Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Monitoring nearest neighbor queries with cache strategies*

PAN Peng, LU Yan-sheng‡
(*College of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*)
E-mail: panppl@163.com; lys@mail.hust.edu.cn
Received Oct. 13, 2006; revision accepted Dec. 29, 2006

Abstract: The problem of continuously monitoring multiple *K*-nearest neighbor (*K*-NN) queries with dynamic object and query dataset is valuable for many location-based applications. A practical method is to partition the data space into grid cells, with both object and query table being indexed by this grid structure, while solving the problem by periodically joining cells of objects with queries having their influence regions intersecting the cells. In the worst case, all cells of objects will be accessed once. Object and query cache strategies are proposed to further reduce the I/O cost. With object cache strategy, queries remaining static in current processing cycle seldom need I/O cost, they can be returned quickly. The main I/O cost comes from moving queries, the query cache strategy is used to restrict their search-regions, which uses current results of queries in the main memory buffer. The queries can share not only the accessing of object pages, but also their influence regions. Theoretical analysis of the expected I/O cost is presented, with the I/O cost being about 40% that of the SEA-CNN method in the experiment results.

Key words: *K*-nearest neighbors (*K*-NNs), Continuous query, Object cache, Query cache
doi:10.1631/jzus.2007.A0529 **Document code:** A **CLC number:** TP311

INTRODUCTION

Early work on the *K*-nearest neighbor (*K*-NN) query in spatial databases focuses on how to find the *K* ($K \geq 1$) closest (according to Euclidean distance) objects from the object dataset to a static query point at the query time, which aims at snapshot queries. Recently, people become interested in continuously monitoring multiple long running *K*-NN queries, which is valuable for many location-based applications. In this situation both the object and query points can move.

The Q-index (Prabhakar *et al*., 2002) is the first method that builds an index of queries to monitor them, with each object being assigned a safe-region according to the query index, and the I/O cost being reduced by only checking moving objects against their safe-regions. The CES-based index (Wu *et al*., 2006) is also a query indexing method to minimize the total (re)evaluation time of multiple queries. Although these two methods are not for *K*-NN queries, the idea of indexing queries is illuminative for monitoring multiple queries. The SEA-CNN (Xiong *et al*., 2005) method improves the method of SINA (Mokbel *et al*., 2004) for multiple *K*-NN queries, where each query dynamically maintains a search-region and is solved by range query of this region in every processing cycle; a spatial join between the object and query tables is used, so that accessing of objects is shared by queries, which aim at reducing the I/O cost for multiple queries. The YPK-CNN (Yu *et al*., 2005) and CPM (Mouratidis *et al*., 2005a) methods use grid-based index structure for objects or queries, with objects being preserved in main memory, and their purpose is to minimize the CPU cost of each query. Although CPM can be used to reduce the I/O cost of static queries, the accessing of objects cannot be shared by multiple queries. The most recent work on multiple *K*-NN queries (Iwerks *et al*., 2006; Gedik *et al*., 2006) uses a kinematic representation to describe the location of moving objects as a function of time, which need the velocity vector of each moving object to be known *a priori*.

‡ Corresponding author
* Project (No. ABA048) supported by the Natural Science Foundation of Hubei Province, China

In this paper, we present a framework for monitoring multiple spatial *K*-NN queries, which are maintained periodically with time interval Γ_p, and we need not know the objects' velocity vectors *a priori*. For large object dataset (more than 1 000 000 objects), preserving it in main memory needs a large buffer, while the buffer needs to preserve the dynamic information of moving objects and query results to respond to data updating quickly, so we assume the dataset is preserved on disk, and our purpose is to reduce the total I/O cost of accessing objects from disk. We deal with *insert*, *delete* and *location-update* operations on both object and query point datasets, and multiple operations on each object during a processing cycle are combined by our algorithm. Like the SEA-CNN algorithm, queries have their influence regions, so that they can share the accessing of object pages.

In many real life applications, although the dataset is dynamic, for a relative long time compared with monitoring intervals, the distribution of objects is almost static (e.g., the distributions of population and vehicles in a city would not change for several hours, while the system refreshes the query results at every minute). Basing on this assumption, we propose the object cache strategy for static queries and query cache strategy for moving queries. The basic idea of our object cache strategy is to monitor changes within a small region for each static query, because the region probably contains the query point's *K*-NN answer for a relatively long time. This can be done only with the operation buffer and the object cache of the query. With the query cache strategy for moving query points, queries share not only the accessing of object pages, but also their influence regions, which can effectively restrict each query's influence region.

PREVIOUS METHODS

We briefly describe the SEA-CNN (SEA for short) method in this section, because to our knowledge, it is the most competitive method for disk-resident data while monitoring multiple *K*-NN queries. The SEA method partitions the whole data space into grid cells which organize most data structures. The object table (OT) is preserved in disk, and divided by spatial girds. The main memory preserves a query table (QT: storing each query's ID, location, *K* value, radius of the answer and searching regions), an answer region grid (ARG: storing each grid cell's list of queries whose answer region overlaps this cell), an object grid buffer (OB: each cell in OB preserves moving objects whose new locations belong to this cell) and a query buffer (QB: storing moving queries' IDs). Basing on these data structures, SEA achieves incremental evaluation and scalability in terms of the number of moving objects and queries, which reduces the I/O cost.

For each moving object p in OB (p currently belongs to cell C_{cur} in OB), SEA finds the last cell C_{old} that p resided in; then, according to ARG and the motion of p, it updates the search-regions of all queries that belong to C_{cur} or C_{old}. This entails an incremental evaluation from former query answers: only queries whose answers are affected by moving objects are assigned non-zero search-regions with further reevaluation. Then, for each moving query q in QB, SEA updates q's search-region according to its moving distance. At last, for each grid cell, objects in it are accessed from disk to evaluate queries whose non-zero searching regions overlap this cell. It looks like a nested-loop join between the grid based OT and QT, which achieves a shared execution paradigm on concurrently running queries. The detailed algorithm can be seen in (Xiong *et al*., 2005).

In (Mouratidis *et al*., 2005b), a thorough checking in current queries' result object lists is performed to initially find some closest objects for a moving query, which is to reduce its searching region. This is a further optimization for moving queries, while the authors aim at minimizing the communication cost between the server and the moving object. We apply and simplify such idea in our query cache strategy to reduce the I/O cost.

FRAMEWORK OF QUERY PROCESSING

Since we just evaluate all queries at time instances with interval Γ_p, we can combine multiple operations on one object without affecting the query results. We give out the rules of combining all possible two consecutive operations as follows:

$$insert + update \rightarrow insert, \quad (1)$$

$$insert + delete \rightarrow \varnothing, \quad (2)$$

$$update + update \rightarrow update, \quad (3)$$

$$update + delete \rightarrow delete. \quad (4)$$

We do not consider the situation '*insert*+*delete*' because we treat the object as a different one when it is inserted again. Applying these rules repeatedly for each operation, we can have only one operation left for each object during each processing cycle, the combination can reduce the workload of maintaining the queries.

We discuss our framework in 2D space. Each object point p has a unique identifier $p.ID$. Similarly, each query point q has a unique identifier $q.ID$. Without loss of generality, we assume that all objects and query points exist in the $[0, 1]^2$ unit square. The whole data space is partitioned into $N\times N$ regular gird cells, the extent of which on every dimension is δ ($=1/N$), the cell at row i and column j is denoted as C_{ij}, and has a unique identifier $C_{ij}.ID$. Like most previous methods, we use such a grid structure as the spatial index of objects for its easiness of maintaining in dynamic environment. Fig.1 shows the data structures of our framework. The dashed arrows show the relation between data structures. The gray structure is preserved in disk, and other structures are in main memory.

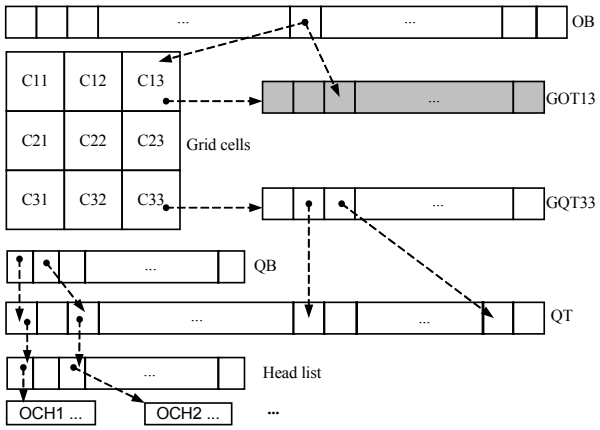

Fig.1 Data structures

Grid-based object table (GOT). It is a disk-based object table, which stores objects' information in the last processing cycle and is divided into $N\times N$ partitions by the grid cells, with each object p locating in grid cell C_{ij} being stored in the GOT partition P_{ij} with the entry form ($p.ID$, x, y), where x and y are p's coordinate values. We also use a B+ tree structure to index all objects by their $p.ID$s, so that we can quickly find a certain object.

Definition 1 (influence region) A query q's influence region IR is a circle centered at q with certain radius value, with objects outside the circle having no influence on the K-NN query of q.

Grid-based query table (GQT). It is a main memory table being divided into $N\times N$ partitions by the grid cells, with partition GQT_{ij} preserving the IDs of queries having their influence regions intersecting cell C_{ij} in the current processing cycle.

Query buffer (QB). It is a main memory structure, which records all queries moving in current processing cycle. Each query q in QB has an entry of the form ($q.ID$, x, y), where x and y are q's new coordinate values.

Object cache (OCH). Each query point q has an object cache OCH in main memory, where we try to preserve the K_τ ($K_\tau \geq K$) closest objects to q. Each object p in OCH has an entry of the form ($p.ID$, $dist(p, q)$), where $dist(p, q)$ is the distance between p and q. Objects in OCH are sorted by $dist(p, q)$ in ascending order. At the head of q's OCH, we add an entry ($q.ID$, NULL), and preserve in memory a list of such heads to get q's OCH quickly. It is different from previous methods in that we preserve more than K objects for a query, and the number can change while processing.

Query table (QT). It is a main memory structure, which preserves all queries' information. Each query has an entry of the form ($q.ID$, x, y, K, L_{max}, L_{cur}, DM, *flag*), where x and y are q's coordinate values, K is the number of objects the query needs, L_{max} ($>K$) the initial number of objects in OCH, L_{cur} the current number of objects in OCH, DM the distance of the last objects in OCH, *flag* an attribute to be '0' or '1'. When *flag*='0', it means the K-NN result of q is contained in OCH now; when *flag*='1', q needs to be revaluated for reconstructing its OCH.

Operation buffer (OB). It is a main memory table, which preserves object operations in current processing cycle. An *insert* operation has an entry as ($CellID$, 'i', $p.ID$, x, y, t), which means object p is inserted into the cell identified by $CellID$ at location (x, y) at time t. A *delete* operation has an entry as ($CellID$, 'd', $p.ID$, x, y, t), which means object p at location (x, y) in the cell identified by $CellID$ is deleted at time t. An *update* operation has two entries: ($CellID_1$, 'd', $p.ID$, x_1, y_1, t) and ($CellID_2$, 'i', $p.ID$, x_2, y_2, $t+\varepsilon$), where ε is a minimal positive value. So, each entry in OB has the form ($CellID$, *flag*, $p.ID$, x, y, t), while *flag*='i' or 'd'.

During the query processing, objects in GOT will be updated according to entries in OB, then, all entries in OB will be removed. Algorithm 1 implements rules (1)~(4) for combining operations with entries in OB.

Algorithm 1 *Combine_OP*(*OB*)
Input: all entries in OB;
Output: entries after combination.

```
1  sort all entries by p.ID;
2  for each p.ID in sorted OB do {
3     op_list←list of entries with the same p.ID sorted in
4        ascending order of t;
5     loop {
6       sequentially scan op_list;
7       if current entry's flag='i' and the next entry's flag='d'
8        then remove these two entries;
9     } until the next to last entry in op_list;
10 }
```

In a processing cycle, not all query points change their locations. If a query point q remains static, we try to solve it with the object cache strategy; otherwise, if the object cache strategy fails or q moves, we evaluate q with query cache strategy. We will discuss these two strategies in the next section.

MONITORING STRATEGIES

Object cache strategy

For queries that remain static during current processing cycle, recomputing them will have I/O cost of accessing objects from disk, even if we optimize the searching range with early answers like SEA-CNN method. So we use the object cache strategy to solve them, which has no I/O cost. The more queries remain static in a processing cycle (or the longer time each query remains static), the more effective our strategy is.

The object cache strategy tries to preserve each static query q's K_τ (K_τ is initially set to be larger than K) closest objects in q's OCH, while K_τ can dynamically change. During the processing, we check the entries in OB to maintain OCH. As long as $K_\tau \geq K$, the K-NN result of q can be found in OCH. If K_τ becomes smaller than K, the object cache strategy fails.

Definition 2 (perfect query) If a static query q can be solved by object cache strategy, q is a perfect query; else, q is an imperfect query, and newly inserted or moving queries are also imperfect queries.

When a query q is first evaluated (in the case q is just inserted) or revaluated (in the case q becomes an imperfect query), we set q's *flag* attribute in QT to be '0', and find the K_τ ($=L_{\max}$) closest objects of q to construct q's OCH. Algorithms like CPM (Mouratidis *et al*., 2005a) can be applied for such a snapshot query by increasing the searching range step by step to restrict the I/O cost. We then consider q as a perfect query, and set q's *DM* attribute in QT to be the distance of the last object in OCH. We will reset $q.DM$ when q moves or $q.flag$ becomes '1' again. In the following consecutive processing cycles that q remains static, we decrease K_τ by 1 if one of the objects in OCH is deleted or moved to a farther distance than $q.DM$, and remove this object from OCH; we increase K_τ by 1 if an object is inserted or moved to a location that is no further than $q.DM$, and add this object into OCH. When K_τ becomes smaller than K, we set $q.flag$ in QT to '1', which leads us to reconstruct OCH and restart its maintenance. According to this process, we have $K_\tau=L_{\text{cur}}$ and the following properties of a perfect query.

Property 1 The OCH of a perfect query q preserves objects inside the circle centered at q with radius $q.DM$.

Property 2 A perfect query q's influence region IR can be the circle centered at q with radius $q.DM$.

Since we have the assumption that the distribution of objects is static for a long time, we can optimistically consider that K_τ will be always near to its initial value $L_{\max}$ during the time it remains static. If we set $L_{\max}$ to be large enough, K_τ would have low possibility to become smaller than K. We do not initialize K_τ to be too large, because it will bring extra I/O cost when constructing q's OCH. If the distribution of objects changes significantly, for a static query with many objects moving toward it, we just replace its OCH with new closest objects, which need no I/O cost; for queries with more than ($K_\tau-K$) objects in OCH moving farther than *DM*, we need to reconstruct their OCHs. So, when the distribution of objects changes significantly and many query points 'loose' their closest neighbors, the object cache strategy may loose its effectivity. Algorithm 2 implements the object cache strategy. We do not update *flag* every time we check an entry in OB, but update it in the last step at Line 12, which makes our strategy more applicable with the static distributing assumption.

Algorithm 2 *Object_Cache_proc*
Input: combined OB, GQT, QT, all queries' OCHs;
Output: QT, GQT, OCHs, all these structures are modified by object cache strategy.

```
1  for each CellID in sorted OB do {
2   entry_list←list of entries having this CellID;
3   for each query q in CellID's GQT partition do {
4     q.out_list←list of entries in entry_list with flag='d' and
5       (x, y) inside q.IR;
6     q.in_list←list of entries in entry_list with flag='i' and
7       (x, y) inside q.IR;
8     update q's OCH and L_cur according to q.out_list and
9       q.in_list;
10  }
11 }
12 for each query q in QT do {if L_cur<K then set q.flag='1'; }
```

Query cache strategy

For imperfect queries, they need to be fully computed to build their OCHs, an intuitive method is to compute them as new K-NN queries, which uses early algorithms for snapshot K-NN queries in spatial database, while some of previous methods optimize the computation by restricting the searching region.

Definition 3 (searching region) A query q's searching region SR is a circle centered at q with certain radius (denoted as $SR.r$), there are at least $q.L_{max}$ objects inside it.

Using the SEA-CNN method, we can get the upper bound of $SR.r$ from the query's early result. For example, if a query point q has moved from location loc_0 to loc_1, and among the $q.L_{max}$ closest objects to loc_0, p is the farthest to loc_0 now, then, $SR.r=Dist(p, loc_0)+Dist(loc_0, loc_1)$, where $Dist$ is a function returning the distance of two location points. According to the triangular inequality, the new searching region is sure to contain at least K objects. But for new inserted queries, such method is inapplicable.

In our query cache strategy, we can further decrease $SR.r$ by QT and OCHs (we consider the combination of QT and OCH structure as the cache of query results), and this strategy can be applied even to newly inserted queries. In the above example, as shown in Fig.2, we can seek through QT for queries having their $L_{cur}\geq q.L_{max}$. Among these queries, if there exists some query point q_η at location loc_2, which has $r_\eta=q_\eta.DM+Dist(loc_1, loc_2)<Dist(p, loc_0)+Dist(loc_0, loc_1)$, we can reduce $SR.r$ of q to r_η, because the region still contains at least $q.L_{max}$ objects. In fact, we can select the minimum r_η as $SR.r$. This strategy can effectively reduce the searching regions of queries moving over long distance or newly inserted ones.

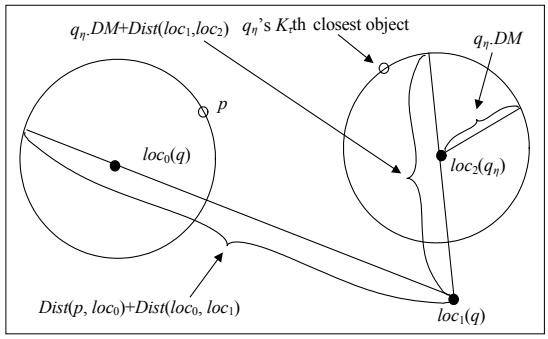

Fig.2 An example of query cache strategy

Definition 4 (ordinary query, bad query) For an imperfect query q, if there exist n ($n\geq 1$) queries (including q itself) in QT having their $L_{cur}\geq q.L_{max}$, q is an ordinary query; otherwise, it is a bad query.

Property 3 An ordinary query q's influence region IR can be the circle centered at q with radius $IR.r=\min\{r_\eta | r_\eta=q_\eta.DM+Dist(QT.q_\eta, q),\ q_\eta\in QT\wedge q_\eta.L_{cur}\geq q.L_{max}\}$.

In applications that all queries have similar K values, according to our setting of L_{max}, they will have similar L_{cur} values, so most imperfect queries will be ordinary queries. For perfect and ordinary queries, we add their IDs to GQT partitions intersecting their influence regions. For a bad query q, we use an approximate searching region SR_Θ: we simply select the closest query point q_η to q, and use the corresponding r_η as q's $SR_\Theta.r$. When all queries have similar K values, it is still possible that SR_Θ contains at least $q.L_{max}$ objects.

If the SR_Θ of a bad query q contains fewer than $q.L_{max}$ objects, it means q fails to be solved by query cache strategy, but we have got part of q's L_{max} closest objects. We can increase SR_Θ step by step, until we find the $q.L_{max}$ closest objects. After using object and query cache strategies, there will be few queries that need to be solved by such incremental searching. The algorithm for query cache strategy is presented as follows.

Algorithm 3 *Qry_cache_proc*
Input: QB, QT, GQT, GOT updated by object cache strategy, OCHs;

Output: OCHs of all queries in current processing cycle.

```
1  q_list←ID list of queries in QB and those in QT with
2    flag='1';
3  for each query q in q_list do {
4    cache_list←ID list of queries in QT with L_cur≥q.L_max,
5    if cache_list≠∅ then
6      {SR←min{SR'|SR'=q_η.DM+Dist(QT.q_η, QB.q),
7        q_η∈cache_list}; }
8    else
9      {q_η←the closest query point to q in QT−{q};
10     SR←q_η.DM+Dist(QT.q_η, q); }
11   update GQT for q according to SR;
12 }
13 for each cell C_ij in GOT do {
14   check_list←ID list of queries in the corresponding
15     partition in GQT_ij∩q_list;
16   if check_list≠∅ then
17     {check all objects in C_ij to construct OCH of queries
18       in check_list; }
19 }
20 for each query q in q_list with q.L_cur=q.L_max do
21   {update GQT for q according to its DM; q.flag='0'; }
```

The query cache strategy can be used as an optimization when initializing the system, since it is effective when there are quite a few queries distributing over the whole data space.

Main algorithm

The whole process loop is described in Algorithm 4, procedure *fill_loc* gets the deleted or moving objects' locations in last processing cycle and set the location attributes of the entries in OB whose *flag*='*d*'.

Algorithm 4 *Query_Monitor_loop*

Input: all data structures in our framework;
Output: OCHs of all queries in current processing cycle.

```
1  remove the deleted queries from QT and GQT;
2  remove their OCHs;
3  Combine_OP(OB);
4  fill_loc;  // get the deleted or moving objects' old locations,
5             // set attributes in OB.
6  Object_Cache_proc;
7  flush_OB(GOT, OB); // update GOT with entries in OB,
8                     // and then clear OB.
9  Qry_cache_proc;
10 for each query q in QT having q.L_cur<q.L_max do {
11 incrementally increase q.DM until the q.L_max closest objects
12   are found; // q.DM is initially set to the value of SR_Θ.r.
13 update GQT for q according to its new DM;
14 q.flag='0'; }
```

ANALYSIS

I/O cost

Assume the total number of objects is N_p, and the number of queries is N_q, we discuss the I/O and CPU costs in uniformly distributing object dataset. For a perfect query q, we do not need to access GOT from disk, so it has no I/O cost. In the best case, when each query is perfect, there will be no I/O cost in the cycle. For an ordinary query q, the I/O cost is equivalent to a circle range query with radius $SR.r$, which is $\alpha_1\cdot\pi\cdot DM^2\cdot N_p\approx\alpha_2\cdot L_{\max}$, where α_1 is a constant parameter describing the time of accessing an object data page, and the constant parameter α_2 describes the amortized time of accessing a single object. Let TI_q denote the average time span that a query q remains static. For a long monitoring time span T_{mon}, the total I/O cost of q solved by our cache strategies is $T_{\mathrm{cache}}=\alpha_2\cdot L_{\max}\cdot T_{\mathrm{mon}}/TI_q$. If q is recomputed in every processing cycle Γ_{p} without our object and query cache strategies, the total I/O cost for q in T_{mon} is at least $T_{\mathrm{nocach}}=\alpha_2\cdot K\cdot T_{\mathrm{mon}}/\Gamma_{\mathrm{p}}$. Let $F=T_{\mathrm{cache}}/T_{\mathrm{nocach}}=L_{\max}\Gamma_{\mathrm{p}}/(K\cdot TI_q)$, we need $F<1$ to get a lower I/O cost for our strategies, so we need $L_{\max}<K\cdot TI_q/\Gamma_{\mathrm{p}}$. In real life applications, Γ_{p} is usually shorter than TI_q to get accurate monitoring of queries, so, $K\cdot TI_q/\Gamma_{\mathrm{p}}$ is usually bigger than K, which means we can access more than K but less than $K\cdot TI_q/\Gamma_{\mathrm{p}}$ objects to construct OCH at the beginning of the TI_q/Γ_{p} cycles in which q remains static, while the SEA method accesses at least K objects for TI_q/Γ_{p} times in these cycles. In our experiments, when $K\geq 20$, and $TI_q/\Gamma_{\mathrm{p}}\geq 2$, $L_{\max}$ should be smaller than $2K$ to ensure $F<1$, and we set $L_{\max}=K+20$. Since T_{nocach} is the ideal I/O cost of recomputing q in a processing cycle, such $L_{\max}$ value can efficiently reduce the I/O cost. It is obvious that F will decrease if TI_q increases, which means the longer q remains static, the more efficient our framework for each query is.

For one cycle, let $N_{q\mathrm{o}}$ denote the number of perfect queries, which is approximately the number of static queries. The total I/O cost of this cycle comes from the range queries of $(N_q-N_{q\mathrm{o}})$ query points, and few of them would be bad queries. Let q be one of these $(N_q-N_{q\mathrm{o}})$ imperfect queries, the possibility of a grid cell intersecting q's searching region SR is no more than that of the cell intersecting the tangent rectangle of SR, which is $(\delta+2SR.r)^2$. Assuming all queries have similar K and TI_q, they will have similar

SR.r values in uniformly distributing object dataset, the possibility of one cell intersecting at least one of the (N_q-N_{qo}) queries is no more than $(N_q-N_{qo})\cdot(\delta+2DM)^2$. There are $1/\delta^2$ cells, so, the expected number of cells to be accessed is $(N_q-N_{qo})(\delta+2DM)^2/\delta^2$, which means the bigger N_{qo} is (or the more static queries exist) the fewer expected I/O cost we have. With SEA-CNN method, the expected number of cells to be accessed is $N_q(\delta+2DM')^2/\delta^2$, where DM' is the radius of searching region for K closest objects. Usually, DM' is the distance of the object that is among the K-NN objects in the last processing cycle and moves farthest to the query point now, so we can consider $DM\approx DM'$, and our strategies have lower expected I/O cost than SEA-CNN. We do not consider the I/O cost for updating GOT, because it is the same for all previous methods that use grid based object index structure, and it can be seen in (Xiong *et al*., 2005).

CPU cost

For a perfect query q, it only checks objects in OB, and updates its OCH with objects in *in_list* and *out_list* of Algorithm 2. In uniformly distributing dataset, the expected number of objects both in *in_list* and *out_list* will be linear to the area of q's influence region IR and the number of objects in OB; similarly, $L_{\max}$ value has the same relation with IR and the total number of objects N_p. Usually, objects in OB are less than N_p in a processing cycle (unless lots of objects are inserted), so, the expected number of objects both in *in_list* and *out_list* will be smaller than $L_{\max}$. As a result, the expected CPU cost for a perfect query is less than $2L_{\max}(T_{\text{comp}}+T_{\text{list}})$, where T_{comp} is the time for comparing distance of an object in *in_list* or *out_list* with DM, and T_{list} is the time of updating the sorted objects in OCH with this object. Both T_{comp} and T_{list} can be seen as constant. For an ordinary query q, it checks objects located in its searching region, and the expected CPU cost in uniformly distributing dataset is $\pi\cdot(SR.r)^2\cdot N_p\cdot(T_{\text{comp}}+T_{\text{list}})$. So, the total expected CPU cost $T_{\text{CPU\_e}}$ in a processing cycle is less than $[N_{qo}\cdot 2L_{\max}+(N_q-N_{qo})\cdot\pi\cdot(SR.r)^2\cdot N_p]\cdot(T_{\text{comp}}+T_{\text{list}})$. Usually, $L_{\max}=K\cdot TI_q/T_{\text{p}}\ll N_p$, $\pi\cdot(SR.r)^2\ll 1$, so $T_{\text{CPU\_e}}\ll[N_{qo}N_p+(N_q-N_{qo})N_p](T_{\text{comp}}+T_{\text{list}})=N_qN_p(T_{\text{comp}}+T_{\text{list}})$.

Main memory consuming

As described in Section 3, we need to preserve GQT, OB, QB, QT, OCHs and the head list of OCHs in main memory. Let M_{I} denote the size of an integer, $Sz(A)$ denote the size of memory that the content of structure A needs, λ_{m} denote the average percentage of moving queries in the query points set in a processing cycle, $K_{\tau\text{avg}}$ denote the average K_τ value of queries, $N_{q\text{c}}$ denote the average cell number a query's searching region intersects, N_{OB} denote the average number of objects in OB in a processing cycle. Then, $Sz(GQT)=N_{q\text{c}}\cdot N_q\cdot M_{\text{I}}$, $Sz(OB)\le 12\cdot N_{\text{OB}}\cdot M_{\text{I}}$, $Sz(QB)=3\lambda_{\text{m}}\cdot N_q\cdot M_{\text{I}}$, $Sz(QT)=8\cdot N_q\cdot M_{\text{I}}$. $Sz(OCH)=2K_{\tau\text{avg}}\cdot N_q\cdot M_{\text{I}}$, $Sz(\textit{head list})=2N_q\cdot M_{\text{I}}$. We need totally about $(N_{q\text{c}}+3\lambda_{\text{m}}+2K_{\tau\text{avg}}+10)\cdot N_q\cdot M_{\text{I}}+12\cdot N_{\text{OB}}\cdot M_{\text{I}}$ size of memory, when $N_q\ll N_p$ and $N_{\text{OB}}\ll N_p$, it is much smaller than the size of object dataset (at least $3N_p\cdot M_{\text{I}}$). For SEA-CNN, it preserves its query table, answer region grid, object buffer and query buffer in main memory, the size of which can be analogous to QT, GQT, OB/2 and QB respectively. So, our method just needs some extra memory for OB/2, OCHs and OCH head list, totally about $(2K_{\tau\text{avg}}+2)\cdot N_q\cdot M_{\text{I}}+6N_{\text{OB}}\cdot M_{\text{I}}$ more than SEA-CNN has.

EXPERIMENTS

We conduced a series of experiments to evaluate the effectiveness of our method (marked as CACHE), and compare it with SEA-CNN. We compute the average I/O cost of a processing cycle over a long running time. We do not test the CPU cost because it is much lower than I/O cost. The SEA-CNN method does not deal with *insert* or *delete* operations on objects and queries, so only *location-update* operation is permitted in the experiments. All experiments were performed on a 2 GHz Pentium PC machine with 1 GB main memory, running Microsoft Windows 2000. The programs are coded by Microsoft Visual C++ 6.0.

For the object dataset, we first generate N_p uniformly distributed objects, and then for each processing cycle, we randomly update N_{OB} objects with moving distance randomly selected from [0, 0.03] (We assume no object can move for more than 3 percent of the width or height of the data space in a processing cycle, for example, no human or cars can move for more than 3 km in one minute while the data space is 100 km×100 km). The N_q query points are generated similarly, except that we partition the query

points into N_G (N_G=3, 4, …) groups. Each point is randomly put into a group, so each group has about N_q/N_G points, and all queries have the same K value. In every processing cycle, one group of query points are alternately selected to be updated, so each query point's static interval $TI_q \approx N_G \cdot \Gamma_p$. The moving distances of query points are randomly selected from [0, 0.03] for the first two experiments, and from more detailed ranges for the third experiment. The size of an object data page is 1 kB, so each page can contain 85 objects. The evaluating interval Γ_p=60 s, N_p=1 000 000, δ=1/100, N_{OB}=9 000, which means there are 150 objects moving per second.

Fig.3 shows the average I/O cost with different N_q values, while K=20, $L_{max}=K+20=40$, N_G=3, $TI_q=N_G \cdot \Gamma_p$=180 s. The I/O cost of our method is about 30%~40% of SEA-CNN. The superiority decreases when N_q becomes larger than 10 000, because SEA-CNN accesses almost all data pages when the N_q is large, which is its worst case.

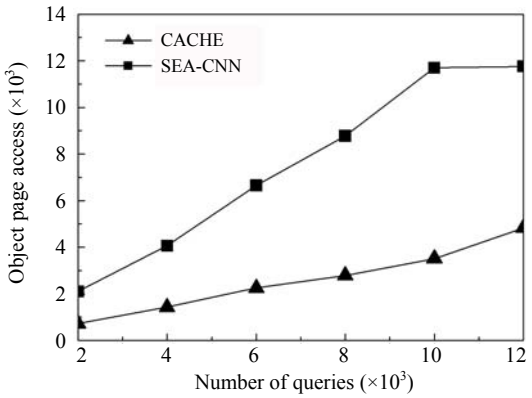

Fig.3 I/O cost with different numbers of queries

Fig.4 shows the average I/O cost with different N_G, while K=20, L_{max}=40, N_q=5 000. The cost of CACHE decreases with the increasing of N_G, which shows that the more static queries exist, the more efficient our object cache strategy is. The cost of SEA-CNN remains almost unchanged, because it recomputes static queries in every processing cycle.

Fig.5 shows the effect of the queries' moving speed on the I/O performance, K=20, $L_{max}=K+20=40$, N_q=5 000, N_G=3, $TI_q=N_G \cdot \Gamma_p$=180 s. Fig.5 shows the average I/O costs with the queries' moving distance randomly selected from [0, 0.01], [0.01, 0.02] and [0.02, 0.03]. The I/O cost of SEA-CNN increases with the increasing of queries' moving distance, while CACHE remains almost unchanged. It shows our query cache strategy is more effective for moving queries, but its effect is less than that of the object cache strategy.

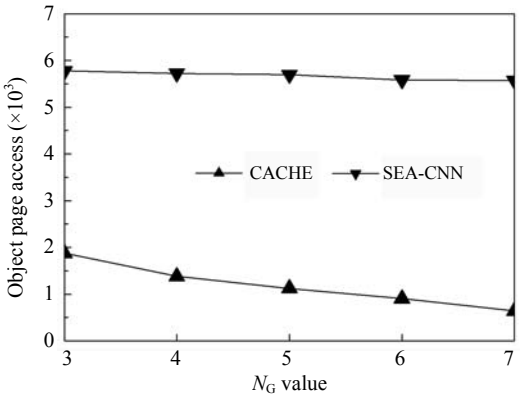

Fig.4 I/O cost with different N_G

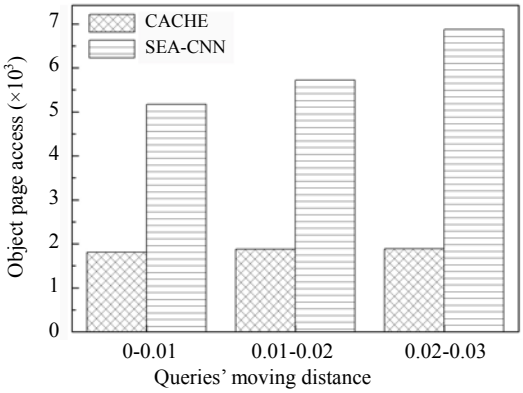

Fig.5 I/O cost with different queries' moving distance

We also test the effectivity of our strategies when the distribution of objects keeps changing. We partition the whole data space into 10×10 regular rectangular areas, with the extent of each area on every dimension being 1/10. After generating N_p uniformly distributed objects, each object belongs to the area it originally located in. Then, in the following cycles, each of the randomly selected N_{OB} objects moves toward the area center it belongs to with a distance randomly selected from certain range, objects will become clustered around these area centers later. This simulates the situation that most people move around several city centers during the day. The objects' moving distance was randomly selected from [0, 0.01], [0.01, 0.02], …, [0.05, 0.06], and [0.06, 0.07] respectively. Fig.6 shows the average I/O cost, K=20, $L_{max}=K+20=40$, N_q=5 000, N_G=3, $TI_q=N_G \cdot \Gamma_p$=180 s, and the moving distances of query points are ran-

domly selected from [0, 0.03]. When the variation of object density is slow (the moving distance is under 0.03), the I/O cost of SEA-CNN is higher; as the variation increases, both I/O cost increases; when the variation becomes quick (between 0.03 and 0.05), CACHE increases more rapidly; when the objects' moving range becomes bigger than 0.05, CACHE even has more I/O cost than SEA-CNN. The increasing of SEA-CNN is due to the increasing of the queries' searching ranges in margin spaces, while objects in center spaces are still accessed by the shared-accessing paradigm. For CACHE, with more and more objects moving toward area centers, some static queries in these areas' margin spaces may fail to be solved by object cache strategy, so the I/O cost for rebuilding OCHs increases. When the variation of object density is slow, such rebuilding work is not too much, so the object cache strategy works well; as the variation increases, the number of object cache failings increases; when the variation becomes quick, the number of such failings increases rapidly, and reconstructing OCHs needs more I/O cost than just finding the K closest objects; when the variation becomes significant, most static queries in margin spaces become imperfect, which decreases the efficiency of object cache strategy. It shows that our object cache strategy is effective under the situation that the distribution of objects does not change significantly.

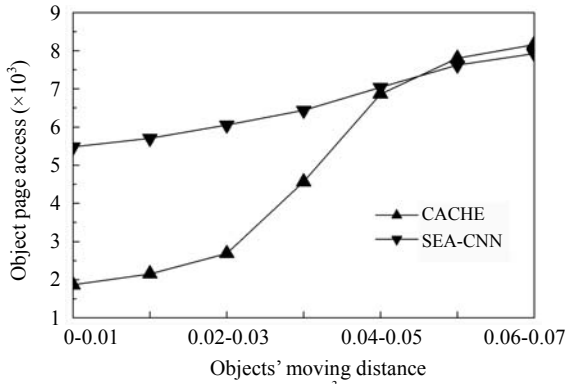

Fig.6 I/O cost with varying distribution

CONCLUSION

In this paper, we present strategies for monitoring multiple K-NN queries with both dynamic object and query datasets, which can reduce the I/O cost of accessing object dataset by previous query results. Besides *update* operation, we also deal with *insert* and *delete* operations. During each processing cycle, the static queries can be quickly maintained by object cache strategy without accessing the object dataset from disk. The strategy monitors the changes of the static queries' influence regions caused by object operations, and this can be achieved within the operation buffer and the K-NN object cache. For moving queries, the query cache strategy can effectively restrict their searching regions, by implementing a further sharing of multiple queries' execution than previous methods. We analyze the expected I/O cost of our strategies for single and multiple queries, and the experiment results showed that the object and query cache strategies are efficient in reducing the I/O cost under the situation that the distribution of objects does not change significantly.

References

Gedik, B., Wu, K.L., Yu, P.S., Liu, L., 2006. Processing moving queries over moving objects using motion-adaptive indexes. *IEEE Trans. on TKDE*, **18**(5):651-668. [doi:10.1109/TKDE.2006.81]

Iwerks, G.S., Samet, H., Smith, K.P., 2006. Maintenance of K-nn and spatial join queries on continuously moving points. *ACM Trans. on Database Systems*, **31**(2):485-536. [doi:10.1145/1138394.1138396]

Mokbel, M.F., Xiong, X., Aref, W.G., 2004. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases. Proc. SIGMOD, p.623-634. [doi:10.1145/1007568.1007638]

Mouratidis, K., Hadjieleftheriou, M., Papadias, D., 2005a. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. Proc. SIGMOD, p.634-645. [doi:10.1145/1066157.1066230]

Mouratidis, K., Papadias, D., Bakiras, S., Tao, Y., 2005b. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *IEEE Trans. on TKDE*, **17**(11):1451-1464. [doi:10.1109/TKDE.2005.172]

Prabhakar, S., Xia, Y., Kalashnikov, D.V., Aref, W.G., Hambrusch, S.E., 2002. Query indexing and velocity constrained indexing: scalable techniques for continuous queries on moving objects. *IEEE Trans. on Computers*, **51**(10):1124-1140. [doi:10.1109/TC.2002.1039840]

Wu, K.L., Chen, S.K., Yu, P.S., 2006. Incremental processing of continual range queries over moving objects. *IEEE Trans. on TKDE*, **18**(11):1560-1575. [doi:10.1109/TKDE.2006.176]

Xiong, X., Mokbel, M.F., Aref, W.G., 2005. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-Temporal Databases. International Conference on Data Engineering, p.643-654. [doi:10.1109/ICDE.2005.128]

Yu, X., Pu, K.Q., Koudas, N., 2005. Monitoring K-Nearest Neighbor Queries Over Moving Objects. International Conference on Data Engineering, p.631-642. [doi:10.1109/ICDE.2005.92]