Indexing, Query and Velocity Constraint CE1

SUNIL PRABHAKAR, DMITRI KALASHNIKOV,
YUNI XIA
Department of Computer Sciences, Purdue University,
West Lafayette, Indiana 47907, USA
sunil@cs.purdue.edu, yxia@cs.purdue.edu,
dvk@cs.purdue.edu

Synonyms

Spatio-temporal indexing; Continuous queries

Definition

Moving object environments are characterized by large numbers of moving objects and numerous concurrent continuous queries over these objects. Efficient evaluation of these queries in response to the movement of the objects is critical for supporting acceptable response times. In such environments the traditional approach of building an index on the objects (data) suffers from the need for frequent updates and thereby results in poor performance. In fact, a brute force, no-index strategy yields better performance in many cases. Neither the traditional approach, nor the brute force strategy achieve reasonable query processing times. The efficient and scalable evaluation of multiple continuous queries on moving objects can be achieved by leveraging two complimentary techniques: *Query Indexing* and *Velocity Constrained Indexing* (*VCI*). Query Indexing relies on i) incremental evaluation; ii) reversing the role of queries and data; and iii) exploiting the relative locations of objects and queries. VCI takes advantage of the maximum possible speed of objects in order to delay the expensive operation of updating an index to reflect the movement of objects. In contrast to techniques that require exact knowledge about the movement of the objects, VCI does not rely on such information. While Query Indexing outperforms VCI, it does not efficiently handle the arrival of new queries. Velocity constrained indexing, on the other hand, is unaffected by changes in queries. A combination of Query Indexing and Velocity Constrained Indexing enables the scalable execution of insertion and deletion of queries in addition to processing ongoing queries.

Historical Background

The importance of moving object environments is reflected in the significant body of work addressing issues such as indexing, uncertainty management, broadcasting, and models for spatio-temporal data. Several indexing techniques for moving objects have been proposed. These include indexes over the histories, or trajectories, of the positions of moving objects, or the current and anticipated future positions of the moving objects. Uncertainty in the positions of the objects is dealt with by controlling the update frequency where objects report their positions and velocity vectors when their actual positions deviate from what they have previously reported by some threshold. Tayeb et al. use quad-trees to index the trajectories of one-dimensional moving points. In one approach, moving objects and their velocities are mapped to points which are indexed using a kD-tree. Another indexes the past trajectories of moving objects treated as connected line segments. Yet another considers the management of collections of moving points in the plane by describing the current and expected positions of each point in the future [3]. They address how often to update the locations of the points to balance the costs of updates against imprecision in the point positions. The two techniques presented here appeared in [2].

Scientific Fundamentals

Continuous Query Processing

Location-based environments are characterized by large numbers of moving (and stationary) objects. To support these services it is necessary to execute efficiently several types of queries, including range queries, nearest-neighbor queries, density queries, etc. An important requirement in location-aware environments is the continuous evaluation of queries. Given the large numbers of queries and moving objects in such environments, and the need for a timely response for continuous queries, efficient and scalable query execution is paramount.

This discussion focuses on range queries. Range queries arise naturally in spatial applications such as a query that needs to keep track of, for example, the number of people that have entered a building. Range queries can also be useful as pre-processing tools for reducing the amount of data that other queries, such as nearest-neighbor or density, need to process.

Model

Moving objects are represented as points, and queries are expressed as rectangular spatial regions. Therefore, given a collection of moving objects and a set of queries, the problem is to identify which objects lie within (i.e., are relevant to) which queries. Objects report their new locations periodically or when they have moved by a significant distance. Updates from different objects arrive continuously and asynchronously. The location of each object is saved in a file. Objects are required to report only their location, not velocity. There is no constraint on the movement of objects except that the maximum possible speed of each object is known and not exceeded (this is required only for Velocity Constrained Indexing).

Ideally, each query should be re-evaluated as soon as an object moves. However, this is impractical and may not even be necessary from the user's point of view. The continuous evaluation of queries takes place in a periodic fashion whereby the set of objects that are relevant to each continuous query are determined at fixed time intervals.

Limitations of Traditional Indexing

A natural choice for efficient evaluation of range queries is to build a spatial index on the objects. To determine which objects intersect each query, the queries are executed using this index. The use of the spatial index should avoid many unnecessary comparisons of queries against objects this approach should outperform the brute force approach. This is in agreement with conventional wisdom on indexing. In order to evaluate the answers correctly, it is necessary to keep the index updated with the latest positions of objects as they move. This represents a significant problem. Notice that for the purpose of evaluating continuous queries, only the current snapshot is relevant and not the historical movement of objects.

In [2], three alternatives for keeping an index updated are evaluated. Each of these gives very poor performance – even worse than the naive brute-force approach. The poor performance of the traditional approach of building an index on the data (i.e. the objects) can be traced to the following two problems: i) whenever *any* object moves, it becomes necessary to re-execute *all* queries; and ii) the cost of keeping the index updated is very high.

Query Indexing

The traditional approach of using an index on object locations to efficiently process queries for moving objects suffers from the need for constant updates to the index and re-evaluation of all queries whenever any object moves. These problems can be overcome by employing two key ideas:

- *reversing* the role of data and queries, and
- *incremental* evaluation of continuous queries.

The notion of *safe regions* that exploit the relative location of objects and queries can further improve performance.

In treating the queries as data, a spatial index such as an R-tree is built on the queries instead of the customary index that is built on the objects (i.e. data). This structure is called a Query-Index or *Q-index*. To evaluate the intersection of objects and queries, each object is treated as a "query" on the Q-index (i.e., the moving objects are treated as queries in the traditional sense). Exchanging queries for data results in a situation where a larger number of queries (one for each object) is executed on a smaller index (the Q-index), as compared to an index on the objects. This is not necessarily advantageous by itself. However, since not all objects change their location at each time step, a large number of "queries" on the Q-index can be avoided by incrementally maintaining the result of the intersection of objects and queries.

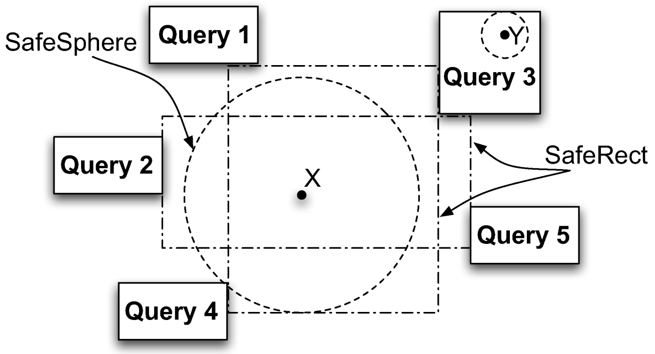
Incremental evaluation is achieved as follows: upon creation of the Q-index, all objects are processed on the Q-index to determine the initial result. Following this, the query results are incrementally adjusted by considering the movement of objects. At each evaluation time step, only those objects that have moved since the last time step are processed, and adjust their relevance to queries accordingly. If most objects do not move during each time step, this can greatly reduce the number of times the Q-index is accessed. For objects that move, the Q-index improves the search performance as compared to a comparison against all queries.

Under traditional indexing, at each time step, it would be necessary to update the index on the objects and then evaluate each query on the modified index. This is independent of the movement of objects. With the "Queries as Data" or the Q-index approach, only the objects that have moved since the previous time step are evaluated against the Q-index. Building an index on the queries avoids the high cost of keeping an object index updated; incremental evaluation exploits the smaller numbers of objects that move in a single time step to avoid repeating unnecessary comparisons. Upon the arrival of a new query, it is necessary to compare the query with all the objects in order to initiate the incremental processing. Deletion of queries is easily handled by ignoring those queries. Further improvements in performance can be achieved by taking into account the relative locations of objects and queries or safe regions.

***Safe Region*s: Exploiting Query and Object Locations**

The relevance of an object with respect to a given query can only change if the object crosses a query boundary. Therefore, a region surrounding an object that does not cross any query boundary represents a region within which the object can move without affecting the result of any query. Such a region is termed a *Safe Region*. Two types of safe regions are maximal circles (sphere in general) and maximal rectangles centered at an object's current location. These are termed *SafeSphere* and *SafeRect* respectively. While there is only one maximal sphere for a given object, there can be multiple maximal rectangles.

Figure 1 shows examples of each type of *Safe Region* for two object locations, X and Y. Note that the union of two safe regions is also a safe region. If an object knows a safe region around it, it need not send updates of its movement to the server as long as it remains within the safe region. The safe region optimizations significantly reduce the need to test data points for relevance to queries if they are far from any query boundaries and move slowly. Using safe regions can significantly improve performance.

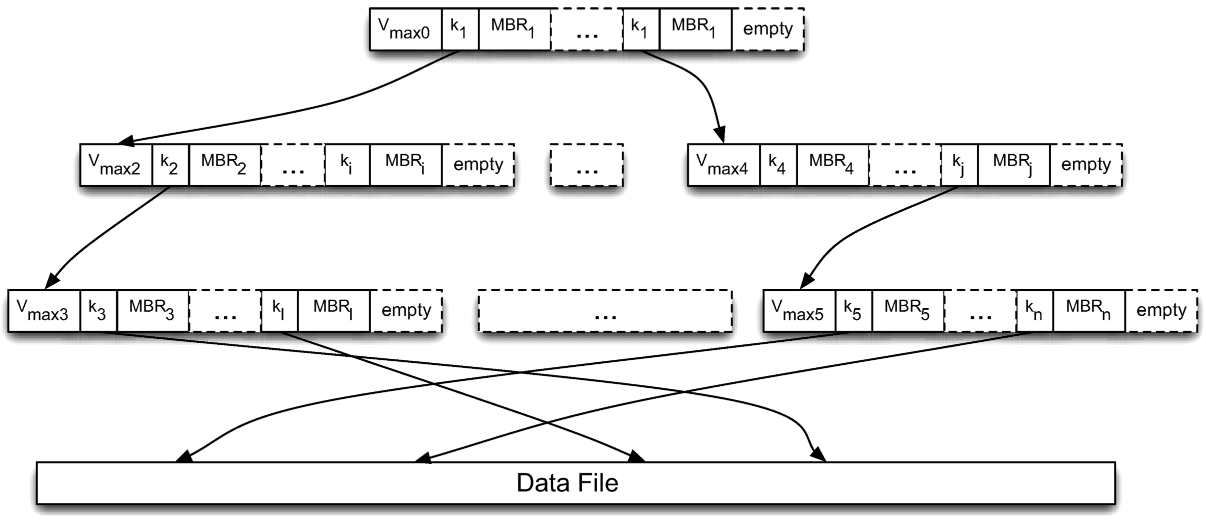


Indexing, Query and Velocity Constraint, Figure 1 Examples of *Safe Regions*

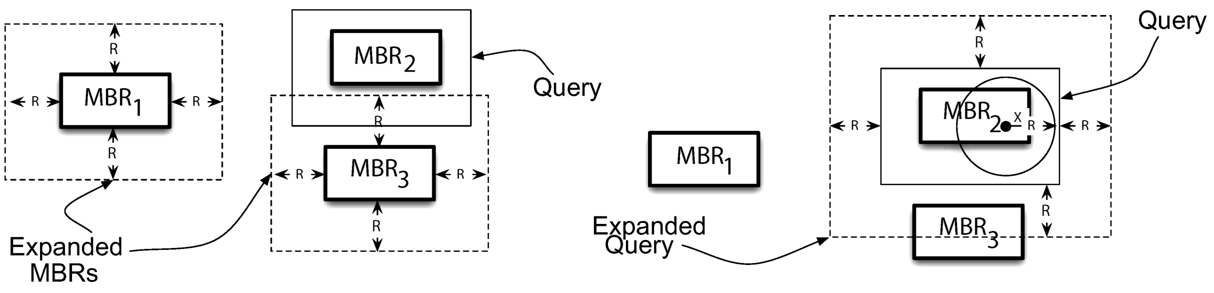
Velocity-Constrained Indexing

The technique of Velocity-Constrained Indexing (VCI) eliminates the need for continuous updates to an index on moving objects by relying on the notion of a maximum speed for each object. Under this model, the maximum possible speed of each object is known.
A VCI is a regular R-tree based index on moving objects with an additional field in each node: $v_{\max}$. This field stores the maximum allowed speed over all objects covered by that node in the index. The $v_{\max}$ entry for an internal node is simply the maximum of the $v_{\max}$ entries of its children. The $v_{\max}$ entry for a leaf node is the maximum allowed speed among the objects pointed to by the node. Figure 2 shows an example of a VCI. The $v_{\max}$ entry in each node is maintained in a manner similar to the Minimum Bounding Rectangle (MBR) of each entry in the node, except that there is only one $v_{\max}$ entry per node as compared to an MBR per entry of the node. When a node is split, the $v_{\max}$ for each of the new nodes is copied from the original node.
Consider a VCI that is constructed at time t_0. At this time it accurately reflects the locations of all objects. At a later time t, the same index does not accurately capture the correct locations of points since they may have moved arbitrarily. Normally the index needs to be updated to be correct. However, the $v_{\max}$ fields enable us to use this old index without updating it. We can safely assert that no point will have moved by a distance larger than $R = v_{\max}(t - t_0)$. If each MBR is expanded by this amount in all directions, the expanded MBRs will correctly enclose all underlying objects. Therefore, in order to process a query at time t, the VCI created at time t_0 can be used without being updated, by simply comparing the query with expanded version of the MBRs saved in VCI. At the leaf level, each point object is replaced by a square region of side $2R$ for comparison with the query rectangle.[1]
An example of the use of the VCI is shown in Fig. 3a which shows how each of the MBRs in the same index node are expanded and compared with the query. The expanded MBR captures the worst-case possibility that an object that was at the boundary of the MBR at t_0 has moved out of the MBR region by the largest possible distance. Since a single $v_{\max}$ value is stored for all entries in the node, each MBR is expanded by the same distance, $R = v_{\max}(t - t_0)$. If the expanded MBR intersects with the query, the corresponding child is searched. Thus to process a node all the MBRs stored in the node (except those that intersect without expansion) need to be expanded. Alternatively, a single expansion of the query by R could be performed and compared with the unexpanded MBRs. An MBR will intersect with the expanded query if and only if the same MBR after expansion intersects with the original query. Figure 3b shows the earlier example with query expansion. Expanding the query once per node saves some unnecessary computation.
The set of objects found to be in the range of the query based upon an old VCI is a superset, S', of the exact set of objects that are currently in the query's range. Clearly, there can be no false dismissals in this approach. In order to eliminate the false positives, it is necessary to determine the current positions of all objects in S'. This is achieved through a post-processing step. The current location of the object is retrieved from disk and compared with the query to determine the current matching. Note that it is not always necessary to determine the current location of each object that falls within the expanded query. From the position recorded in the leaf entry for an object, it can move by at most R. Thus its current location may be anywhere within a circle of radius R centered at the position recorded in the leaf. If this circle is entirely contained within the unexpanded query, there is no need to post-process this object for that query. Object X in Fig. 3b is an example of such a point.
To avoid performing an I/O operation for each object that matches each expanded query, it is important to handle the post-processing carefully. We can begin by first preprocessing all the queries on the index to identify the set of objects that need to be retrieved for any query. These objects are then retrieved only once and checked against

[1]Note that it should actually be replaced by a circle, but the rectangle is easier to handle.



Indexing, Query and Velocity Constraint, Figure 2 Example of Velocity Constrained Index (VCI)



Indexing, Query and Velocity Constraint, Figure 3 Query Processing with Velocity Constrained Index (VCI)

all queries. This eliminates the need to retrieve the same object more than once. To avoid multiple retrievals of a page, the objects to be retrieved can first be sorted on page number. Alternatively, a clustered index can be built. Clustering may reduce the total number of pages to be retrieved. Clustering the index can improve the performance significantly.

Refresh and Rebuild

The amount of expansion needed during query evaluation depends upon two factors: the maximum speed $v_{\max}$ of the node, and the time that has elapsed since the index was created, $(t-t_0)$. Thus over time the MBRs get larger, encompassing more and more dead space, and may not be minimal. Consequently, as the index gets older its quality gets poorer. Therefore, it is necessary to *rebuild* the index periodically. This essentially resets the creation time, and generates an index reflecting the changed positions of the objects. Rebuilding is an expensive operation and cannot be performed too often. A cheaper alternative to rebuilding the index is to *refresh* it. Refreshing simply updates the locations of objects to the current values and adjusts the MBRs so that they are minimal. Following refresh, the index can be treated as though it has been rebuilt.
Refreshing can be achieved efficiently by performing a depth-first traversal of the index. For each entry in a leaf node the latest location of the object is retrieved (sequential I/O if the index is clustered). The new location is recorded in the leaf page entry. When all the entries in a leaf node are updated, the MBR for the node is computed and recorded it in the parent node. For directory nodes when all MBRs of its children have been adjusted, the overall MBR for the node is computed and recorded it in the parent. This is very efficient with depth-first traversal. Although refresh is more efficient than a rebuild, it suffers from not altering the structure of the index – it retains the earlier structure. If points have moved significantly, they may better fit under other nodes in the index. Thus there is a trade-off between the speed of refresh and the quality of the index. An effective solution is to apply several refreshes followed by a less frequent rebuild. In practice, refreshing works very well thereby avoiding the need for frequent, expensive rebuilds.

Performance

Detailed evaluation of the approaches can be found in [2]. Overall, the experiments show that for Query Indexing, *SafeRect* is the most effective in reducing evaluations. Q-Index is found to give the best performance compared to VCI, traditional indexing, and sequential scans. It is also robust across various scales (numbers of objects, queries), rates of movement, and density of objects versus queries. VCI, on the other hand, is more effective than traditional approaches for small numbers of queries. Moreover, the total cost of VCI approaches that of a sequential scan after some time. Clustering can extend the utility of the VCI index for a longer period of time. Eventually, refresh or rebuilds are necessary. Refreshes are much faster and very effective and thereby reduce the need for frequent rebuilds. VCI is not affected by how often objects move (unlike Q-Index). Thus the costs would not change even if all the objects were moving at each time instant. On the other hand, the VCI approach is very sensitive to the number of queries.

Combined Indexing Scheme

The results show that query indexing and safe region optimizations significantly outperform the traditional indexing approaches and also the VCI approach. These improvements in performance are achieved by eliminating the need to evaluate all objects at each time step through incremental evaluation. Thus they perform well when there is little change in the queries being evaluated. The deletion of queries can be easily handled simply by ignoring the deletion until the query can be removed from the Q-index. The deleted query may be unnecessarily reducing the safe region for some objects, but this does not lead to incorrect processing and the correct safe regions can be recomputed in a lazy manner without a significant impact on the overall costs.

The arrival of new queries, however, is expensive under the query indexing approach as each new query must initially be compared to every object. Therefore a sequential scan of the entire object file is needed at each time step that a new query is received. Furthermore, a new query potentially invalidates the safe regions rendering the optimizations ineffective until the safe regions are recomputed. The VCI approach, on the other hand, is unaffected by the arrival of new queries (only the total number of queries being processed through VCI is important). Therefore to achieve scalability under the insertion and deletion of queries *combined scheme* works best. Under this scheme, both a Q-Index and a Velocity Constrained Index are maintained. Continuous queries are evaluated incrementally using the Q-index and the *SafeRect* optimization. The Velocity Constrained Index is periodically refreshed, and less periodically rebuilt (e.g. when the refresh is ineffective in reducing the cost). New queries are processed using the VCI. At an appropriate time (e.g. when the number of queries being handled by VCI becomes large) all the queries being processed through VCI are transferred to the Query Index in a single step. As long as not too many new queries arrive at a given time, this solution offers scalable performance that is orders of magnitude better than the traditional approaches.

Key Applications

Mobile electronic devices that are able to connect to the Internet have become common. In addition, to being connected, many devices are also able to determine the geographical location of the device through the use of global positioning systems, and wireless and cellular telephone technologies. This combined ability enables new location-based services, including location and mobile commerce (L- and M-commerce). Current location-aware services allow proximity-based queries including map viewing and navigation, driving directions, searches for hotels and restaurants, and weather and traffic information.

These technologies are the foundation for pervasive location-aware environments and services. Such services have the potential to improve the quality of life by adding location-awareness to virtually all objects of interest such as humans, cars, laptops, eyeglasses, canes, desktops, pets, wild animals, bicycles, and buildings. Applications can range from proximity-based queries on non-mobile objects, locating lost or stolen objects, tracing small children, helping the visually challenged to navigate, locate, and identify objects around them, and to automatically annotating objects online in a video or a camera shot. Another example of the importance of location information is the Enhanced 911 (E911) standard. The standard provides wireless users the same level of emergency 911 support as wireline callers.

Future Directions

Natural extensions of these techniques are to support other types of important continuous queries including nearest-neighbor and density queries. Query indexing can easily be extended for these types of queries too. More generally, there are many instances where it is necessary to efficiently maintain an index over data that is rapidly evolving. A primary example is sensor databases. For the applications too, query indexing can be an effective tool.

An important area for future research is the development of index structures that can handle frequent updates to data. Traditionally, index structures have been built with the assumption that updates are not as frequent as queries. Thus, the optimization decisions (for example, the split criteria for R-trees) are made with query performance in mind. However, for high-update environments, these decisions need to be revisited.

Cross References

► Continuous Queries in Spatio-Temporal Databases
► Indexing the Positions of Continuously Moving Objects

Recommended Reading

1. Kollios, G., Gunopulos, D., Tsotras, V.J.: On indexing mobile objects. In: Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), June (1999)
2. Prabhakar, S., Xia, Y., Kalashnikov, D., Aref, W., Hambrusch, S.: Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. IEEE Trans. Comput. **51**(10):1124–1140 (2002)
3. Wolfson, Ouri, Xu, Bo, Chamberlain, Sam and Jiang, L.: Moving objects databases: Issues and solutions. In: Proceedings of the SSDBM Conf. pp. 111–122 (1998)