ORIENTAL JOURNAL OF
COMPUTER SCIENCE & TECHNOLOGY
An International Open Free Access, Peer Reviewed Research Journal
www.computerscijournal.org

ISSN: 0974-6471
December 2011,
Vol. 4, No. (2):
Pgs. 371- 377

Enhanced Cache Gird Partitioning Technique for K-NN Queries

SHATADAL PATRO[1] and ASHA AMBHAIKAR[2]*

[1]RCET, Bhilai, Sri Ram Colony, Rajnandgaon (India).
[1]Department of Computer Science & Engineering, RCET, Bhilai (India).
*Corresponding author: E-mail: asha31.a@rediffmail.com

(Received: November 10, 2011; Accepted: November 16, 2011)

ABSTRACT

Mobile database applications through wireless equipments e.g., PDAs, laptops, cell phones and etc. are growing rapidly. In such environment, clients, servers and object may change their locations. A very applicable class of query is continuous k-NN query which continuously returns the k nearest objects to the current location of the requester. Respect to limitations in mobile environments, it is strongly recommended to minimize number of connections and volume of data transmission from the servers. Caching seems to be very profitable in such situations. In this paper, a enhanced cache grid partitioning technique for continuous k-NN queries in mobile DBSs is proposed. In this, by square grid partitioning the complete search space is divided into such grid areas so that we can impose a piecemeal ordering on the query targets. Simulation results show that the proposed cache grid partitioning schema provides a considerable improvement in response time, number of connections and volume of transferred data from DB server.

Key words: Cache grid partitioning, k-nearest neighbour, Grid range, Cache hit rate.

INTRODUCTION

Wireless technology is growing rapidly and it's beneficial applications which cause use of PDAs, laptops, cell phones and etc. To access data anywhere and anytime, are very common now days. Mobile database is such a technology which confronts with some new problems, limitations and challenges (e.g., bandwidth limitations, missing connectivity, unreliable and asymmetric links). In wireless data broadcast systems, a client has to stay active to continuously receive and check the broadcast data until the data the data objects of interest arrive. Since the mobile user may repeatedly submit the same type of query, each time for different values of one or more attributes, and also from the same position or from different positions, a cache maintained by the user can cut down the total processing and network activity time. Consider a wireless data broadcast system that periodically broadcasts a collection of data objects to mobile clients. Each data object consists of a set of attribute values. Among them, location is particularly important. Access efficiency and energy

conservation are two critical issues for mobile users, concerning how fast a request could be satisfied and how energy efficient a technique is. To facilitate energy conservation, a smart mobile device is expected to support two operation modes: active mode and doze mode. The device normally operates in active mode; and switches to doze mode to save energy when the system becomes idle.

In the literature, two performance metrics, namely access latency and tuning time, are used to measure access efficiency and energy conservation for mobile clients in a wireless data broadcast system:

Access Latency

The time elapsed from the moment a query is issued to the moment it is satisfied.

Tuning Time

The time a mobile client stays active to receive the requested data items.

In wireless data broadcast systems, a client has to stay active to continuously receive and check the broadcast data until the data objects of interest arrive. This process consumes a lot of energy.

K-Nearest Neighbor Search

A k-nearest neighbour (kNN) query finds the nearest k objects to a query point. The basic idea behind kNN algorithms is to determine a search space based on the partial knowledge of object distribution obtained from index table. The search space will continuously shrink as more knowledge of the data distribution is obtained. The challenge is to quickly determine a precise search space that contains all the k objects. The initial search space is the whole spatial region covering all the data objects in the system. As a client tunes into the broadcast channel to receive the first index table, a circle centred at the query point (which specifies a search space) is drawn to include at least k data objects. If the query point is located far away from the current broadcast frame, the circle could be very large because the index table has very limited information about distribution of data objects far away (due to exponential increase of data objects covered by index table entries). As the client continues to monitor the broadcast channel and to obtain more information about object distribution from index tables of subsequently broadcast frames, the circle can be shrunk to avoid retrieval of frames containing unwanted objects. On the other hand, if a query point is close to the current broadcast frame, the search space will converge very rapidly and the search process typically will terminate quickly, because there are more index entries in DSI covering data objects to be broadcast soon. The search space is finalized when no more objects could further reduce the radius. The search is completed when all the objects within the search space are retrieved.

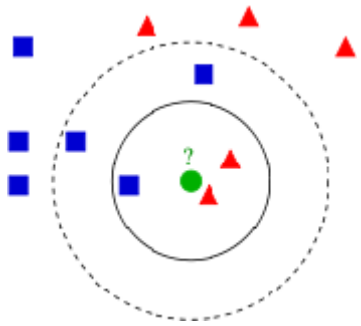


Fig. 1: Representation of Basic k-NN search

Mobile Cache with square box grid partitioning

Since the mobile user may repeatedly submit the same type of query, each time for different values of one or more attributes, and also from the same position or from different positions, a cache maintained by the user can cut down the total processing and network activity time.

Our approach towards designing such a cache is based on square box grids with axis-aligned boundaries. The following figure depicts the search space divided into three such grids with the mobile user at the origin.

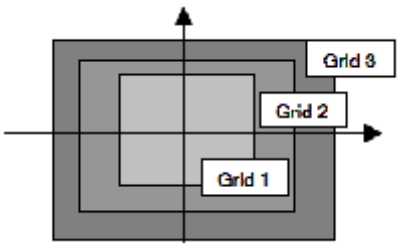


Fig. 2: Square box grid partitions

In Figure 2, the origin denotes the query point.

Grids are numbered starting from 1. Each grid is associated with a grid range, which is the maximum absolute x or y coordinate value for that grid relative to the query point. A point P with coordinates (xp,yp) is assigned gridi, when gridrangei-1 < max(abs(xp), abs(yp)) <= gridrangei, where i > 1. For i = 1, 0 <= max (abs (xp), abs (yp)) <= gridrangei.

This means all points on the outer boundary of a grid belong to the grid, but the inner boundary does not belong to the grid.

A faster search can be expected if the values of the grid range increase geometrically. In practice, the ranges for the grids can be assigned incrementally as follows:-

A suitable initial range s is chosen for the 1st grid. Then the range of the subsequent grid is assigned a value which is the next higher integer of s.√2 and so on.

An example of 8 grids with minimum grid-range 10 and maximum 100 is shown in Table 1.

Table 1: A Sample Grid Partition

Grid Range (max & min values in positive x or y direction)	Grid Number
0-10	1
11-15	2
16-22	3
23-32	4
32-46	5
47-66	6
67-94	7
95-100	8

This means by partitioning the complete search space into such grid areas, we can impose a piecemeal ordering on the query targets and thereby reduce the total search space.

For finding the k nearest points, we start from grid 1 and go on traversing the grids in ascending order, till we find a non-empty grid. We calculate the distances of the points in the grid from the origin and try to find up to k nearest points. All points whose distances are less than the grid range must be the nearest points. If we find k such points our search ends here. If we have found less than k points (say k1 points) and if among the nearest points we have found in the current grid, there are points whose distances are larger than the grid range then we need to compare their distances only with points in the next non-empty grid. Say there are k2 such points. So we scan up to the next non-empty grid and find the points with the shortest distances in it. Since we have already found k1 nearest points, then we need to find only up to k – k1 nearest points and compare them with the k2 points in the previous grid. Say we have found k3 (k3 <= k –k1) nearest points. So we sort the (k3 + k2) points on the distance and of these the k2 nearest points are the nearest from the origin. So we have thus found (k1 + k2) nearest points. Of the rest k3 points, if there are k4 points whose distances are less than the current grid's range, then these are also nearest points. Thus we have found (k1 + k2 + k4) nearest points and we need to compare the remaining (k3 –k4) points with the next non-empty grid. We proceed in this way until we have found all k nearest points or all the grids are processed.

Architecture

Base Tower (BT) of each cell has data about landmarks located in its own cell as well as in its neighbouring cells. Each landmark has associated attributes like Name, Latitude, Longitude, and other attributes specifying its other characteristics, e.g. "Star" information for Hotels.

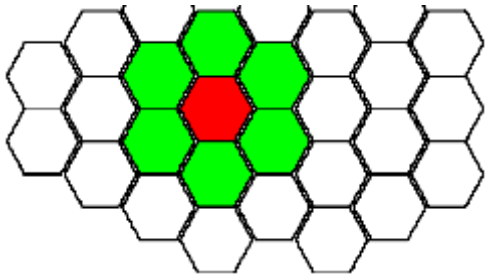


Fig. 3:

The BT broadcasts these data on a broadcast channel

For example, if a mobile client is in the red coloured cell (as shown in the Figure 3), then the BT of this cell will broadcast data of landmarks located in this cell as well as in neighbouring six cells (shown in green colour in the above figure). When a mobile client enters the cell under a BT, it tunes into the broadcast channel, downloads the data and stores them into cache memory.

As long as mobile client is in the area covered by a BT, all queries will be processed with the data stored in the cache. When it crosses over to an area under the jurisdiction of another BT, the cache will be reloaded with new data downloaded from the broadcast channel of that BT.

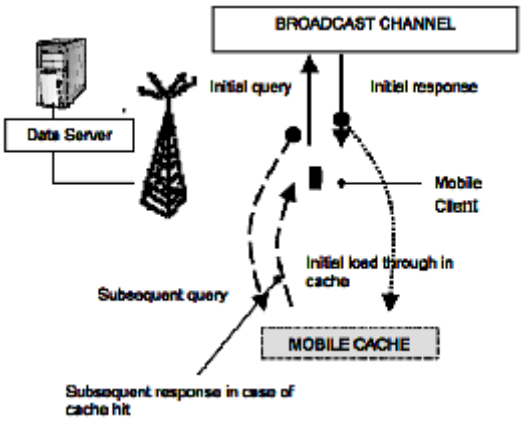


Fig. 4:

Cache Maintenance

The three important issues for cache maintenance considered are:

- Fast cache insertion and deletion
- A fast cache search algorithm
- A reasonable cache hit rate

Fast cache insertion and deletion

We assume that the maximum extent in either x and y direction of the complete search space is known a-priori and we denote it by MAX_EXTENT (which we assume to be the same for both x and y direction). Also we assume that the coordinate system has been set up in such a way that every position can be specified by integer coordinates. In case this is not true, then the coordinate values can be multiplied by a suitable scaling factor (multiple of 10) and only the integer part of the product need to be considered. The cache is organized in the form of a number of separate buffers corresponding to each grid. The buffers can be implemented as lists or as dynamic tables.

The insertion operation is similar to a disjoint set find operation. An integer array rangebuf [MAX_EXTENT] is set up before the processing starts and each element in the array is initialized with the corresponding grid partition number. For example, for the grid partition as shown in Table 1, MAX_EXTENT=100, rangebuf[0] to rangebuf^{10} contains 1, rangebuf^{11} to rangebuf^{15} contains 2 and so on. It is to be noted that the actual search space covered in this case is 200X200, i.e., the total number of possible search points is 40000.

During cache insertion, for each target point its grid number is looked up from the array with the subscript max (abs(x), abs(y)), where x and y are the coordinates of the target point. Once the grid number corresponding to a target point is known, the point can be inserted into the buffer corresponding to that grid. Thus, a cache insertion will involve a lookup into an n-element array where the maximum possible number of targets is (2n) 2.

The cache will be maintained in the RAM of the mobile device with a limited capacity (typically 8-12 MB). Also the user should be allowed to run other applications during the same time. So it is possible that while inserting into the cache, an out of memory condition may occur. In this case, the buffer belonging to the farthest grid with all the points already inserted into the buffer will be deleted and the corresponding memory will be freed. Additionally, no points belonging to that grid or any grid farther than that will be read into the cache from that point onwards. This will ensure that a fast deletion and memory clean up happen every time the cache is filled up.

General Cache Algorithm

Initially the cache is empty, so for the initial query the network is searched and simultaneously the cache is loaded from network data. The cache is organized in the form of grid partitions and the user maintains the count of points in each grid partition. For subsequent queries, first it is tested if

the cache needs to be reloaded and/or repartitioned based on the updated Query Point location and the partition point counts. If neither is required the existing cache is searched to answer queries.

The main cache algorithm is given below:-

```
If EmptyCache() Then    // Initial Query
    SearchNetworkAndLoadCacheFromNetwork()
    UpdateQPLocationAndPartitionCounts()
ElseIf QueryPointMoved(QPLocation) Then  //
Repeat query at a new position
    TestIfCacheRepartitionIsRequired()
    If CacheRepartitionNotRequired Then
    SearchInCache(query)
    Else TestIfCacheReloadIsRequired()
    If CacheReloadingNotRequired Then
   RepartitionCache()
   SearchInCache(query)
 Else    // Moved out of range
   SearchNetworkAndLoadCacheFromNetwork()
   UpdateQPLocationAndPartitionCounts ()
Else   // Repeat query at same position
 SearchInCache(query)
```

Analysis of the algorithm

Assuming a uniform distribution of objects over the search space, the number of objects in each partition is proportional to its area. Therefore the number of objects in the i-th partition is found from the relation:

$$n_i = n_1 + n_2 + \ldots + n_{i-1}, \quad \text{for } i > 1$$

$$\text{It follows, } n_i = 2 * n_{i-1}, \quad \text{for } i > 2$$

In order to find k objects, in the best case exactly k objects are found within some i-th partition starting from the 1st partition. In this case only the objects in the (i+1)-th partition are to be considered, whose number can also be assumed as k. Thus a total 2k objects are to be compared.

In the worst case, $k-\delta$ ($\delta < k/2$) objects are found within the i-th partition, thus the next two partitions need to be considered. So, the maximum number of objects to be compared is $k + k + 2k = 4k$.

As per the algorithm, a sorted array of length k is maintained. Therefore the total number of comparisons to find the k nearest objects is 2k2 in the best case and 4k2 in the worst case. The number of operations to partition the n-objects is O(n). Therefore, the time complexity of the k-n-n search algorithm is O(n+k2).

In a linked list implementation, n object nodes plus r header nodes (where r is the number of partitions) have to be maintained. For a dynamic array implementation, arrays of twice the number of objects need to be kept for each partition. Thus the space complexity is O(n).

In case k2 is less than n, or to be more exact, if $k<\sqrt{n}/2$, then the O(n) term dominates. Since partitioning needs to be done only once, the total time for a number of queries from a fixed location is thus only O(n) and does not depend significantly on the number of queries issued.

Ensuring a reasonable cache hit rate

The cache-hit rate primarily depends on the available space in the cache. If all index entries can be successfully stored in the cache, then all nearest neighbors must be found in the cache. Even when it does not happen, there is a fair possibility that the nearest neighbors will be found in the cache. It follows from the fact that, since the grid ranges and so grid areas increase geometrically after the first grid, it can be expected that most of the points in the search space will be in the last grids and so, they do not enter consideration from the very outset.

There can be three possible outcomes from the cache search:

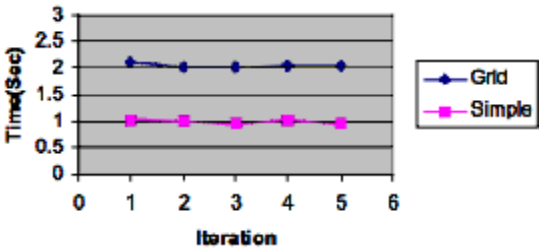
- The nearest neighbors are found in the cache
- No target satisfying the given condition is found in the cache
- Nearest neighbors are found in the last grid, but their distances are larger than the last grid range, i.e., these are at one of the corners (outside the inscribed circle) of the last grid and there may be points nearer than those.

In case of the last two outcomes, the nearest neighbors have to be sought in the network data.

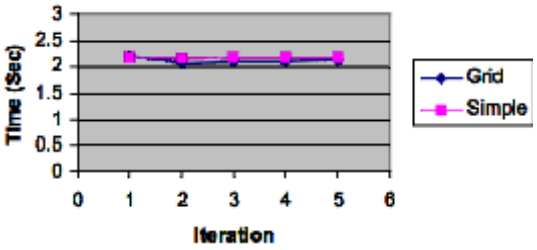
Performance Comparison

We have done simulation studies with both

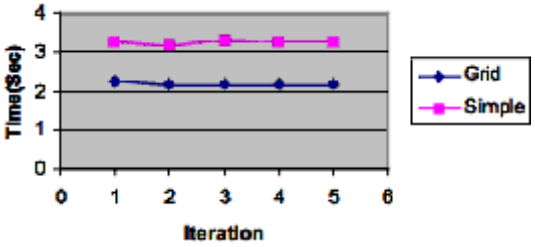
uniform random datasets and a real dataset (sequoia1.zip at http://isl.cs.unipi.gr/db/projects/rtreeportal/, which contains California locations). Here we present the results obtained with the real dataset consisting of 62556 location points. We have executed a sample application where the user wants to know the 3 nearest locations in 4 directions (East, West, North and South) in consecutive queries. In each case we have executed 100 to 500 iterations.



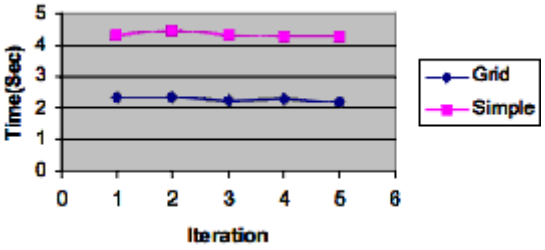
Time taken for one (E) direction



Time taken for 2 (EW) direction



Time taken for 3 (EWN) directions



Time taken for 4 (EWNS) directions

Following are the graphs showing the time taken over 5 iterations (each iteration consisting of 100 nested iterations). The 'Simple' refers to the method without using a cache.

The results show that for the first query (only points in the EAST), the cache strategy takes slightly more time than without using a cache. For 2 or more queries (where more than one direction data were queried), our strategy is 25% to 40% faster in most cases.

Conclusion & Future enhancement

In this project work, I have presented a cache strategy for mobile users, which can improve the performance of k-nearest-neighbor queries in the Wireless networks. We have also presented the results from a performance evaluation done with real data for a specific query, which is meant to find the nearest neighbor in a user-specified direction. This shows that by using the cache, the processing time and network activity time of the mobile user can be improved.

The application developed by me, "Enhanced Cache Grid Partitioning Technique for LBS" finds landmarks of interests with the criteria specified by the user, displays on the screen their names, latitudes and longitudes. My work can be further extended by feeding the latitude and longitude values of landmarks to a GPS Navigation System that will guide the user to reach the destination landmark.

REFERENCES

1. Ali A. Safaei, Mehdi Mallah, Fatemeh Abdi, Shahab Behjati,"Semantic Cache Schema for Continuous k-NN Queries in Mobile DBSs", *Information Technology International Research Journal* **1**(1): 21-32 (2011).
2. Mr. C. Gopala Krishnan, Dr. V. Kavitha, Ms. J. Jesu Vedha Nayahi,"Reducing Latency by Providing Location Based Services using Hybrid Cache in Ad Hoc Networks", *Int. J. of Advanced Networking and Applications* **2**(01): 428-436 (2010).
3. Wei Zhang, Jianzhong Li, and Haiwei Pan,"Processing Continuous k -Nearest Neighbor Queries in Location-Dependent Application", *IJCSNS International Journal of Computer Science and Network Security*, **6**(3A): (2006).
4. J. Xu, Q. Hu, W.-C. Lee and D.L. Lee "Performance Evaluation of an Optimal Cache Replacement Policy for Wireless Data Dissemination under Cache Consistency", *IEEE Transaction on Knowledge and Data Engineering* (TKDE), **16**(1): 125-139 (2004).
5. A. Y. Seydim, M. H. Dunham , V. Kumar, "Location Dependent Query Processing" Proceedings of the 2nd ACM International Workshop on Data engineering for Wireless and Mobile Access, Santa Barbara, California, United States (2001).
6. S. Berchtold, D. A. Keim, H. P. Kriegel, and T. Seidl. Indexing the solution space: A new technique for nearest neighbor search in high-dimensional space. *IEEE Transactions on Knowledge and Data Engineering (TKDE),* **12**(1): 45-57 (2000).
7. K. L. Cheung and W.-C. Fu. Enhanced nearest neighbour search on the r-tree. *SIGMOD Record,* **27**(3): 16-21 (1998).
8. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *In Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD'95), 71-79 (1995).