory information is changed, the query needs to be reevaluated. [28] and [29] are restricted to the case of moving queries over stationary objects. There is no direct extension to query moving objects. [11] works only when the object trajectory functions are known. Moreover, the above techniques do not scale well. There is no direct extension of these algorithms to address scalability.

The scalability in spatio-temporal queries is addressed recently in [4, 8, 12, 19, 22, 32]. The main idea is to provide the ability to evaluate concurrently a set of continuous spatio-temporal queries. However, these algorithms are limited either to stationary range queries [4, 22], distributed systems [8], continuous range queries [19, 32], or to requiring the knowledge of trajectory information [12]. Utilizing a *shared execution* paradigm as a means to achieve scalability has been used successfully in many applications, e.g., in NiagaraCQ [7] for web queries, in PSoup [5, 6] for streaming queries, and in SINA [19] for continuous spatio-temporal range queries. Up to the authors' knowledge, there is no existing algorithms that address the scalability of continuous k-nearest-neighbor queries for both moving and stationary queries by making none object trajectory assumptions.

Orthogonal but related to our work, are the recently proposed k-NN join algorithms [2, 31]. The k-nearest-neighbor join operation combines each point of one data set with its k-nearest-neighbors in another data set. The main idea is to use either an R-tree [2] or the so-called *G-ordering* [31] for indexing static objects from both data sets. Then, both R-trees or G-ordered sorted data from the two data sets are

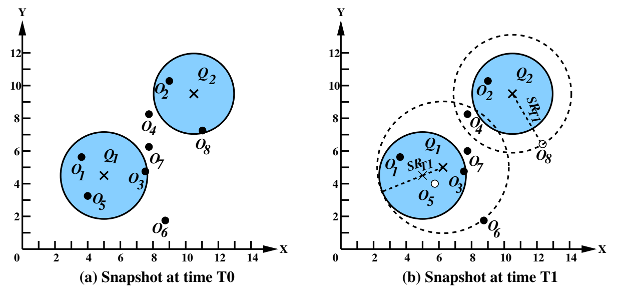


Figure 1. k-NN queries.

joined either with an R-tree join or a nested-loops join algorithm, respectively. The CKNN problem is similar in spirit to that of [2, 31]. However, we focus on spatio-temporal applications where both objects and queries are highly dynamic and continuously change their locations.

3. Processing One CKNN Query

As a preliminary to SEA-CNN, we discuss how we evaluate incrementally one CKNN query. We assume that the distance metric is the Euclidean distance. However, other distance metrics are easily applicable. We do not make any assumptions with respect to the velocity or trajectory of the moving objects and/or queries.

Ideally, the answer to a CKNN query should be updated as soon as any new location information arrives to the server. However, this approach is neither practical nor even necessary when the number of objects is large. In this paper, we adopt the same scenario as the one in [22], where the query result is updated periodically. An efficient CKNN query processing algorithm should allow a smaller time interval between any two consecutive evaluations, and consequently shortens the query response time.

A continuously running CKNN query q at time t is represented in the form $(q.Loc_t, q.k)$, where $q.Loc_t$ is the query location at time t, $q.k$ is the required number of nearest neighbors. When q is evaluated at time t, the answer $(q.A_t)$ is kept sorted based on the distance from $q.Loc_t$. Throughout the paper, we use the following definitions:
Answer radius $q.AR_t$**:** *The distance between $q.Loc_t$ and the $q.k_{th}$ NN in $q.A_t$.*
Answer region: *The circular region defined by the center $q.Loc_t$ and the radius $q.AR_t$.*
Searching radius $q.SR_t$**:** *The evaluation distance with respect to $q.Loc_t$ when q is reevaluated at time t.*
Searching region: *The circular region defined by the center $q.Loc_t$ and the radius $q.SR_t$.*

Due to the highly dynamic environment, the query answer $q.A_{t_0}$ (evaluated at time t_0) becomes obsolete at a later time t_1. The *incremental processing* algorithm associates a *searching region* $q.SR_{t_1}$ for q, based on the former *answer radius* $q.AR_{t_0}$ and the recent movements of both objects and queries. Then, only objects inside $q.SR_{t_1}$ are checked with q. To determine $q.SR_{t_1}$ we follow the following three steps:

Step 1: Check if any object (either in $q.A_{t_0}$ or not) moves in $q.AR_{t_0}$ during the time interval $[t_0, t_1]$. If this is the case, some new objects may become part of the query answer or the ordering of former K-NNs is changed. Hence, $q.SR_{t_1}$ is set to $q.AR_{t_0}$, otherwise, $q.SR_{t_1}$ is set to zero to indicate a nil searching region.

Step 2: Check if any object that was in $q.A_{t_0}$ moves out of $q.AR_{t_0}$ during the time interval $[t_0, t_1]$. If this is the case, q must be reevaluated since some objects that were out of $q.AR_{t_0}$ are candidates to be part of the current query answer. Then, $q.SR_{t_1}$ is updated to be the maximum distance from $q.Loc_{t_0}$ to the new locations of the set of objects that were in $q.A_{t_0}$ and are out of $q.AR_{t_0}$ at time t_1. If no such objects exist, $q.SR_{t_1}$ inherits the old value from Step 1.

Step 3: If q moves during the time interval $[t_0, t_1]$, i.e., $q.Loc_{t_0} \neq q.Loc_{t_1}$, $q.SR_{t_1}$ needs to be updated accordingly. There are two cases: (1) If $q.SR_{t_1}$ from Step 2 is zero, then $q.SR_{t_1}$ is set to the sum of $q.AR_{t_0}$ and the distance between $q.Loc_{t_0}$ and $q.Loc_{t_1}$. (2) If $q.SR_{t_1} \neq 0$, $q.SR_{t_1}$ is updated by adding the distance between $q.Loc_{t_0}$ and $q.Loc_{t_1}$ to the existing $q.SR_{t_1}$ from Step 2.

As a result from the *incremental processing* algorithm, $q.SR_{t_1}$ defines a minimum searching region that includes all new answer objects. A nil $q.SR_{t_1}$ indicates that q is not affected by the motion of objects/queries, thus it requires no reevaluation.

Example. Figure 1 gives an illustrative example for the *incremental processing* algorithm. At time T_0 (Figure 1a), two CKNN queries are given; Q_1 and Q_2. Q_1 is a C3NN query with an initial answer (at time T_0) as $\{o_1, o_3, o_5\}$. Q_2 is a C2NN query with an initial answer $\{o_2, o_8\}$. $Q_1.Loc_{T_0}$ and $Q_2.Loc_{T_0}$ are plotted as cross marks. The shaded regions represent $Q_1.AR_{T_0}$ and $Q_2.AR_{T_0}$. At time T_1 (Figure 1b), the objects o_5, o_8 (depicted as white points) and the query Q_1 change their locations. The objects and query change of location indicates a movement in the time interval $[T_0, T_1]$. By applying the *incremental processing* algorithm, o_5 moves inside $Q_1.AR_{T_0}$ which involves the changing of the order among $Q_1.AR_{T_0}$ (Step 1). Hence $Q_1.SR_{T_1}$ is set to $Q_1.AR_{T_0}$. Notice that there are no object movement within $Q_2.AR_{T_0}$. Thus, $Q_2.SR_{T_1}$ is set to zero at this step. In Step 2, since o_8 was in $Q_2.AR_{T_0}$ (at time T_0) and is out of $Q_2.AR_{T_0}$ (at time T_1), $Q_2.SR_{T_1}$ is set to the distance between $Q_2.Loc_{T_0}$ and the new location of o_8. $Q_1.SR_{T_1}$ inherits its value from Step 1. In Step 3, since Q_1 changes its location, $Q_1.SR_{T_1}$ is updated by adding the distance between $Q_1.Loc_{T_0}$ and $Q_1.Loc_{T_1}$ to the former $Q_1.SR_{T_1}$ from Step 2. Figure 1b plots the final $Q_1.SR_{T_1}$

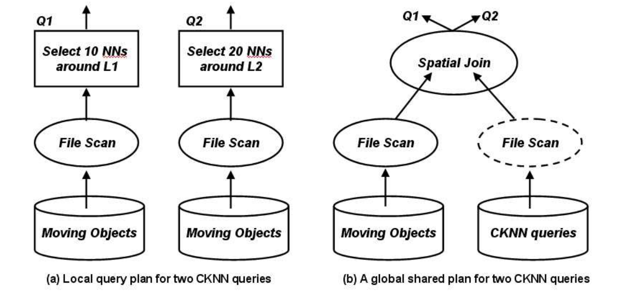


Figure 2. Shared execution of CKNN queries

and $Q_2.SR_{T_1}$ with dashed lines and dashed circles. Once we get $Q_1.SR_{T_1}$ and $Q_2.SR_{T_1}$, Q_1 and Q_2 need only to evaluate objects inside their own searching region. Moving objects that lie outside the searching region are pruned. Finally, the answer sets for Q_1 and Q_2 are $\{o_3, o_5, o_7\}$ and $\{o_2, o_4\}$, respectively (does not show in Figure 1).

4. SEA-CNN: Shared Execution Algorithm for CKNN queries

In this section, we present a *Shared Execution Algorithm* for processing a large set of concurrent CKNN queries (SEA-CNN, for short). SEA-CNN utilizes a *shared execution* paradigm to reduce repeated I/O operations. The main idea behind shared execution is to group similar queries in a query table. Then, the problem of evaluating a set of continuous spatio-temporal queries is reduced to a spatial join between the objects and queries. To illustrate the idea, Figure 2a gives the execution plans for two CKNN queries, Q_1: *"Return the 10 nearest neighbors around location $L1$"*, and Q_2: *"Return the 20 nearest neighbors around location $L2$"*. In the traditional way, one independent query execution plan is generated for each query. Each query performs a file scan on the moving objects table followed by a selection filter that is determined by the query region. With *shared execution*, a global shared plan is generated for both queries as depicted in Figure 2b. The query table contains the searching regions of CKNN queries. A spatial join algorithm is performed between the table of objects (points) and the table of CKNN queries (circular searching regions). Having a shared plan allows only one scan over the moving objects table. Thus, we avoid excessive disk operations. Once the spatial join is completed, the output of the join is split and is sent to the queries. For more details of the *shared execution* paradigm, readers are referred to [5, 6, 7, 18, 19].

SEA-CNN groups CKNN queries in a query table. Each entry stores the information of the corresponding query along with its searching region. Instead of processing the incoming update information as soon as they arrive, SEA-CNN buffers the updates and periodically flushes them into a disk-based structure. During the flushing of updates, SEA-CNN associates a searching region with each query entry. Then, SEA-CNN performs a spatial join between the moving objects table and the moving queries table.

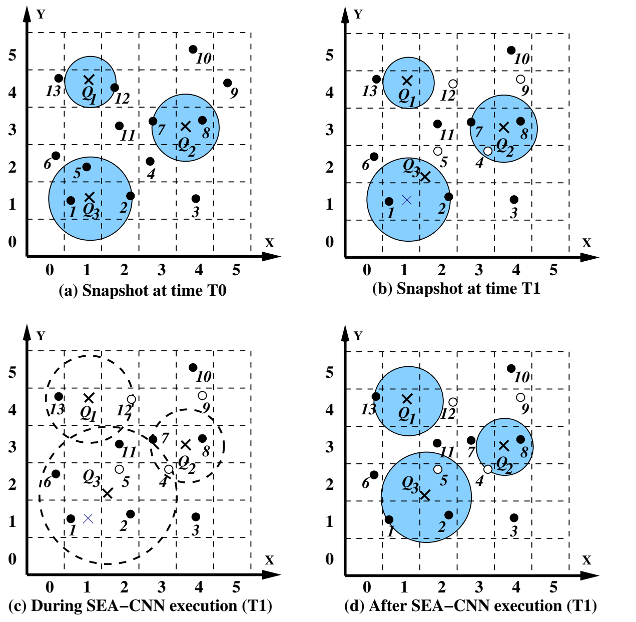

(a) Snapshot at time T0
(b) Snapshot at time T1
(c) During SEA−CNN execution (T1)
(d) After SEA−CNN execution (T1)

Figure 3. Example of execution for SEA-CNN

Throughout this section, we use the example given in Figure 3 to illustrate the execution of SEA-CNN. Figure 3a gives a snapshot of the database at time T_0 with 13 moving objects, o_1 to o_{13}, and three CKNN queries: Q_1 (C1NN), Q_2 (C2NN), and Q_3 (C3NN). The query center points are plotted as cross marks. The initial query answers at time T_0 is given in Figure 3a. The shaded regions represent the answer regions.

4.1. Data Structures

During the course of its execution, SEA-CNN maintains the following data structures:

- **Object table (OT)**. A disk-based $N \times N$ grid table. The data space is divided into $N \times N$ grid cells. Objects are hashed based on their locations to the grid cells. Every object in the space has one entry in OT where its latest information is stored. An object entry has the form of $(OID, OLoc)$ where OID is the object identifier and $OLoc$ is the latest reported location information. An index is maintained on OID for locating the cell number of each object. Data skewness in the grid structure is treated in a similar way as in [21] where the grid is partitioned to tiles that are mapped to disk either in a round-robin or hashing fashion.

- **Query table (QT)**. CKNN Queries are organized within the in-memory sequential table QT. The query entry has the form of $(QID, QLoc, k, AR, SR)$. QID is the query identifier, $QLoc$ is the latest location of the query *focal* point, k indicates the required number of nearest-neighbors, AR is the *answer radius* of the latest answer result (see Section 3), and SR is the *searching radius* (see Section 3). The query table is indexed on QID.

- **Answer Region Grid (ARG).** ARG is an in-memory $N \times N$ grid structure that has the same grid organization as OT. For each grid cell, ARG maintains a list of QIDs of queries whose *answer regions* overlap with this cell.

- **Object Buffer (OB) and Query Buffer (QB).** Both OB and QB are in-memory structures used to buffer the incoming object and query updates. OB is a $N \times N$ grid buffer as OT. Each grid cell in OB stores the update of moving objects whose new locations belong to this cell. QB is a linear buffer that stores update tuples of querying points.

For the rest of the section, we use subscripts to denote one particular cell in grid structures, e.g., $OT_{(1,4)}$, $OB_{(2,5)}$, $ARG_{(3,3)}$, where the cell order along the x-axis is ahead of the cell order along the y-axis.

Discussion. Since the number of objects is expected to be huge, the moving objects table is restricted to reside on disk. However, the moving queries table is kept in memory where the number of moving queries is much smaller than that of moving objects and each CKNN query can be represented by a small fixed entry. More importantly, the searching region of a moving query changes dynamically during the movement of objects, which suggests that only memory-based structures are suitable for the moving queries table. Notice that while the query itself is concisely represented, the query answer has potentially a large size (consider a 1000-NN query). To utilize memory resources efficiently, the query table does not keep the former query answers. Instead, only the former *answer radius* of each query is stored.

Due to the highly dynamic environment, maintaining a complex spatial index (e.g., R-tree [9]) on moving objects is not practical. Also, R-tree-like structures (e.g., the LUR-tree [15] and the FUR-tree [17]) and B-tree-like structures (e.g., [13]) that are designed to cope with frequent updates of moving objects are not scalable in the presence of a large number of continuous queries. In SEA-CNN, we use grid cells to group moving objects where the grid structure is inexpensive to maintain for its static organization. Section 6 demonstrates that SEA-CNN with the grid structure largely outperforms former R-tree-based k-nearest-neighbor algorithms.

```
Procedure SEA-CNN()
  • Repeat
      1. $T_{last}$ = current_time
      2. For every query q in QT
          (a) q.SR = 0
      3. While ((($current\_time$ - $T_{last}$)<$INTERVAL$)
         AND ($OB$ and $QB$ are not full))
          (a) If (there is an object update o)
                i. c = h(o), the cell number where o resides
               ii. Add o to $OB_c$
          (b) If (there is a query update q)
                – Add q to $QB$
      4. For (c=0;c<MAX_GRID_CELL;c++)
          (a) For each tuple o in $OB_c$
                – Call FlushingObject(o, c)
      5. For each tuple q in $QB$
          (a) Call FlushingQuery(q)
      6. For (c=0;c<MAX_GRID_CELL;c++)
          (a) Search $QT$ for queries whose non-zero search-
              ing regions overlap with cell c, denote this set of
              queries as $QSet_c$
          (b) For each object tuple o in $OT_c$
                – For each query q in $QSet_c$, if o is in q's
                  searching region, o is checked against q
      7. Send the new query answers to the users
      8. For each reevaluated query q, based on the new an-
         swer
          (a) Update q.AR and q's entries in $ARG$
```

Figure 4. Pseudo code for SEA-CNN

Example. In the example of Figure 3, OT is a 6×6 grid. Consequently, OB and ARG have the same 6×6 grid organizations. Objects o_1 to o_{13} are stored in different cells of OT. QT contains three entries for Q_1, Q_2 and Q_3, respectively. For ARG, each ARG cell keeps a list of QIDs whose answer regions overlap with this cell. For instances, $ARG_{(1,4)}$ contains the QID of Q_1 and $ARG_{(3,3)}$ contains the QID of Q_2. OB and QB are empty for now.

4.2. The SEA-CNN Algorithm

In this section, we provide the details of the SEA-CNN algorithm. Figure 4 gives the pseudo code for SEA-CNN. For each course of execution, SEA-CNN first initializes the time indicator (Step 1 in Figure 4), and the searching radius of each query to zero (Step 2 in Figure 4). For a short period of evaluation $INTERVAL$, SEA-CNN buffers the incoming update information of objects and queries before they are further processed (Step 3 in Figure 4). The main pur-

```
Procedure FlushingObject(Tuple $o_{cur}$, Cell $c_{cur}$)
  1. If $o_{old}$, the old entry of $o_{cur}$ is not found in $OT_{c_{cur}}$
     (a) Search the index on $OT$ to find the cell number $c_{old}$ where $o_{old}$ resides
     (b) Search $OT_{c_{old}}$ for $o_{old}$
  2. Else $c_{old}$ = $c_{cur}$
  3. For each $QID$ in $ARG_{c_{cur}}$
     (a) Call UpdatingSR($o_{old}$,$o_{cur}$, $QID$)
  4. For each $QID$ that is in $ARG_{c_{old}}$ and not in $ARG_{c_{cur}}$
     (a) Call UpdatingSR($o_{old}$,$o_{cur}$, $QID$)
  5. Replace $o_{old}$ with $o_{cur}$
  6. Delete $o_{cur}$ from $OB_{c_{cur}}$
```

Figure 5. Pseudo code for flushing object update

```
Procedure UpdatingSR(Tuple $o_{old}$, Tuple $o_{cur}$, QID $qid$)
  1. Search QT for qid and let q be the corresponding query entry
  2. $d_{cur}$ = distance($o_{cur}.OLoc$, $q.QLoc$)
  3. $d_{old}$ = distance($o_{old}.OLoc$, $q.QLoc$)
  4. If ($d_{cur} \leq q.AR$)
     (a) $q.SR$ = max(q.AR, q.SR)
  5. Else if ($d_{old} \leq q.AR$)
     (a) $q.SR$ = max($d_{cur}$, q.SR)
```

Figure 6. Pseudo code for updating searching region

pose of the buffering is to avoid redundant access to disk pages when flushing every single update. SEA-CNN hashes the incoming updates based on their locations into different grid cells. Later, updates in the same grid cell are flushed to disk in a batch. For each object update o, the grid cell number that o belongs to is calculated by a location-based hash function h. Then o is added to the buffer cell OB_c (Step 3(a) in Figure 4). For each query update q, q is added to QB directly (Step 3(b) in Figure 4).

When the updating time interval times out or when the memory is filled out, SEA-CNN starts to flush all the buffered updates to the moving objects table (Step 4 in Figure 4) and the moving queries table (Step 5 in Figure 4). The flushing process serves two purposes: (1) Materializing the updates of moving objects and queries into the corresponding moving object table and moving query table, respectively. (2) Determining the searching radius and searching region for each query. Figures 5, 6, and 7 give the pseudo code for the flushing process of the SEA-CNN algorithm.

```
Procedure FlushingQuery(Tuple $q$)
  1. Search QT for the old query entry Q
  2. Q.SR = Q.SR + distance(Q.QLoc, q.QLoc)
  3. Update Q.QLoc with $q.QLoc$
  4. Delete q from $QB$
```

Figure 7. Pseudo code for flushing query update

Figure 5 gives the pseudo code for flushing one object update. The algorithm starts by searching for the old entry of this object in OT. In case the old entry is not found in the current OT cell, then the index on OID is exploited to obtain the old cell number of this object (Step 1 in Figure 5). For each object update, only queries whose answer regions overlap the old cell or the current cell of the object are candidates to be *affected* queries (i.e., the query search region needs to be redetermined). The QIDs of candidate queries are kept in the corresponding ARG cells. For each candidate query, the procedure `UpdatingSR` in Figure 6 is called to determine the effects of this object update (Step 3 in Figure 5). Notice that if the object changes its cell, the same processing is required for the queries whose answer regions overlap the old cell. Possibly, the object was in some query answer whose answer region overlapped with the old cell and not with the new cell (Step 4 in Figure 5). Finally, the old entry is updated with the new location information (Step 5 in Figure 5), and the update entry is released from the object buffer (Step 6 in Figure 5).

Given one object update and a query identifier, the algorithm in Figure 6 determines the effect on the query search radius by that update. First, the corresponding query entry is obtained by searching the QID through the index on QT (Step 1 in Figure 6). Then the new distance and the old distance from the object to q are calculated (Steps 2 and 3 in Figure 6). If the new distance is less than or equal to the former $q.AR$, the object update results in either a new answer or a new order to the former answer. In this case, the search radius $q.SR$ is set to the maximum value of $q.AR$ and the existing $q.SR$ (Step 4 in Figure 6). Otherwise, if the current distance is larger than the former $q.AR$, the algorithm further checks whether this object was in q's answer or not. This checking is performed by comparing $q.AR$ with the distance between q and the former location of the object (Step 5 in Figure 6). If the object was in the answer of q, q also needs to be reevaluated since some other objects may become part of the query answer. In this case, $q.SR$ is set to the maximum value of the current distance to the object and the existing $q.SR$ value.

Figure 7 sketches the steps for flushing one query update. Basically, we search the query table for the old entry of the query (Step 1 in Figure 7). Then the searching radius SR of the query is updated in the same manner as in Section 3. Namely, the query's SR is calculated by adding the query *focal* point moving distance to the existing SR (Step 2 in Figure 7). At last, the location of the query *focal* point is updated (Step 3 in Figure 7), and the corresponding update entry is released from the query buffer (Step 4 in Figure 7).

Once updates of objects and queries are flushed, SEA-CNN performs a spatial join between the moving objects table and the moving queries table. Following the discussion in Section 3, if for any query q, the searching radius is zero, then q should not be considered for the join. So only queries with non-zero search regions are processed in the join. Step 6 in Figure 4 illustrates the execution of the joining step. For each disk-based grid cell c in OT, we join all moving objects in OT with all queries that their searching regions overlap with the grid cell c. Notice that each page in the disk-based grid cell is read only once to be joined with all overlapped queries.

Finally, after getting the new k-nearest-neighbors for all continuous queries, these query answers are sent to clients (Step 7 in Figure 4). At the end of the execution, the answer radius of each *affected* moving query and the entries in ARG are updated according to the new answer set (Step 8 in Figure 4).

Example Figure 3b gives a snapshot of the database at time T_1. From time T_0 to T_1, only objects o_4, o_5, o_9, and o_{12} change their locations. These objects are plotted with white points in Figure 3b. Additionally, Q_3 changes its location. The current query location is plotted with bold cross mark while the old query location is plotted with slim cross mark. By the end of the buffering period (Step 3 in Figure 4), OB keeps the updated tuples of o_4, o_5, o_9, and o_{12} in their corresponding cells. QB contains only the update tuple for Q_3. Figure 3c gives the execution result after flushing (Steps 4 and 5 in Figure 4). The dashed circles in Figure 3c represent the calculated SRs for Q_1, Q_2, and Q_3, respectively. For Q_1, since only the motion of o_{12} affects its SR by leaving the former answer region, Q_1's SR is calculated as the distance from Q_1 to the updated o_{12}. For Q_2, its SR is determined as its former answer region. This is because only o_4 moves in Q_2 former answer region, and no former answer object is moved outside. In the case of Q_3, the SR is first calculated as the distance from the old query point to the updated o_5 when flushing o_5 to OT (Step 4 in Figure 4). At the moment of flushing the query updates, since Q_3 moves to a new querying point, Q_3's SR is updated as the sum of the former calculated SR and the distance that Q_3 has moved (Step 5 in Figure 4). When performing the joining between OT and QT (Step 6 in Figure 4), the grid cell $OT_{(2,3)}$ is evaluated by Q_1 and Q_3 where the SRs of Q_1 and Q_3 overlap with this cell. Similarly, the cells $OT_{(3,2)}$ and $OT_{(3,3)}$ are evaluated by Q_2 and Q_3, respectively. Other grid cells are either evaluated only by one single query (e.g., the cells containing o_1, o_2, o_5, o_6, o_8, o_9, o_{12} and o_{13}) or evaluated by no query (e.g., the grid cells containing o_3 and o_{10}). Finally, Figure 3d gives the new answer sets for Q_1, Q_2 and Q_3 after one execution cycle of SEA-CNN.

5. Analysis of SEA-CNN

In this section, we analyze the performance of SEA-CNN in terms of the execution costs, memory requirements, and the effects of tunable parameters. As a dominating metric, the number of I/Os is investigated for the cost analysis. The theoretical analysis is based on a uniform distribution of moving objects and moving queries in a unit square.

Assume that there are N_{obj} moving objects in the data space while N_{qry} CKNN queries are concurrently running. The arrival rates of object updates and query updates are r_{obj} and r_{qry} per second, respectively. The size of an object entry in the object table is E_{obj} and the size of a query entry in the query table is E_{qry}. The page size is B bytes. SEA-CNN equally divides the space into G cells, and the evaluation interval of SEA-CNN takes I seconds.

I/O Cost. The number of pages in each disk-based grid cell in the moving objects table is estimated as:

$$P_{cell} = \lceil \lceil \frac{N_{obj}}{G} \rceil / \lfloor \frac{B}{E_{obj}} \rfloor \rceil$$

Hence, the total number of pages in the object table is:

$$P_{OT} = \lceil \lceil \frac{N_{obj}}{G} \rceil / \lfloor \frac{B}{E_{obj}} \rfloor \rceil * G$$

The I/O cost of SEA-CNN has two parts: (1) Flushing buffered updates from moving objects, and (2) The spatial join process between the moving objects table and the moving queries table. The I/O cost of flushing buffered updates is calculated by:

$$\begin{aligned} IO_{flush} &= 2\min(P_{OT}, C_{update} * P_{cell}) \\ &+ 2\delta * I * r_{obj} * P_{cell} \end{aligned} \quad (1)$$

where C_{update} is the number of cells that receive moving object updates during the evaluation interval. δ is the percentage of objects that change their cells in two consecutive updates. Each such object needs to search for its old entry in its former grid cell, which introduces the second part of Equation 1. The coefficient "2" is introduced to indicate one reading and one writing per flushed cell. Finding the expected value of C_{update} can be reduced to and solved by the canonical *"coupon-collecting problem"* in the field of probability theory [23]. The expected value of C_{update} is:

$$C_{update} = G[1 - (\frac{G-1}{G})^{I * r_{obj}}]$$

Assume that the average number of grid cells that overlap with a query search region is C_{SR}. Since each disk-based grid cell is read only once and is processed by all queries, the I/O cost of the spatial join in SEA-CNN is:

$$IO_{join} = \min(P_{OT}, C_{SR} * P_{cell}) \tag{2}$$

The total I/O cost of SEA-CNN is the sum of the I/O cost for flushing moving objects updates (Equation 1) and the I/O cost for the spatial join process (Equation 2), that is,

$$\begin{aligned} IO_{SEA-CNN} = & \min(P_{OT}, C_{SR} * P_{cell}) \\ & + 2\min(P_{OT}, C_{update} * P_{cell}) \\ & + 2\delta * I * r_{obj} * P_{cell} \end{aligned} \tag{3}$$

Thus, the upper bound of total I/O cost is:

$$IO_{SEA-CNN} = 3P_{OT} + 2\delta * I * r_{obj} * P_{cell} \tag{4}$$

Equation 4 indicates that the upper bound of the total I/Os for SEA-CNN is jointly decided by: (1) The total number of object pages, (2) The percentage of objects that report cell changes since last evaluation, (3) The evaluation time interval, and (4) The arrival rate of object updates. δ is affected by the velocity of objects and the size of the grid structure. r_{obj} is affected by the policy of reporting updates and the number of moving objects. However, usually the latter part of the sum is far less than the first part. In this case, we declare that the I/O cost of SEA-CNN is bounded primarily by the total number of object pages.

Memory requirements. We maintain four memory structures, namely, the query table QT, the answer region grid ARG, the object buffer OB, and the query buffer QB. Suppose that the buffer entry has the same format as its according table entry. During any evaluation interval, the memory size consumed by these structures is:

$$N_{qry}*E_{qry}+N_{qry}*C_{AR}*E_{ARG}+I*r_{obj}*E_{obj}+I*r_{qry}*E_{qry}$$

where C_{AR} is the average number of grid cells that overlap a query answer region, and E_{ARG} is the size of the entry in the answer region grid. In the above equation, the four parts represent the memory sizes consumed by QT, ARG, OB, and QB, respectively. Suppose that the available memory size is M. In a typical spatio-temporal application (e.g., location-aware environments), the number of moving queries that the server can support is determined by the following equation:

$$N_{qry} = \frac{M - I*(r_{obj}*E_{obj}+r_{qry}*E_{qry})}{E_{qry}+C_{AR}*E_{ARG}} \tag{5}$$

Equation 5 suggests that once the available memory size and the environment parameters (i.e., the size of entries (E_{obj}, E_{qry}, E_{ARG}), the arrival rates for objects and queries (r_{obj}, r_{qry})) are fixed, the number of supported queries is determined by the evaluation time interval I and the average number of grid cells that a query answer region overlaps with C_{AR}. In a specific environment and grid structure, C_{AR} is affected only by the number of the required k nearest-neighbors in a CKNN query. Thus the only independent factor to the number of supported queries is the evaluation interval I.

The evaluation interval. The evaluation interval I plays an important role in SEA-CNN. By decreasing I, Equations 3 and 4 indicate that the I/O cost and I/O upper bound for each evaluation decrease. Given fixed-size memory, Equation 5 shows that a smaller I also enables a larger number of concurrent queries as we consider memory availability.

During a period of time T, the total I/O cost is given by:

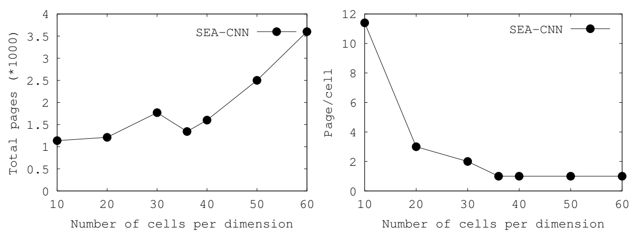
$$IO_T = I/O_{SEA-CNN} * \lfloor \frac{T}{I} \rfloor \tag{6}$$

By combining Equations 4 and 6, we observe that a short interval incurs a larger total cost on the long run. When the interval is too short, the system may not sustain it because the processing may not terminate in the interval time. To the contrary, choosing a long evaluation interval enables a smaller total cost on the long run. However, the cost at each evaluation round increases. Additionally, the number of supported queries for a given memory size decreases. Following the above observations, the interval parameter must be tuned carefully according to the application requirements and system configurations.

6. Performance Evaluation

In this section, we evaluate the performance of SEA-CNN with a set of CKNN queries. We compare SEA-CNN with a variant of the traditional *branch and bound* k-nearest-neighbor algorithm [24]. The branch-and-bound R-tree traversal algorithm evaluates any k-nearest-neighbor query by pruning out R-tree nodes that cannot contain a query answer. To be fair in our comparison, we apply the kNN algorithm in [24] to deal with the Frequently Updated R-tree (FUR-tree, for short) [17]. The FUR-tree modifies the original R-tree and efficiently handles the frequent updates due to the moving objects. We refer to the branch-and-bound kNN algorithm combining with FUR-tree by the *FUR-tree approach*. To evaluate CKNN queries, the FUR-tree approach updates continuously the FUR-tree for moving objects, and evaluates periodically every query (perhaps with new query *focal* points) against the FUR-tree.

The remaining of this section is organized as follows. First, Section 6.1 describes our experimental settings. In Section 6.2, we study the scalability of SEA-CNN in terms



(a) Total Page (b) Pages/Cell

Figure 8. The impact of grid size

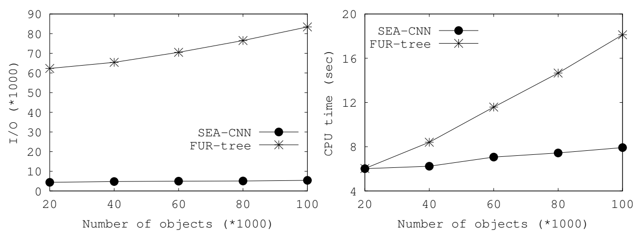
of the number of objects and the number of queries. In Section 6.3, we study the performance of SEA-CNN under various mutability combinations of objects and queries. Finally, Section 6.4 studies the performance of SEA-CNN while tuning some performance factors (e.g., the number of neighbors, velocity of objects).

6.1. Experimental Settings

All the experiments are performed on Intel Pentium IV CPU 3.2GHz with 512MB RAM. We use the *Network-based Generator of Moving Objects* [3] to generate a set of moving objects and moving queries. The input to the generator is the road map of Oldenburg (a city in Germany). Unless mentioned otherwise, the following parameters are used in the experiments. The set of objects consists of 100,000 objects and the set of queries consists of 10,000 CKNN queries. Each query asks for ten nearest-neighbors, i.e., k=10 for all queries. When normalizing the data space to a unit square, the default velocity of objects is equal to 0.000025 per second[1]. The evaluation interval of SEA-CNN is set to 30 seconds, which we call one *time step*. At each time step, some objects and queries change their locations. The default moving percentage is set as 10%. In all experiments, we compare both the number of I/Os and the CPU time for SEA-CNN and the FUR-tree approach. For the FUR-tree approach, the cost has two parts: updating the FUR-tree and evaluating the queries. The page size is 4096 bytes. Consequently, the fan-out of the FUR-tree node is 256. An LRU buffer of 20 pages is used. The first two levels of the FUR-tree reside in main memory.

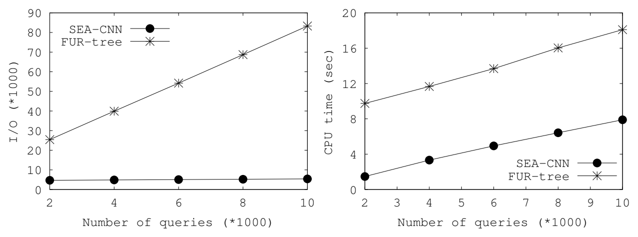
An important parameter for the SEA-CNN performance is the grid size (i.e., the number of grid cells) for the moving objects table. If the grid size is too small (e.g., less than 10×10), each grid cell contains a large number of disk pages, which incurs unnecessary I/O and CPU cost when a

1 This velocity corresponds to 90 miles per hour if the unit square represents area of 1000×1000 miles^2



(a) I/O (b) CPU Time

Figure 9. Scalability with number of objects



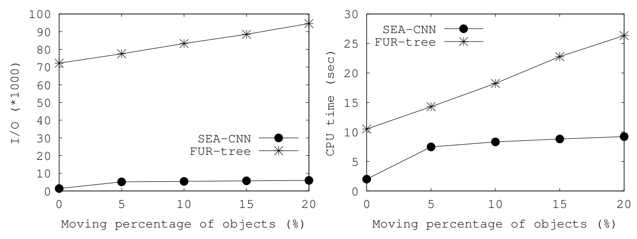
(a) I/O (b) CPU Time

Figure 10. Scalability with number of queries

cell is touched by a query. If the grid size is too large (e.g., larger than 60×60), each cell is under-utilized for containing only a few tuples, which results in an excessive number of disk pages. For the ideal choice of grid size, each cell should contain only one page that is reasonably utilized. For our experimental setting (100,000 moving objects), Figure 8a gives the number of total grid pages with different grid sizes. Figure 8b gives the number of pages per cell with different grid sizes. Figure 8 suggests that the optimal grid size is 36×36, so this grid size is chosen for the rest of our experiments.

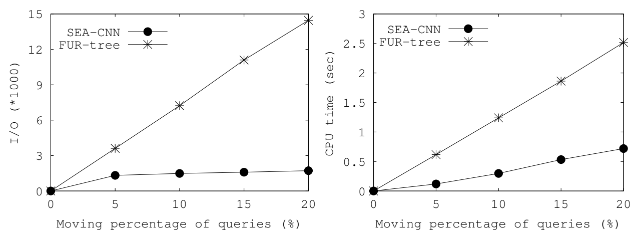
6.2. Scalability

In this section, we compare the scalability of SEA-CNN with the FUR-tree approach in terms of the number of objects and the number of queries. Figure 9 gives the effect of increasing the number of objects from 20K to 100K, given the presence of 10K queries. On the other hand, Figure 10 gives the effect of increasing the number of queries from 2K to 10K, given 100K objects. From Figure 9, the FUR-tree approach incurs an increasing number of I/Os and CPU time where a large number of moving objects results in a larger-size R-tree. Thus, each CKNN query needs to search



(a) I/O (b) CPU Time

Figure 11. Stationary queries on moving objects



(a) I/O (b) CPU Time

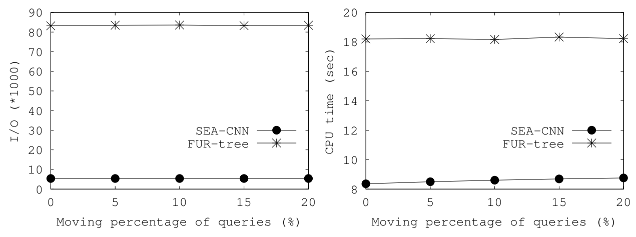
Figure 12. Moving queries on stationary objects

more nodes before getting a complete answer. Figure 9a illustrates that one query needs to search at least 6 disk pages even in the case of only 20K objects, which results in a large number of I/Os. However, for SEA-CNN, the number of I/Os is not affected apparently by the number of objects. The reason is that the number of I/Os is determined primarily by the grid size. For the CPU time, SEA-CNN has much slower increase rate than that of the FUR-tree and outperforms the FUR-tree approach in all cases. In terms of the scalability with the number of queries, Figure 10 demonstrates that SEA-CNN largely outperforms the FUR-tree approach in both I/O and CPU time. The number of I/Os in SEA-CNN is nearly stable, and is an order of magnitude less than that of the FUR-tree approach. The FUR-tree approach increases in a sharp slope where each single CKNN query independently exploits the FUR-tree. The CPU time of the FUR-tree is 2 to 5 times higher than that of SEA-CNN. The reason is that the cost of updating the FUR-tree as well as the cost of evaluating queries are much higher than that of SEA-CNN.

6.3. Mutability

In this section, we evaluate the performance of SEA-CNN and the FUR-tree approach under various mutability combinations of objects and queries. Figure 11 gives the performance when 10K stationary queries are issued on 100K moving objects. For each evaluation time, the percentage of moving objects varies from 0% to 20%. In all I/O cases, SEA-CNN outperforms the FUR-tree approach by one order of magnitude. The FUR-tree approach has a sharp increase in the number of I/Os where each moving object incurs I/O operations when updating the FUR-tree. However, SEA-CNN groups updates and flushes them in batches. Hence, SEA-CNN maintains a stable number of I/Os. For similar reason, SEA-CNN outperforms the FUR-tree approach in CPU time.

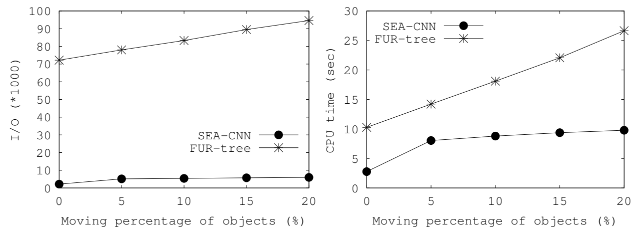
Figure 12 gives the performance when 10K moving queries are issued on 100K stationary objects. For each evaluation time, the percentage of moving queries varies from 0% to 20%. Since the objects are stationary, only the queries that move need to be reevaluated in SEA-CNN and the FUR-tree approach. Again, SEA-CNN outperforms the FUR-tree approach in all cases. Comparing to the FUR-tree approach, SEA-CNN saves a large number of I/O operations because each object page is read only once for all queries. SEA-CNN outperforms in CPU cost, since for any query, only the object cells that overlap the query search region are evaluated. The number of evaluated objects is smaller than that of the FUR-tree approach, where R-tree nodes overlap with each other.



(a) I/O (b) CPU Time

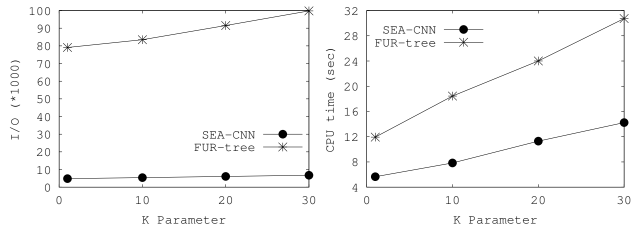
Figure 13. Moving queries on moving objects

Figures 13 and 14 give the performance of SEA-CNN and the FUR-tree approach when 10K moving queries are running on 100K moving objects. In Figure 13, the percentage of moving objects is fixed at 10%, while the percentage of moving queries varies from 0% to 20%. In this case, the number of I/Os and CPU time of the FUR-tree approach are constantly high. The reason is that the FUR-tree approach requires that each query exploits the FUR-tree regardless of whether the query moves or not. SEA-CNN demonstrates a constant and low I/O cost since the shared execution paradigm shares object pages among queries. SEA-CNN has a slight increase in CPU time where more queries



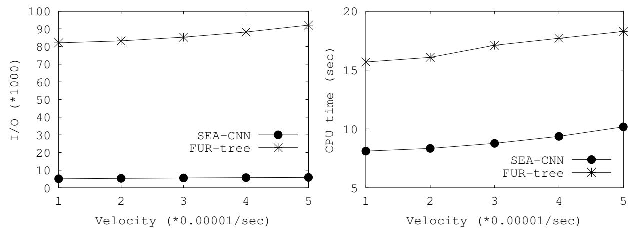
(a) I/O (b) CPU Time

Figure 14. Moving queries on moving objects



(a) I/O (b) CPU Time

Figure 15. Number of neighbors



(a) I/O (b) CPU Time

Figure 16. Velocity of objects

are reevaluated when the percentage of moving queries increases.

In Figure 14, the percentage of moving queries is fixed at 10%, while the percentage of moving objects varies from 0% to 20%. With the increase in object updates, the FUR-tree approach receives a large number of I/Os from both the increasing cost of updating FUR-tree and the constantly high cost of evaluating queries. Without surprise, SEA-CNN still achieves stable low number of I/Os. In this case, the CPU time performance is similar to the situation when stationary queries are issued on moving objects. Compared to Figure 11b, the only difference is that the cost for SEA-CNN is slightly higher by evaluating some more queries, however, the difference is trivial comparing to the high cost of the FUR-tree approach.

6.4. Affecting Factors

In this section, we study the effect of various factors on the performance of SEA-CNN. We consider two factors, namely, the number of nearest neighbors and the velocity of moving objects. Figure 15 gives the performance when the number of required nearest neighbors for each query varies from 1 to 30. Figure 15a illustrates that the number of I/Os of SEA-CNN keeps constantly low under all cases, while the number of I/Os of the FUR-tree approach is high and shows an increasing trend. The reason is that when more NNs are required, the searching region extends in both SEA-CNN and FUR-tree approach. However, SEA-CNN avoids repeated I/O by sharing object pages. Moreover, SEA-CNN outperforms in CPU time as given in Figure 15b.

Another factor that affects the performance of SEA-CNN is the velocity of moving objects. When the velocity of objects increases, for the FUR-tree approach, more objects change from their original R-tree nodes to new nodes, which involves more I/O update operations. SEA-CNN avoids the increase in I/O by buffering updates and grouping them based on cell locality. The only increase of I/O in SEA-CNN is the additional I/Os when searching the old entries of cell-changing objects. However, this increase is still very small, compared to the number of I/Os of the FUR-tree approach. Figure 16a gives the I/O comparison when the velocity of objects increase from 0.00001 to 0.00005. Figure 16b gives the CPU cost comparison under the same setting. While both the CPU time for SEA-CNN and the FUR-tree approach increase, the CPU time of SEA-CNN is only about one third of that of the FUR-tree approach.

7. Conclusion

In this paper, we investigate the problem of evaluating a large set of continuous k-nearest neighbor (CKNN) queries in spatio-temporal databases. We introduce the *Shared Execution Algorithm* (SEA-CNN, for short) to efficiently maintain the answer results of CKNN queries. SEA-CNN combines incremental evaluation and shared execution to minimize the costs when updating the query answers. With incremental evaluation, only queries affected by the motion of objects are reevaluated. To minimize the evaluation time, each affected query is associated with a searching region based on its former query answer. Under the shared execution paradigm, concurrent queries are grouped in a common query table. Thus the problem of evaluating multiple queries is solved by a spatial join between the query table and the object table. Furthermore, SEA-CNN is a generally applicable framework. First, SEA-CNN does not make any assumptions about the movement of objects (e.g., velocities, trajectories). Second, SEA-CNN is suitable for processing moving/stationary queries issued on moving/stationary objects. We provide theoretical analysis of SEA-CNN in terms of the execution costs, the memory requirements and the effects of tunable parameters. Comprehensive experiments illustrate that SEA-CNN is highly scalable and is more efficient than R-tree-based CKNN techniques in terms of both the number of I/Os and the CPU cost.

References

[1] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *IDEAS*, 2002.

[2] Christian Bohm and Florian Krebs. The k-Nearest Neighbor Join: Turbo Charging the KDD Process. In *Knowledge and Information Systems (KAIS), in print*, 2004.

[3] Thomas Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2), 2002.

[4] Ying Cai, Kien A. Hua, and Guohong Cao. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. In *Mobile Data Management, MDM*, 2004.

[5] Sirish Chandrasekaran and Michael J. Franklin. Streaming Queries over Streaming Data. In *VLDB*, 2002.

[6] Sirish Chandrasekaran and Michael J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB Journal*, 12(2):140–156, 2003.

[7] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

[8] Bugra Gedik and Ling Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2004.

[9] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[10] Gsli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems , TODS*, 24(2), 1999.

[11] Glenn S. Iwerks, Hanan Samet, and Kenneth P. Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *VLDB*, 2003.

[12] Glenn S. Iwerks, Hanan Samet, and Kenneth P. Smith. Maintenance of Spatial Semijoin Queries on Moving Points. In *VLDB*, 2004.

[13] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *VLDB*, 2004.

[14] Norio Katayama and Shinichi Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *SIGMOD*, 1997.

[15] Dongseop Kwon, Sangjun Lee, and Sukho Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Mobile Data Management, MDM*, 2002.

[16] Iosif Lazaridis, Kriengkrai Porkaew, and Sharad Mehrotra. Dynamic Queries over Mobile Objects. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2002.

[17] Mong-Li Lee, Wynne Hsu, Christian S. Jensen, and Keng Lik Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, 2003.

[18] Mohamed F. Mokbel, Walid G. Aref, Susanne E. Hambrusch, and Sunil Prabhakar. Towards Scalable Location-aware Services: Requirements and Research Issues. In *Proceedings of the ACM symposium on Advances in Geographic Information Systems, ACM GIS*, 2003.

[19] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, 2004.

[20] Apostolos Papadopoulos and Yannis Manolopoulos. Performance of Nearest Neighbor Queries in R-Trees. In *ICDT*, 1997.

[21] Jignesh M. Patel and David J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD*, 1996.

[22] Sunil Prabhakar, Yuni Xia, Dmitri V. Kalashnikov, Walid G. Aref, and Susanne E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Trans. on Computers*, 51(10), 2002.

[23] Sheldon Ross. *A First Course in Probability*. Pearson Education, 6 edition, 2001.

[24] Nick Roussopoulos, Stephen Kelley, and Frederic Vincent. Nearest Neighbor Queries. In *SIGMOD*, 1995.

[25] Simonas Saltenis and Christian S. Jensen. Indexing of Moving Objects for Location-Based Services. In *ICDE*, 2002.

[26] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.

[27] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and Querying Moving Objects. In *ICDE*, 1997.

[28] Zhexuan Song and Nick Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*, 2001.

[29] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous Nearest Neighbor Search. In *VLDB*, 2002.

[30] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB*, 2003.

[31] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jin Hu. Gorder: An Efficient Method for KNN Join Processing. In *VLDB*, 2004.

[32] Xiaopeng Xiong, Mohamed F. Mokbel, Walid G. Aref, Susanne Hambrusch, and Sunil Prabhakar. Scalable Spatio-temporal Continuous Query Processing for Location-aware Services. In *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, 2004.