Scaling Location-based Services with Dynamically Composed Location Index

Bhuvan Bamba, Sangeetha Seshadri and Ling Liu
Distributed Data Intensive Systems Laboratory (DiSL)
College of Computing, Georgia Institute of Technology
Email: bhuvan,sangeeta,lingliu @cc.gatech.edu

Abstract

Performance and scalability of location-based services (LBSs) are crucial to the wide deployment of mobile enterprise systems. Location queries are a fundamental capability of LBSs. Conventional approaches to location query processing have been centered on an object-centric architecture where static and moving objects are processed under a unified framework through an object index or query index. In this paper, we identify and exploit the performance benefit of a location-centric framework by promoting a clean separation of location query processing over static objects from query processing over moving objects. We show the performance benefits of treating locations as first class citizens *instead of objects. Concretely, we develop a* federated location indexing *scheme for processing location queries over static objects and maintain a grid-based object index for processing queries over moving objects. Our experimental results demonstrate that the location-centric framework enables highly efficient processing of different types of location queries in a mobile environment, prevails under all types of query workloads compared to the object-centric approaches, and in some scenarios it requires as low as only* 6% *of the IOs required by a corresponding object index for query evaluation.*

1 Introduction

Location queries can be classified into two major categories depending on the motion properties of the target objects being queried. The first class of location queries consists of *location queries over static objects*. An example of such location queries is *"show me the locations of all gas stations within 5 miles"*. In this example, the query issuer is called the *focal object* of the query and may be a driver on the road or a pedestrian on foot. The second class of location queries comprises of *location queries over moving objects*. An example of such queries is *"find all customers who are looking for cabs and are within five miles of my current position"*. In this example both the target objects of the query and the issuer of the query are mobile users. Most existing research efforts to date have been dedicated to spatial and temporal methods for indexing objects or indexing queries [14, 16, 10, 4]. Even though some indexing techniques such as TPR-tree [16] have shown higher effectiveness in indexing moving objects than static objects, few studies provide an in-depth understanding of the potential performance benefit of separating the indexing over static objects from the indexing for moving objects.

We show that a clean separation of query processing over static objects from query processing over moving objects provides significant performance gains for scaling the processing of queries over a mixed workload of both types of objects. Processing requirements for evaluating queries over static objects are significantly different from those for queries over moving objects. For example, queries over static objects are typically issued by mobile clients on the move. The target objects of the queries are static objects, but the spatial query range changes as the query issuer (i.e., the focal object of the query) moves on the road. Thus, the set of static objects to be retrieved from the database is highly dependent on the motion behavior and the current location of the query focal object. In contrast, queries over moving objects need to continuously track the positions of moving objects in the vicinity of the focal object. The ability to efficiently track the precise positions of the target moving objects is critical for processing such queries in terms of both result quality and system performance.

Bearing these observations in mind, we develop a location indexing framework to advocate a clean separation of static objects from moving objects in terms of both spatial indexing structure and location query processing. By promoting locations as *first class citizens*, we show that the location indexing framework enables scaling of location-based services. Concretely, we propose to build a location index for managing all the static objects in terms of their geographical locations in the real world. Our experimental evaluation shows that our location indexing framework can provide fast access capability with low maintenance cost compared to existing object indexing schemes. There are several factors for such performance gains. First, location index is served as a primary clustered index, in the sense that all static objects stored on disk are ordered by their locations instead of their object identifiers, thus requiring fewer disk IOs compared to the existing object indexing schemes. Second, managing moving objects, their location

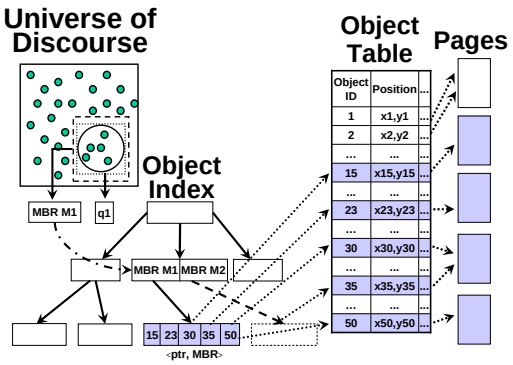


Fig. 1: Object-Based Modeling

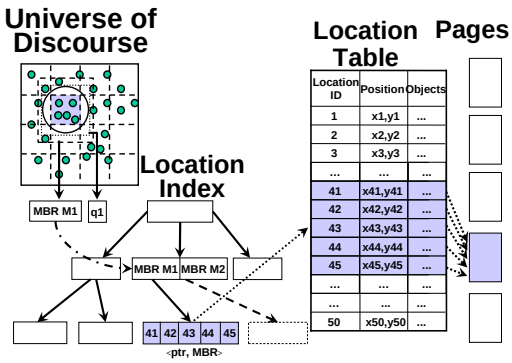


Fig. 2: Location-Centric Modeling

updates, and queries over moving objects using a separate object index, considerably reduces the retrieval and maintenance costs of indexing structures for both static and moving objects.

The processing efficiency of location queries over static objects can be greatly enhanced along several dimensions, including reduced index size, improved disk locality, and fast searches at varying granularity of spatial approximations. Also, by employing a grid-based indexing scheme for processing location queries over moving objects, we can reduce index maintenance cost in the presence of location updates of moving objects. Speed up for query evaluation is achieved by parallel processing, specifically using a location index for queries over static objects and grid index for those over moving objects.

2 Motivation and Problem Statement

We dedicate this section to discuss the motivation for our location-centric framework for query processing. An example is used to illustrate the key differences in terms of processing requirements between our location centric framework and the object-centric approach, and the potential inefficiencies associated with the object-centric framework, especially in the context of processing location queries over static objects.

Figure 1 shows a set of static objects in the *Universe of Discourse U*. Data for each object in *U* is represented by a corresponding entry in the object table. In an object-centric framework, the data maintained in the object table is indexed by the *'object ID'* using a R-tree based object index which uses *minimum bounding rectangles* (MBRs) to index the objects. Consider a location based query *q1* over this object table as shown in Figure 1. Assuming that an R-Tree based object index exists, the query retrieves a set of index entries that correspond to the list of objects belonging to the region being queried by *q1*. Now for further query processing the object entries need to be retrieved from the table using this index on the object ID column. However, since the data organization on disk is based on object ID rather than spatial locality, the objects being queried, though in close vicinity, may be spread across multiple disk pages resulting in highly inefficient disk bandwidth utilization. For instance in this example, five objects reside in the region being queried by *q1*. Thus as many as four different *disk pages* may need to be fetched. Furthermore, if we incorporate moving objects into this model some of the MBRs may need to be split into smaller ones as new mobile objects join the system, not only increasing the number of MBRs to be maintained in the R-Tree based object index and the index search cost, but also increasing the maintenance cost of the index since the location updates of moving objects often cause the mobile objects to be moved from one MBR to another.

We reach the following conclusions from the above discussion. First, when the system needs to maintain an index for large number of static and moving objects, such maintenance cost can be significant, since each position update of a moving object requires at least one index search and one update at each of the corresponding index nodes. Second, the spatial locality of the objects is the dominating property for both static and moving objects in a mobile environment. Location queries typically attempt to retrieve objects of interest within a particular spatial region in the vicinity of the focal object. Thus static objects or moving objects within certain spatial vicinity are typically accessed together. Access of static objects is often separated from the access of moving objects due to the fact that location queries are typically targeted at either static objects or moving objects, but rarely targeted at both types. Thus mixing static and moving objects in one index is not an optimal solution for processing location queries. Last but not the least, object-based indexing and storage of static objects fail to capture the spatial locality-based access patterns for both disk access and processing of queries over static objects. Thus, object indexing and object-based disk management will lead to inefficient retrieval of large number of irrelevant object tuples in order to find the target objects relevant to a location query.

In order to benefit from the properties associated with static objects, we develop a location indexing framework that cleanly separates static objects from moving objects. The basic principles for making such a clean separation include (1) creating and maintaining separate indexing structure for static objects and moving objects; (2) developing location index to speed up the retrieval of static objects where location is used as the *primary key* instead of objects for both index search and disk access; and (3) developing a dedicated object indexing structure for managing moving objects.

Our location indexing framework offers several advantages. First, it offers significant performance gains by separating operations over moving objects from the retrieval of static objects. Second, the location indexing framework encourages higher level of parallel processing since queries over static objects can be processed independently from queries over moving objects, leading to another level of performance and throughput enhancement. Figure 2 displays our approach for location-centric modeling of static objects. Our approach first divides the entire universe of discourse into *spatial regions* based on factors such as spatial locality, object density and page size. The example in Figure 2 displays a simple grid-based division of the region. The static object data is organized in a location-centric manner with spatial locality of objects being mapped to locality of the data on the disk. By enabling the use of spatial locality to access static object data on disk, we improve query processing by making more efficient use of disk bandwidth. For example, as shown in Figure 2, the data for the five objects relevant to query *q1* can now be retrieved in a single page fetch. Armed with these insights for location-dependent data, we next describe our framework for modeling and indexing static data.

3 Location-Centric Framework

In this section we describe our approach towards modeling and indexing location-dependent data in terms of location tuples rather than object tuples. Our approach modifies the object-based modeling and indexing approach to emphasize on modeling static objects with their location as the primary key instead of object identifiers. As a result, each relevant location in our model has one or more static objects associated with it. A location-centric model for static objects requires: (1) modeling object tuples as location tuples and, (2) constructing a location index to efficiently answer queries associated with static objects.

3.1 Location Centric Data Modeling

Our location-centric framework stores and retrieves static objects in terms of their spatial locations. In order to uniquely identify the static objects using their spatial locations, we build a grid-based overlay on top of the geographical area of interest, namely the Universe of Discourse, and use this grid-based layout to determine the static objects that can be stored and retrieved together.

The task of creating a location table consists of the following steps.

Step 1 - Determining grid partition parameters: The first step in building a location table is to determine the grid partition parameter β, which determines the size of each cell depending on the maximum number of locations permitted in a single cell. In general, the grid may consist of cells of different sizes dependent upon the distribution of locations of interest in the Universe of Discourse. Areas in the Universe of Discourse, which have higher density of relevant locations, will need to be divided into smaller cells to ensure that the number of locations in each cell does not exceed the system-defined maximum limit. The decision on the cell size is based on a number of factors, including the density distribution of the static objects, the page size of the disk access, the size of a location tuple, to name a few. The goal is to accommodate the data related to all location points within one cell in a single disk page. We refer readers to our technical report [1] for a discussion on determining the appropriate values for the grid cell size.

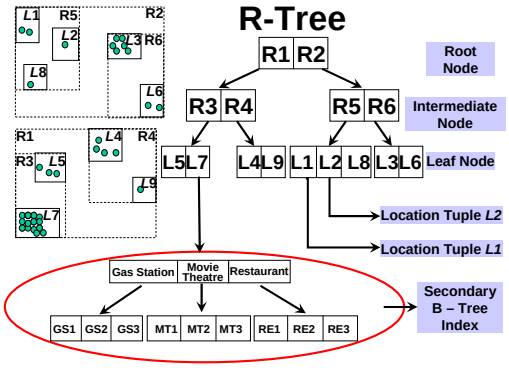


Fig. 3: Location Index – An Example

Step 2 - Obtaining relevant locations: Location can refer to a position or a spatial region. In our framework, each location entry in the location table refers to a spatial region represented in terms of grid cell. Each entry in the location table is uniquely identified by its location ID and may refer to multiple static objects. We scan the static object table and obtain relevant locations in terms of the grid cell in which they reside.

Step 3 - Generating location identifiers: In the next step, we map each location entry to a single cell on the grid and assign a *location identifier (lid)* to uniquely identify the list of locations within each cell. Each location in a single cell is assigned a unique *lid*, with all locations belonging to a particular cell being assigned location identifiers in sequential order. The order of the location identifiers will also determine the order in which the tuples are arranged on disk. Note that ordering of the tuples is important for efficient retrieval of static object data as discussed in section 2.

Step 4 - Associating objects with each location: In the final step, all static objects associated with a location entry in the location table are assigned to the location. We store the object identifier oid and associated values for other attributes of the static objects as vectors related to this particular location tuple.

In case of skewed distributions, we can construct a quadtree-based structure instead of grid structure. Also, if a particular location tuple has a large number of associated objects leading to an overflow, we can create an overflow table and provide a pointer to this overflow table which will store the list of objects associated with this location.

3.2 Location Index

In this section we give a detailed description of the various aspects of the location index. The design of our location indexing scheme follows a number of basic principles. First, we model spatial locations in terms of two dimen-

sional geographical coordinates and extend the R-tree data structure [5] to model locations as first class citizens. For example, leaf nodes in our R-tree based location index refer to a set of locations instead of a set of objects. Second, we build the R-tree based location index bottom up through sorting rectangles and merging nearby rectangles to form a hierarchical indexing tree structure. There are several ways to sort the MBR rectangles, such as tree-based variations [5, 2, 18, 3, 17] and methods that use space filling curves [9, 7]. According to [9], two dimensional Hilbert curve through centers only (2D-c Hilbert) achieves the best clustering among all space filling curve algorithms. In this algorithm, each data rectangle is represented by its center only. The Hilbert value of the center is the Hilbert value of the rectangle. The third design principle for building a high performance location index is to balance the access latency to all locations. We build a location index in three steps. Figure 3 illustrates the building of a location index by example.

Step 1 - Determining MBRs associated with each location: The first step in building a location index for fast retrieval of static objects is to determine the MBR associated with each location identifier. Figure 3 displays a set of locations *(L1, L2,..., L9)*, where each location is represented by its MBR. A location can be a region of any shape and be approximated by its *MBR*.

Step 2 - Constructing R-tree: By sorting the *MBRs* for locations, an R-tree is constructed bottom up for indexing the unique locations. The leaf nodes in the R-tree structure contain entries of the form $\langle ptr, MBR \rangle$, where ptr is a pointer referring to a particular location entry and MBR is the minimum bounding rectangle enclosing this location. Intermediate level nodes contain entries of the form $\langle childptr, MBR \rangle$, where $childptr$ is a pointer to a lower level node in the R-tree and MBR is a region enclosing the MBRs for all entries in the child node.

Step 3 - Adjusting leaf node pointers: Some leaf nodes may have locations associated with a large number of static objects (hundreds or thousands of offices, stores, restaurants, for example). One way to handle this situation is to create a secondary index over some other attribute associated with the static objects at this location, speeding up the access to these objects. The attribute that is frequently used in the filter conditions of location queries is a natural candidate to build such a secondary index. For example, in Figure 3 location *L7* has a large number of buildings associated with it. In this step, a second level B-tree index is constructed over the attribute *Associated Business* for the static objects associated to the *L7* location, allowing us to access these objects in alphabetical order of their *Associated Business*. The pointers for leaf nodes referring to such locations are directed at the root node of the secondary B-tree index over the static objects. For other locations in the leaf nodes like *L1* or *L2*, which have only a few objects associated with each location, the leaf node points to the location tuple that contains these objects. In general secondary B-tree indices are maintained for locations with large number of static objects. The decision is primarily based on the trade-offs involved in maintaining B-tree indices and the advantage gained in query performance. Our current simulator constructs B-tree indices for those locations that are associated with more than a specified number of objects.

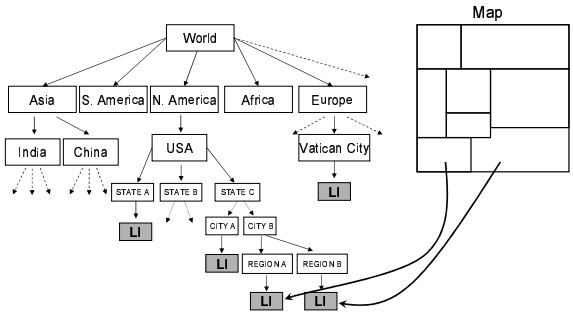


Fig. 4: Dynamically Composed Location Index

An update on the location index requires a search on the R-tree index to first locate the relevant location which needs to be updated. The list of objects or the B-tree index on the objects is then appropriately updated. Note that the R-tree index is fixed and *ideally* no locations are added to or removed from the index.

3.3 Dynamic Composition of Location Index

The location indexing approach proposed above can be used to index all relevant locations of interest in the entire world. This requires engineering modifications to enable the system to handle queries efficiently. It is impossible to handle all locations of interest in a single LI; multiple LIs are required to handle the vast magnitude of location-dependent data that is generated. In order to answer queries efficiently, it is important to build a dynamically composed location index (DCLI) from several smaller LIs. This is handled by engineering the DCLI as a hierarchical structure where we index locations with increasing resolution in a hierarchical manner.

Due to space constraints, we briefly explain our methodology for constructing a DCLI. The root node of the DCLI represents all relevant locations of interest in the entire world. The next level in this hierarchical structure represents the different continents which are further divided according to the geographical extent of countries. Larger countries will be divided according to natural geographical boundaries; for example, USA can be further represented by the states at the next level of the DCLI. The geographical division can be further performed by splitting states into counties, cities and so on till we are able to identify a region with has a reasonable number of locations of interest which can be inserted in a single LI. In order to direct queries to the appropriate LI, we maintain a hashmap which stores a hash of the path to the LI beginning at the root node of the hierarchical structure (the *World* node). Each hash key for a particular path is mapped to the root node of the LI for the end point of the path. For example, the path $World \rightarrow Europe \rightarrow Vatican\ City$ is hashed and stored as a key in

our hashmap. The value for this key points to the database server which stores the LI for the region *Vatican City*. This mapping enables us to direct all queries to the appropriate LI based on the *region of interest* for the query.

4 Location Query Evaluation

In this section we describe our approach for evaluating different types of location queries, simultaneously discussing the concepts associated with each situation. We provide evaluation techniques for two representative types of location queries: *Moving Location Queries over Static Objects* and *Moving Location Queries over Moving Objects*. In order to handle moving queries, updating positions of mobile objects is essential for the precision and freshness of query results. However, frequent updates to the database server are expensive in terms of both communication costs and CPU and Disk IO costs for update processing and index maintenance at the database server. A number of techniques have been proposed to handle the location update of mobile objects, which attempt to balance the contrasting requirements of precision of query results and limited bandwidth usage.

A naive approach for updating positions of mobile objects is the *periodical update* of object positions in which each moving object reports its current position after an interval of time. However, this approach suffers from low precision. Moreover, the approach also leads to a heavy load on the database server as motion updates to the server may need to be synchronized in order to compute query results consistently. Modeling motions of the moving objects for predicting their positions is another commonly used method in moving object indexing [10]. Motion modeling uses approximation for predicting the position of a moving object based on its available motion parameters using techniques such as *dead reckoning*. The disadvantage with dead reckoning arises from the fact that the method is based on estimation, each object needs to sample its motion parameters at regular intervals and may not necessarily report all updates as soon as they occur. Another problem with these methods is that they might exclude some relevant results.

4.1 Moving Object Indexing

The objective of designing a moving object index is to find a data structure that can effectively capture the dynamics of moving objects in terms of their position changes and be capable of retrieving moving objects in a specified location efficiently. We propose to use a grid based index structure with pre-defined cells to model and track the movement of mobile objects. Concretely, we divide the Universe of Discourse in which mobile objects move from one location to another into a number of cells of size α. Each moving object will map its current position to a particular grid cell in which it resides. Thus the system uses the current grid cell of the moving object to track its whereabout. Each moving object is responsible for initiating a position update when it moves from one α-cell to another. The database server is only aware of the current grid cell for each moving object and process location queries over moving objects using the current α-cell as the position for each moving object.

There are several advantages of using the grid-based approach for building a moving object index. First, it does not make any assumptions regarding the motion parameters of moving objects. Second, the approach is guaranteed to return at least all relevant results for a location query and updates results as motion updates are received. Last but not the least, motion updates are asynchronous and each object triggers updates when it moves from one cell to another; as updates do not occur at the same time the load on the database server is distributed.

4.2 Query Evaluation Procedure

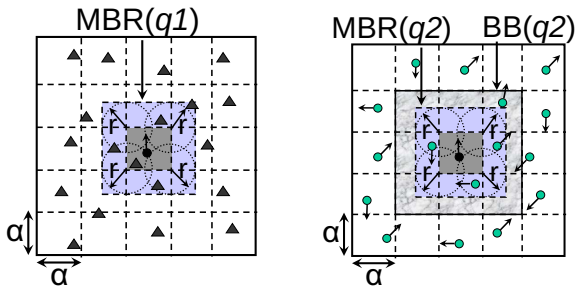
We now briefly describe the evaluation procedure for different types of location queries.

Moving Query over Static Objects: The procedure for evaluation of moving queries over static objects is illustrated by an example displayed in Figure 5(a). A location query *q1* is associated with a focal object; the position of this focal object is denoted using the current α-cell in which it is located. This α-cell is displayed using a dark grey cell in the figure. As the actual position of the object may lie anywhere inside this α-cell the bounding box for the query comprises of the MBR for circles with radius *r* (radius of query) drawn at the four corners of the α-cell. The light grey area in the figure displays the bounding box for the query *q1*. Any static objects associated with locations lying inside this bounded region will form the answer for the query. As long as the focal object of the query remains inside its α-cell, the query result will remain unchanged. When the focal object moves to another cell, the query results need to be revised as the bounding box for query *q1* will change. Note that the bounding box and MBR for the query are the same in this case.

Moving Query over Moving Objects: The procedure for evaluating moving queries over moving objects is illustrated by example in Figure 5(b). The MBR for location query *q2* in Figure 5(b) is defined in a similar manner as the MBR for location query *q1* in Figure 5(a). The bounding box $BB(q2)$ is the set of α-cells intersecting the MBR of the query. Any moving objects lying within the bounding box are potential candidates for answering the query and may be returned as the results for the query *q2*. Each α-cell is associated with a set of queries that have their bounding box intersecting with this α-cell. When an object moves from one α-cell to another the set of queries associated with both α-cells may need to be updated. For each moving query over moving objects, the set of moving objects residing in its bounding box constitute the result of the query.

5 Experimental Evaluation

This section describes three sets of experiments for evaluating the performance and effectiveness of our location index framework. The first set of experiments analyzes the behavior of the location index. The second set of experiments exhibits the advantages of location indexing over ob-



(a) Moving Query Static Objects

(b) Moving Query Moving Objects

Fig. 5: Location Query Evaluation

ject indexing for evaluating queries over static objects. The third set of experiments considers a realistic environment comprising of static and moving queries over a set of static and moving objects.

5.1 System Parameters and Setup

For all our experiments, we consider the Universe of Discourse to be a rectangular region expanding around 500,000 sq. miles. Moving or static queries over moving or static objects are considered in each scenario. Moving queries are assigned range values from the list $\{1, 2, 3, 4, 5\}$ miles using a Zipf distribution with parameter 0.6. Similarly, static queries are assigned side range values from the list $\{2, 3, 4, 6, 8\}$ miles using a Zipf distribution with parameter 0.6. The above mentioned parameters and default object densities closely follow previous work in the area; objects and queries are randomly distributed over the Universe of Discourse. Moving objects follow random paths, each motion update will lead to a random direction and random speed being chosen for the object, with object speeds categorized into different values. We consider different classes of moving objects like pedestrians (0-5 miles/hour), slow moving vehicles (30-60 miles/hour) and fast moving vehicles (70-100 miles/hour). As each object is responsible for initiating an update when it moves from one α-cell to another, no information regarding the motion of the object is required for query evaluation. Both the location index and object index are R-tree based indices, with a 100 *page* LRU buffer, each page 4 KBytes in size. Internal tree nodes have a branching factor of 100 with a fill factor of 0.7 in order to optimize performance [5].

5.2 Location Index vs. Object Index

In this set of experiments we study the advantages of location indexing (LI) over traditional object indexing (OI). Figure 6(a) plots the size of an OI and the size of the corresponding LI. The simulation setup involves a universe of discourse (UoD) containing 100K static objects. The distribution of static objects in the UoD is varied so that the average number of static objects per location (N_{static}) increases from one to ten. As can be seen in Figure 6(a) increasing N_{static} does not affect the size of OI as it still needs to index 100K data rectangles. On the other hand, the size of LI decreases with increasing N_{static}; the number of rectangles to be indexed by LI decreases, as LI only needs to index (100K/N_{static}) rectangles. Figure 6(b) plots the total IO operations required by both indices as we vary the average number of static objects per location. We also vary the number of queries (5K-10K) over the static objects. As can be observed from the figure LI, performs better than a corresponding OI over the static objects. Even with a single object per location, the location-centric approach performs better than object-based approach as retrieval of tuples from the relational table is optimized for the location-centric approach. This is due to the spatial locality of data being mapped to locality of data on disk. For $N_{static} = 1$, better organization of data on disk is solely responsible for superior performance of LI. As N_{static} increases, the advantage associated with the LI further increases as search operations are carried out over a smaller index in case of LI. In fact, with ten objects per location on an average, LI requires only around 6% of the number of IO operations required by OI. As the number of objects remains the same throughout the experiment, the performance of OI almost remains the same as we vary N_{static}. Our simulator can provide approximations for disk IOs for relational data access which is included in the number of IOs. The evaluation times for the same scenario, as shown in Figure 6(c), show that evaluation over LI is much faster than evaluation over OI for static objects (except for $N_{static} = 1$).

5.3 Location Index Performance

Figure 7 plots the query evaluation times for a set of queries over the location index as the size of the index increases. The size of the location index is determined by the number of rectangles indexed which depends on the number of relevant locations in the *UoD*. As for any index structure, the query evaluation performance for the location index declines as the size of the index increases (Figure 7). The horizontal lines at $t = 30$ sec. and $t = 60$ sec. help us benchmark the index performance. Depending on the average number of expected queries the service provider may guarantee a certain *Quality of Service(QoS)*, measured in terms of minimum query evaluation intervals at which user queries can be answered. For example, as can be seen from the figure, for a *QoS* guarantee of $t_s = 30$ sec., if the service is expected to handle 2.5K queries on average, then this location index can index around 115K locations. However, if the service is expected to handle 5K queries, the maximum number of locations that can be indexed falls to around 50K.

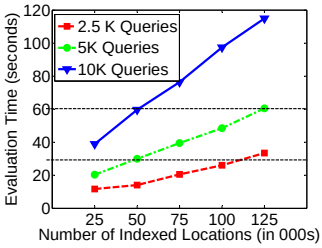
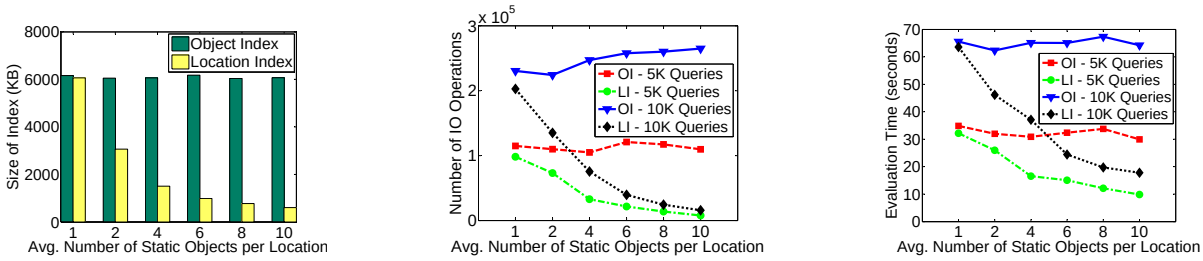


Fig. 7: Evaluation Time vs. Size of Location Index



(a) Index size (b) Number of IO Operations (c) Query Evaluation Time

Fig. 6: Comparison of Location Index and Object Index Performance

5.4 Location Query Performance Evaluation

Now we consider a scenario involving static and moving objects and display that the location-centric framework outperforms object-based modeling and indexing for a mixed workload comprising of moving and static queries. We compare the performance of three different approaches in this experiment. The first approach is the traditional object indexing approach, which requires all static and moving objects to be indexed as an object index. The second approach indexes locations of all static objects using the LI that we have developed; moving objects are still indexed using a traditional OI. We refer to this approach as the LOI approach. Our third approach indexes locations for all static objects as a LI and maintains an OI over positions of moving objects using the grid-based framework described in section 4. We call this approach the LGI approach. For this set of experiments, we perform query evaluation for a set of 20K queries over a UoD having 100K (50% static and 50% moving) objects.

5.4.1 Determining α

The parameter α determines the size of the grid cell for approximating positions of moving objects. It is important to determine the optimal value of α for efficient system performance. Figure 8 displays the update, search and total IO costs for different values of α. For this experiment half of the queries are over moving objects and the other half over static objects. The following conclusions can be reached from Figure 8. As α increases the update IO cost for the system decreases; larger values of α imply objects will have to travel greater distances to shift α-cells. For larger α values, fewer objects are expected to shift α-cells for any time interval, thus leading to fewer updates. Similarly, the search IO costs rise with increasing α values due to larger bounding boxes as explained earlier. α is set to the value which balances the update IO and search IO costs thus providing lowest total IO costs. For the current set of experiments, we can observe that $\alpha = 4$ is the ideal value.

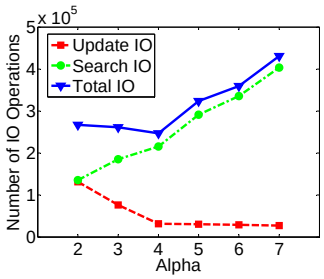
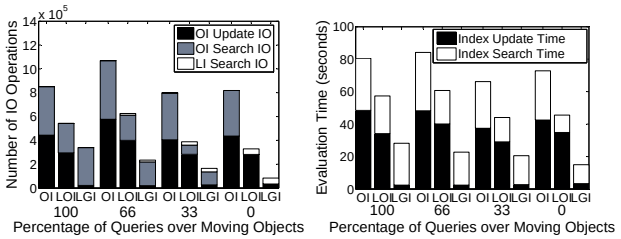


Fig. 8: IO Costs with Varying α

5.4.2 Performance Evaluation

Figure 9(a) plots the number of IO operations required for query evaluation, as we vary the fraction of queries over moving objects and explore the performance of the three indexing approaches as discussed above. The corresponding query evaluation times for all approaches are shown in Figure 9(b). Among queries over moving objects, half of the queries are static queries over moving objects and the other half are moving queries over moving objects. The IO operations consist of three different components: (a) object index update IO for moving objects, (b) object index search IO and (c) location index search IO. The OI approach has just the first two components as it does not support a location index. As can be observed from the figure LOI approach works much better than the OI approach and the LGI approach outperforms both the OI and LOI approaches. Even when all queries are over moving objects, LOI approach works better as the object index for moving objects in the LOI approach is smaller than the object index of the OI approach. This is because the object index of the OI approach indexes static as well as moving objects whereas the object index in the LOI approach indexes only the moving objects. As only half the objects are moving objects, this index is roughly half the size of the index of the OI approach. The search component of the IO for the LOI approach is much lower than the search IO required by the OI approach. Further, as the percentage of queries over moving objects decreases, the difference in search IOs required by LOI approach and the OI approach increases. As all queries over static objects are directed to the location index, the search performance of the LOI approach compared to the OI approach improves as more queries are directed to the location index. LOI approach has lower update costs too as updates are required over a smaller index compared to the OI approach. The LGI approach, further improves the associated update costs by adopting a grid-based framework for handling moving objects. The search costs for the LGI approach are a little higher than those for the LOI approach. This is simply due to the fact that the bounding boxes for the queries and MBRs for moving objects using the grid-based framework are larger than the corresponding query bounding boxes and MBRs based on



(a) Number of IO Operations (b) Query Evaluation Time

Fig. 9: Query Evaluation Performance

exact positions of moving objects. Hence, depending on the value of α, the result sets in the LGI approach are a little larger than the result sets in the LOI approach.

6 Related Work

Research on object indexing in a mobile environment has been focused either on indexing current positions of moving objects [6, 16, 10, 4] or indexing the trajectories of moving objects [14, 13, 11]. R-tree and its variants are the most commonly used indexing structures for spatio-temporal indexing of mobile objects. Indexing moving object positions poses problems due to frequent updates to the object positions. To deal with this problem [15] proposes indexing queries instead of objects for evaluating static location queries over moving objects. Some work attempts to leverage the advantages associated with object indexing and query indexing by using both types of indexing to perform query evaluation [4]. Work has also been done to introduce new indexing structures like the TPR-tree [16], B+-tree based indexing [8], trajectory-based indexing [13], and to make the R-tree more update efficient [12]. However, all research exclusively focuses on update and indexing for mobile objects. Static objects are simply considered to be a special case of moving objects where its velocity is zero, and thus are treated in a similar manner as moving objects in most of the literatures to date. Our location index and location-centric framework exploits the performance benefits of separating static objects from moving objects and exhibits significant performance gains over conventional object-centric approaches.

7 Conclusion

We have presented a *location-centric framework* that advocates a clean separation of static location data from moving objects in terms of both indexing structure and location query processing. We design a location indexing scheme to manage all static objects of interest, separately from the moving objects and their location management. Through experimental evaluation, we show that our location-centric framework has a number of advantages over object-based spatial indexing methods, such as significant gain in processing efficiency of location queries through reduced index size, improved disk locality, and fast searches at varying granularity of spatial approximations.

Acknowledgement

This research is partially funded by grants from NSF CISE SGER, CSR, and CyberTrust program as well as an AFOSR grant, an IBM SUR grant, and an IBM faculty award.

References

[1] B. Bamba and L. Liu. Scaling Location-based Services with Location Indexing. Technical report, Georgia Institute of Technology, 2006.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM SIGMOD*, 1990.

[3] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *CACM*, 18(9):509–517, 1975.

[4] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Motion Adaptive Indexing for Moving Continual Queries over Moving Objects. In *ACM CIKM*, 2004.

[5] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD*, 1984.

[6] H. Hu, J. Xu, and D. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *ACM SIGMOD*, pages 479–490, 2005.

[7] H. Jagadish. Linear Clustering of Objects with Multiple Attributes. In *ACM SIGMOD*, pages 332–342, 1990.

[8] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *VLDB*, 2004.

[9] I. Kamel and C. Faloutsos. On Packing R-trees. In *ACM CIKM*, 1993.

[10] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. In *ACM PODS*, 1999.

[11] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic Queries over Mobile Objects. In *EDBT*, 2002.

[12] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting Frequent Updates in R-trees: A Bottom-Up Approach. In *VLDB*, 2003.

[13] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *ACM SIGMOD*, 2004.

[14] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *VLDB*, 2000.

[15] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10):1124–1140, October 2002.

[16] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *ACM SIGMOD*, 2000.

[17] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[18] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A Dynamic Index for Multidimensional Objects. In *VLDB*, 1987.