VLDB Journal manuscript No.
(will be inserted by the editor)

A Motion-Aware Approach for Efficient Evaluation of Continuous Queries on 3D Object Databases

Mohammed Eunus Ali · Egemen Tanin · Rui Zhang · Lars Kulik

Received: date / Accepted: date

Abstract With recent advances in mobile computing technologies, mobile devices can render 3D objects realistically. Users of these devices such as tourists, mixed-reality gamers, and rescue officers, often need real-time retrieval of 3D objects over wireless networks. Due to bandwidth and latency restrictions in mobile settings, efficient continuous retrieval of 3D objects is a major challenge. In this paper, we present a motion-aware approach to this problem in a client-server model. Specifically, we propose: (i) representing 3D objects in multiple resolutions through wavelets to facilitate motion-aware incremental retrieval, (ii) motion-aware buffer management schemes for both client and server, (iii) an efficient index structure for 3D objects represented by wavelets, and (iv) techniques for processing group queries exploiting group motion behavior of clients. The results of our extensive experimental study demonstrate the effectiveness of our solution.

Keywords Continuous queries · Spatial indexing · Spatial databases · Wavelets · 3D Objects · Augmented-reality

1 Introduction

Recent advances in mobile computing have delivered competitive rendering capabilities, which have enabled a new level of realism for 3D representations of objects on small computing devices such as cell phones, or head-mounted displays. For example, in the LifeClipper project [1], a head-mounted display is used to offer virtually enhanced travel experiences for tourists visiting foreign locations. Virtual 3D objects are added to the view according to the current position and viewing direction of a tourist. While the current status of this particular project requires the tourist to carry a portable storage device for providing the data, we envision that users will only need to wear a head-mounted display. In our scenarios, clients can obtain 3D data on the fly through a wireless connection: a tourist might use a mobile device, such as a smartphone, to see the 3D interior details of restaurants along a street without physically entering them; an electrician with augmented-reality glasses can see 3D layouts of wiring and pipes inside a wall before a repair; a rescue officer can see the structure of a building even if the building is on fire and filled with smoke. These application scenarios all require real-time retrieval of 3D data.

M. E. Ali, E. Tanin, R. Zhang, L. Kulik
NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne
VIC 3010, Australia
Tel.: +61 3 8344 1350
Fax: +61 3 9348 1184
E-mail: eunus,egemen,rui,lars@csse.unimelb.edu.au

The data access operations in all of these applications fit into the basic client-server model. This model consists of *mobile clients*, a *server*, a large set of *3D objects* located on the server, and *wireless connections* between clients and the server. A client has a *view* attached to it. At any time, according to the client's location and viewing direction, the client retrieves all the objects within the range of its view from the server through a wireless connection, and then renders the retrieved objects on its display. As the client moves, objects that fall within the range of the view will continuously be retrieved. This view can also be considered as a window. Therefore, the total process can be viewed as a *continuous window query on a 3D object database*. To guarantee a realistic visual experience, the objects in the query window have to be retrieved at a high rate. Typically, the client and the server will be connected through a wireless network such as a mobile phone network, which has a high latency and low bandwidth [2]. Thus the wireless connection becomes the bottleneck for our application domain. Furthermore, the server's I/O costs will add to the response time observed by the clients, especially when there are a large number of clients in the system.

The data retrieval incurs the highest delay when the client changes its view rapidly. This is because (i) in the same amount of time, there are more objects that are swept by the client's view when the view changes at a high speed, leading to more objects to be retrieved per time unit from the server, (ii) for a moving client, the usable bandwidth of a connection in a network such as mobile phone networks drops compared to the bandwidth that is available for a static client [2].

In this paper, we provide a systematic solution to the problem of continuous retrieval of 3D objects for mobile clients. Our solution is based on the key observation that when a client's view is moving at a high speed, the client is only interested in and capable of absorbing a very coarse resolution of the information from the environment viewed. Even if we visualize all 3D objects in full detail, the user will not be able to see most of the details presented while moving quickly. Therefore, we represent objects in multiple resolutions and retrieve only the data necessary to visualize the objects at the required resolution based on the speed of the movement of the client's view. We use wavelets for multi-resolution representations of 3D objects. Wavelets are ideal for our needs because of the following two advantages: (i) wavelets can easily represent an object at different resolutions; (ii) for an increased object resolution, we only need to retrieve the difference between the two resolutions, which incurs only incremental costs. Our contributions can be summarized as follows:

- Motion-aware data retrieval: (i) representation of 3D objects at multiple resolutions through wavelet decomposition, (ii) retrieval of data necessary for a certain resolution, determined by the change of the speed in the view frame, (iii) incremental retrieval of the difference of two successive view frames when increasing the resolution.
- Motion-aware buffer management: a pre-fetching and caching strategy (i) at the client based on the view's movement and (ii) at the server based on the movement of groups of clients.
- Group query processing for 3D object retrieval to reduce I/O costs for simultaneous queries from multiple clients.
- An efficient index for 3D object databases in multiple resolutions based on wavelets, and pathways to the index to accelerate index traversal.

A preliminary version of this paper has appeared in [3], where we focused on 3D data retrieval for a single client. We presented the motion-aware continuous retrieval technique, motion-aware buffer management for a client and an index structure for 3D objects represented by wavelets. In this paper, we investigate further motion-aware techniques when processing a group of queries from multiple clients. First, we propose group query processing techniques by exploiting the movement patterns of co-located clients. Second, we propose a buffer management technique for the server, which buffers the data based on the movement characteristics of clients. Third, we introduce the mechanism of pathways to our index to improve its performance for continuous queries. Finally, we conduct experiments to verify the effectiveness of these new techniques.

The rest of this paper is organized as follows. Section 2 provides some preliminaries and discusses related work. For clarity, we discuss our proposed techniques in two separate sections: client-side (Section 3), and server-side (Section 4). In Section 3, we describe our client-side optimization techniques, which include motion-aware continuous data retrieval and motion-aware buffer management. In Section 4, we present our server-side optimization techniques, which include motion-aware group query processing, buffer management, an efficient index for 3D objects represented by wavelets, and a linked-index for more efficient access to the data. In Section 5, we report the results of our experimental study and finally, we conclude the paper in Section 6.

2 Preliminaries and Related Work

Location based continuous queries on point objects (e.g., location of a restaurant or a museum) have been extensively studied in recent years. However, research on continuous query processing for a 3D object database is still in its infancy. In this section, we present preliminaries and review different aspects of existing works that are related to continuous window query processing on a 3D object database. In Subsection 2.1, we present the preliminaries and our system assumptions. In Subsection 2.2, we review existing work on continuous window query processing on point objects, and then discuss buffer management techniques for continuous queries. Since we introduce a multi-resolution approach for efficient query processing on 3D objects, we discuss some background on multi-resolution modeling of 3D objects in Subsection 2.3. In this section, we also give an overview of wavelet based multi-resolution representations of 3D objects because in this paper we represent 3D objects using wavelets. Finally, in Subsection 2.4, we present existing indexing techniques for 3D object databases.

2.1 Preliminaries

Let DB be a set of 3D data objects from a three-dimensional data set stored in a server. Let Q_t be the view window (or the query window) of a client at time t in the data space. For a moving client, the view window of the client changes continuously. Let $Q_t, Q_{t+1}, ...Q_{t+n}$ be the query windows at times $t, t+1, ..t+n$, respectively, of the client. The query is issued by a client to a server through a wireless link. In this setting, we provide a systemic solution for efficient processing of continuous window queries $Q_t, Q_{t+1}, ...Q_{t+n}$ on a 3D object database DB. The performance measure of this type of query processing includes both data transfer cost and data retrieval cost from a server.

The two major determining factors of data transfer cost over a wireless link are: (i) latency and (ii) bandwidth. While new technologies, such as HSPA+, promise a relatively high bandwidth, the latency of an wireless link is still a problem for these emerging technologies. Most common cellular links are provided today by general packet radio service (GPRS) and code division multiple access (CDMA) systems, and these links have the bandwidth in the range 0.01-1 Mbps, with high one-way latency of 100-500 ms [4, 5]. Though, recently introduced wireless link based on universal mobile telecommunications system (UMTS) (e.g., HSPA+) has the peak-rate bandwidth over 20Mbps, the cell edge users only get 1.5-3 Mbps [6]. More importantly, the latency of UMTS based links is in the range 50-100 ms which is still a bottleneck [4, 7]. Moreover, wireless links have intrinsic characteristics such as variable bandwidth, delays, and unstable connectivity that affect the performance. For example, the throughput of multi-hop wireless network drops to as low as a few Kbps, even though individual wireless link evolves to higher speeds such as 20 Mbps, in addition, the mobility of a client contributes to the delays in wireless network [5, 8]. Although bandwidth and latency in wireless systems will continue to evolve, with the increasing demand from users regarding heavy weight data consumption we assume that data transfer will continue to be a contentious topic.

We propose a systematic solution that retrieves and transfers 3D objects in multiple resolutions to optimize the usage of bandwidth in a wireless network. To overcome the high latency, variable bandwidth, and unpredictable wireless links (e.g., broken links when the client enters in a tunnel), we propose new buffer management techniques based on the likelihood of visits to objects.

Though the data transfer cost is a dominant factor in delay-sensitive applications for a single client setting, the high data retrieval cost at the server may also be a major contributor to the delays when the server is overloaded with a large number of simultaneous queries. For this, we propose a suite of server-side optimization techniques that include group query processing, buffer management, and indexing 3D objects to reduce the I/O costs.

Server data processing and in particular I/O becomes a dominant factor in data retrieval, especially for large databases[9, 10], which is also the case in our application domain. More specifically, in our case, the I/O costs can largely affect the performance when the server needs to process a group of simultaneous queries on a large 3D object database. In this paper, we measure the I/O costs as the average number of disk-resident index nodes that need to be accessed for a query. While future technologies, such as Storage Class Memory (SCM) [11], may reduce the access time, the cost of the tree-traversal, i.e., searching and retrieving the required data will still remain to be a problem for large 3D data as data sets are becoming more and more detailed and complex. In this paper, we assume the current standard for I/O costs [12, 13].

The key contribution of this work is multi-resolution data retrieval techniques that allow different resolution of 3D objects to be efficiently retrieved for continuous window queries. While we use applications where the speed of a client determines the desired resolution, our approach may be less applicable to some scenarios such as in digital battleground, where the resolution requirement should be a choice of a user rather than a system parameter based on speed. Even in such a scenario, our system could be beneficial, as the underlying techniques for the data retrieval will remain the same. We conduct a set of experiments (Section 5.4) showing that other parameters such as distance of objects from the client can also be used for multi-resolution retrieval of objects to optimize the data retrieval cost.

For our application domain, we choose the client-server model over a traditional client-only solution (e.g., LifeClipper [1], or GPS navigation) where the client device pre-loads 3D objects and processes the query locally, for the following reasons: (i) It is not realistic for some databases to remain unchanged. (ii) It is not feasible for a client to download whole databases on to the mobile device for large data, e.g., 3D representation of a city. (iii) The location-based service provider may not prefer the client to download the whole database for commercial reasons.

2.2 Continuous Queries from Mobile Clients

Current research on continuous window/range queries from mobile clients focus on both static (e.g., [13, 14]) and moving objects (e.g., [15, 16, 17]). However, these techniques are commonly based on point data sets since only the locations of objects are relevant for these systems.

Continuous window query processing techniques on static point objects generally tessellate the data space based on the position of objects, and then exploit the location and/or the movements of the client to reduce communication and processing overheads. Existing approaches in this domain fall into two major categories: (i) safe region based, (ii) time-parameterized based. In safe region based techniques (e.g., [14, 18]), the server returns the query result along with a region around the client's location within which the result remains same. In time parameterized based techniques (e.g., [13, 19]), a result set (up to a future timestamp) is returned along with the validity time of the result so that the client can incrementally compute the new result.

On the other hand, for moving objects, the data space cannot be divided into distinct regions based on the objects' positions because the objects change their position continuously. Among existing research in this field, [15, 16] are of particular interest to our work as they also use motion of the query issuer for processing continuous queries on moving objects. Lazaridis et al. [16] focus on continuous queries for retrieving moving objects in virtual tour-like applications where all objects within the view are retrieved. They bound the next query window based on reported velocity of the user, and retrieve the data for all predicted query windows. This approach also reduces the I/O costs by retrieving a disk block only once and cache them in the server to reuse it as long as it is required for successive query requests. In [15], a motion adaptive indexing scheme for moving objects is proposed for efficient processing of moving continuous queries. The authors use the concept of motion-sensitive bounding boxes that automatically adapt their sizes to the motion parameter of moving objects and moving queries. Both of these approaches [15, 16] reduce the number of independent searches in the index by pre-calculating future query results. These works [15, 16] are not tailored for 3D objects. Moreover, none of the existing research utilizes the motion of clients for efficient processing continuous queries on a 3D object database. In this paper, we use the motion of a client to retrieve necessary 3D data with appropriate resolutions for the required visualization.

SINA [17] and Q-Index [20] have been proposed to optimize the continuous processing of multiple simultaneous queries on moving object data. SINA uses shared execution paradigm to optimize the performance, and proposes a hashing based technique for evaluating multiple continuous spatio-temporal range queries. On the other hand, Q-Index uses an index structure to index queries for efficient processing of continuous range queries for moving objects. Since these multi-query processing techniques only focus on point data sets, they are not able to utilize the multi-resolution nature of queries based on the motion of clients.

In summary, we are different from [15, 16, 17, 20] on a number of aspects: (i) We model 3D objects in multiple resolutions that allow us to retrieve only necessary data required by the client which is one of the key features of our methods; on the other hand in these works, since they work on point data there is no such concept of multi-resolution processing. (ii) We use certain properties of wavelets to efficiently index multi-resolution data for 3D objects, on the other hand since these approaches only use point data sets they are not suitable for 3D object indexing. (iii) We propose novel buffer management schemes by utilizing the movements of clients which does not exist in these works.

For multi-query optimization techniques, similar to [17], we use a shared execution paradigm to optimize multiple simultaneous queries at the server. However, the fundamental difference of our approach to [17] is that our group query processing technique exploits variable resolution requirements of different clients to optimize the processing of multiple simultaneous queries. Moreover, our proposed linked index is optimized for the data retrieval for a group of continuous queries. Since [20] deals with stationary queries on moving objects, this approach is not applicable for moving queries.

In our experiments, we show that our approach for 3D objects retrieval significantly outperforms approaches that do not take the multi-resolution concept into account.

Buffer Management for Continuous Queries: To reduce the impact of high latency low-bandwidth wireless links, different pre-fetching schemes for continuous retrieval of data using mobile clients have been proposed [21, 22]. Since a mobile client has a limited buffer and all of the pre-fetched data may not be used by the client, non-uniform one-dimensional motion patterns are modeled by [22] to define regions that are likely to be visited subsequently. Alternative techniques (e.g., [21]) assume linear movement of objects that use the speed and the movement direction of the client to define the region to be pre-fetched. These pre-fetching techniques do not perform well when the movement patterns are non-trivial which is commonly the case for mobile clients. Moreover, none of these techniques exploits the prediction capabilities of techniques such as the Kalman filter [23]. With the recent development of computing power of mobile devices, we argue that a mobile client is capable of deploying estimation techniques. In this paper, we propose sophisticated *state-estimation* based techniques to determine the regions that are likely to be visited by the client in subsequent timestamps. Based on these estimations, our buffer manager fetches or caches complex 3D data at the client. Furthermore, we generalize our buffer management technique for the server that buffers the data based on the grouped movement of clients.

2.3 Multi-resolution Modeling

Multi-resolution Modeling for 3D Objects: Triangular mesh based surface representation of 3D objects is common in computer graphics and geometric modeling [24]. These mesh representations are usually large data sets limiting the transmission of complex 3D data to clients over low-bandwidth networks. One common approach to deal with this problem is to decompose 3D objects in multiple level of details, and allow the application to retrieve only the necessary details of objects. To represent an object with various levels of details, a wide variety of models are available for mesh simplification on *multi-resolution* modeling. Two

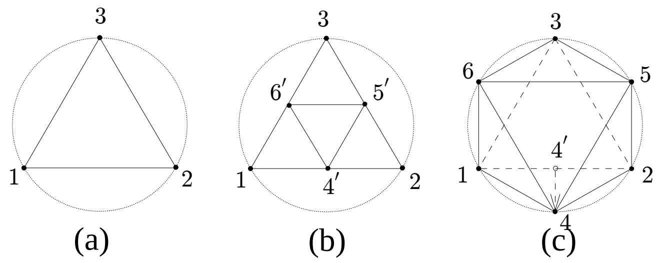


Fig. 1 (a) A coarse approximation (level 0) of a circle by a triangle, mesh M^0, (b) Mesh obtained by a splitting of M^0, (c) M^1 a level 1 approximation of the circle.

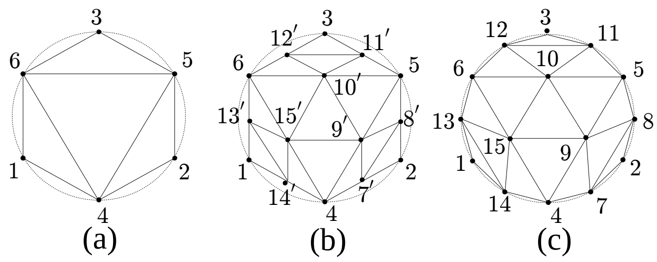


Fig. 2 (a) A level 1 approximation of the circle by a mesh, (b) Regular sub-division of the mesh at level 1, (c) M^2 as a level 2 approximation of the circle.

widely used approaches for multi-resolution modeling are: (i) progressive meshes [25] and (ii) wavelets [26]. Traditionally, graphics applications use either of these approaches to reduce the cost of 3D object rendering. However, it is known that wavelet-based approaches offer a more compact coding for progressive transmission of data and thus require less bandwidth for wireless transmissions [27, 28]. In a wavelet-based approach, a given mesh is simplified to a base mesh, together with a sequence of missing details of the original mesh. These missing details are called wavelet coefficients. Since the wireless link forms the main bottleneck for mobile clients, we use a wavelet-based approach for representing and storing 3D objects data in multiple resolutions, and furthermore, we retrieve only necessary wavelet coefficients for the visualization of 3D objects based on the speed of the client.

Wavelet Representation of 3D Objects: In this part, we give a brief overview on wavelet-based multi-resolution representations of 3D objects [26]. 3D objects can be approximated by their surfaces using triangular meshes. Let M^j be a triangular mesh representation of the surface of a 3D object at resolution j. An object can be represented in different levels of resolution by a sequence of meshes M^0,M^1,...,M^J, where, M^0 is the *base mesh* and M^J is the *final mesh*. For example, Figure 1(a) shows a triangular mesh M^0 $(1, 2, 3)$, which is a coarse approximation for the surface of the given circle.

To obtain a higher resolution approximation of the given surface, the triangle $\triangle(1, 2, 3)$ is first divided into four subfaces by introducing new vertices $(4', 5', 6')$ at edge midpoints as shown in Figure 1(b). The new set of vertices are now deformed to make the mesh to fit the surface to be approximated. For example, vertex $4'$ is shifted to a new position on the surface and is renamed as 4. The new, finer resolution mesh M^1, is shown in Figure 1(c). Since the mesh in Figure 1(b) is obtained from the mesh in Figure 1(a) by using a regular subdivision, level 1 mesh M^1 can be obtained by adding the required displacement of three midpoint vertices $4', 5'$, and $6'$. In this case, the missing details of the mesh M^0 from the mesh M^1 can be shown as the displacement of the vertices 4, 5, and 6 from the vertices $4', 5'$, and $6'$, respectively. Thus the wavelet coefficients that represent the difference between M^0 and M^1 are d^0_4, d^0_5, and d^0_6. For example, d^0_4 is obtained by $v^1_4 - \frac{v^0_1+v^0_2}{2} = v^1_4 - v^1_{4'}$. Similarly, Figure 2 shows the steps for obtaining a level 2 mesh M^2 that is a more refined approximation of the surface than level 1 mesh M^1. Each face of the mesh in Figure 2(a) is divided into four faces by introducing edge midpoints as shown Figure 2(b). Then the new set of vertices are shifted towards the original surface to obtain M^2 in Figure 2(c).

The wavelet decomposition of a mesh M^J produces a base mesh M^0 and the sets, $\{W_0, W_1, ..., W_{J-1}\}$, of wavelet coefficients. Each W_i contains a set of wavelet coefficients at level i that represents the missing details between mesh M^i and M^{i+1}. Coefficients with higher values have greater significance than that of lower valued coefficients. Since the smaller magnitude coefficients have less role to play in the overall structure of the object, the bandwidth utilization can be improved by discarding lower valued coefficients. In a selective transmission scenario, the coefficients that are necessary to modify the clients' currently available version of objects are retrieved. Thus, at a given point in time, only a subset of coefficients are needed to be retrieved by the client.

There are many other methods for decomposing the surface of 3D objects using wavelets. Our techniques described in subsequent sections also work with other wavelet-based representation of 3D objects.

2.4 Indexing 3D Objects

Indexing techniques for Multi-resolution Data: Spatial access methods such as R-tree [29] and its variants [30], k-d-trees [31] and quadtrees [32] have been extensively studied for indexing spatial objects. In R-trees, nearby objects are first grouped together to form leaf-level nodes, then these nodes are recursively grouped to form upper-level nodes until we reach to the root of the tree, where the root represents the total data space. Each entry of a leaf node maintains a pointer to a data object (or the object itself) and a minimum bounding rectangle (MBR) that encloses these objects. Each entry of an index node (or non-leaf node) maintains a pointer to a child node and an MBR that encloses all entries of that child node. On the other hand, quadtrees and k-d-trees use space partitioning-based techniques to recursively partition the space and then assign objects on to the resultant partitions. We use R-tree based indexing at our base to access multi-resolution objects.

Several spatial access methods have been proposed to speed up the retrieval of progressive mesh based multi-resolution terrain data [33, 34, 35, 36, 37]. A pioneering work, LOD-R-tree, is proposed to index different level of detail (LOD) of a terrain using R-tree [33]. Hoppe proposes an approach similar to the LOD-R-tree, but uses a two-dimensional region quadtree [34] for indexing. Shou et al. propose an improved version of the LOD-R-tree by including the visibility of data and propose a new indexing structure called the HDoV-tree [35]. Xu proposes a LOD-quadtree, in which the LOD dimension is added with the dimensions of terrain data [36]. Recently proposed direct mesh based R-tree [37] avoids unnecessary data retrieval by using a new data structure called Direct Mesh that allows the reconstruction of a terrain approximation with less I/O overhead than other approaches. All of these methods discussed above are designed for progressive mesh based multi-resolution terrain modeling, and cannot work with wavelet-based multi-resolution representations of 3D objects. Furthermore, none of these methods consider the motion of clients for optimizing the processing of continuous queries. We propose an R-tree based index using motion-awareness for 3D object retrieval in multiple resolutions using wavelets.

Indexing for Improved Traversal: An index that maintains links among neighboring nodes of the index can facilitate faster access to the data for continuous queries. Though no existing work specifically address this issue for continuous queries, there is research in other fields for data retrieval that introduce links in indexing among leaf nodes for faster traversal (e.g., [38, 39]). In [38], a leaf node maintains links with its neighbor-nodes, called ropes to facilitate a faster ray-traversal algorithm. The authors modify the binary space partitioning trees by ropes to avoid the lengthy traversal from the root of the tree. On the other hand, in a disk-based data structure named Corner Stitching [39], the rectangular objects are stitched together at their corners to facilitate faster (linear time) searching for VLSI layout editing. In this structure the empty spaces are also needed to be modeled requiring large overheads for spatial objects of varying sizes. Since, these indexing techniques are not efficient for our purpose, we introduce links in our motion-aware R-tree based index that reduces I/O and accelerates the access to 3D data for continuous queries, especially in the presence of multiple clients.

3 Motion-Aware Processing on the Client

In this section, we first discuss our motion-aware continuous data retrieval scheme that retrieves only the necessary data for the visualization based on the speed of the client's view. Then we propose a motion-aware buffer management scheme that buffers the data with high probabilities of being visited by a client.

3.1 Continuous Data Retrieval

In our continuous data retrieval scheme, we propose an approach that allows a client to incrementally retrieve 3D objects based on the change of the view of the client and its speed. In this section, we elaborate this scheme from a client's perspective, that is, how a client incrementally decides what to retrieve from the server, and then briefly discuss how the server determines the query result at the end of this section. The details of the server side query processing is given in Section 4.3.

In our approach, we assume that a client has a function to map the speed of the client to an appropriate resolution of 3D objects for the necessary visualization at a given point in time. This mapping of a speed to a resolution can be done automatically by the system in combination with some user input. For example, the system can be trained to map a speed to a resolution that satisfy expected users' requirements. Then a specific user might be interested in setting the required resolutions for different speeds based on different environmental settings. A user can tune this mapping function using a GUI with sliders depending on parameters such as the display size and available bandwidth. It is important to note that we also take the acceleration of a client into consideration in the mapping process using the motion prediction approach.

As we have discussed, a mesh representing a 3D object is decomposed into a set of wavelet coefficients. Each wavelet coefficient representing a vertex of the mesh has an associated value w in a range of $[0, 1]$. The larger the value of a coefficient, the greater the significance it has on the visualization of the object. So for the lowest resolution object, only few wavelet coefficients having larger w values are sufficient to represent the overall shape of the object. On the other hand, for the highest resolution object, all wavelet coefficients are necessary to represent every detail of the object. Let, at time t, the speed of the moving client be s_t and the view of the client be mapped to a query window Q_t. The client needs to retrieve all the coefficients necessary for visualizing the objects that fall inside Q_t. For example, when the speed is very low $(s_t \approx 0)$ the client needs to see the objects inside Q_t with the highest resolution (or full level of details). In this case all the coefficients whose values range from 0.0 to 1.0 and satisfy Q_t need to be retrieved. Similarly, when

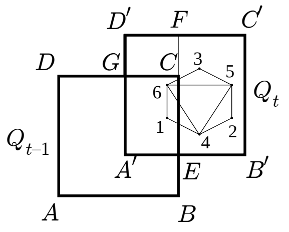


Fig. 3 Continuous data retrieval

the speed is higher (say, $s_t = 0.5$) the client needs to retrieve less details for objects inside Q_t. In this later case, the client may only retrieve wavelet coefficients whose values range from 0.5 to 1.0, since these coefficients are sufficient for the visualization of objects for this client moving at speed 0.5. It can be observed from the above discussion that to retrieve objects with required resolutions from a 3D object database, one needs to set the following parameters: a query window and a range determining the candidate wavelet coefficients for visualization with required resolutions.

Our algorithm incrementally retrieves 3D data from a server for a continuous query. A continuous query is represented as a time-ordered sequence of window queries (or query frames). As the client moves, it sends queries to the server at different timestamps. The client only retrieves the data that is not already retrieved by the previous query window. Figure 3 shows two rectangular query windows (A, B, C, D) and (A', B', C', D') for the two query windows Q_{t-1} and Q_t at times $t-1$ and t, respectively. Suppose the client has already obtained all the data for the objects for Q_{t-1}. For an object, the client needs all of its vertices that fall inside the query window and also all the neighboring vertices that connect to this set of visible vertices. These are required to properly render the objects inside a query window. For example, for the query window Q_{t-1}, the client retrieves vertices $\{1, 6\}$ that fall inside the query window and the neighboring vertices $\{3, 4, 5\}$ that connect to 1 and/or 6. Thus, at timestamp t, the client only needs to retrieve information for the region $Q_t - Q_{t-1} = (E, B', C', D', G, C)$. After receiving the region (E, B', C', D', G, C) from the client, the server divides the region along the x-axis into two query windows (G, C, F, D') and (E, B', C', F) and executes these sub-queries separately.

After retrieving the results for all the sub-queries, the server filters the results to avoid transmitting the data that is already available at the client. In this example, the server filters out those vertices that connect to the vertices that fall inside the previous query window (A, B, C, D). Therefore, the server filters out vertices $\{3, 4, 5\}$ and only sends vertex 2 to the client as the final result for the query window Q_t.

Algorithm 1 shows the steps of our data retrieval process. Let Q_t and Q_{t-1} be the query windows at time t and $t-1$, respectively. First, the algorithm finds the overlapping region O_t between Q_t and Q_{t-1}. The region N_t of Q_t, which is not overlapping with Q_{t-1}, is also determined. Then the function ***MapSpeedToResolution*** converts the speed at time t, s_t, to the resolution at time t, r_t. ***Retrieve*** is a remote function that is invoked by the client to retrieve objects from the server. It takes a set of parameters, where each element of this set consists of a group of parameters, i.e., a region and lower and upper limits for resolutions. If the required resolution r_t is greater than r_{t-1} (the resolution of the previous query window), then the client needs to retrieve additional object details for region O_t. These details are necessary to convert the objects within O_t from resolution r_{t-1} to resolution r_t. In addition, objects for the non-overlapping region N_t are to be retrieved with resolution r_t. If $r_t \leq r_{t-1}$, then objects for the region O_t are available at the client; so only objects for the non-overlapping region N_t are retrieved with resolution r_t. Finally, if there is no overlap between Q_t and Q_{t-1}, then all the objects that fall inside Q_t are retrieved with resolution r_t.

Algorithm 1: ContinuousDataRetrieval

```
1.1  $O_t \leftarrow Q_t \cap Q_{t-1}$;
1.2  $N_t \leftarrow Q_t - Q_{t-1}$;
1.3  $r_t \leftarrow$ MapSpeedToResolution($s_t$);
1.4  if ($O_t \neq \emptyset$) then
1.5      if ($r_t > r_{t-1}$) then
1.6          $R_t \leftarrow$ Retrieve($\{(O_t, r_{t-1}, r_t), (N_t, 0, r_t)\}$);
1.7      else
1.8          $R_t \leftarrow$ Retrieve($\{(N_t, 0, r_t)\}$);
1.9  else
1.10     $R_t \leftarrow$ Retrieve($\{(Q_t, 0, r_t)\}$) ;
```

Using our multi-resolution continuous data retrieval scheme, the data transmission cost can be significantly reduced by selecting objects with appropriate resolutions according to the speed of a client. However, the process still incurs a significant latency because the client needs to communicate with the server for each of the query windows. During this time, the wireless connection between the mobile client and the server may also become unstable (e.g., broken links). In such cases, the proposed incremental technique may not be sufficient for real time retrieval of 3D objects using wireless links. To help overcome these problems, we propose a client-side buffer management scheme. In our buffer management scheme, the client pre-fetches 3D objects based on the likelihood of client's interest on 3D objects in subsequent timestamps.

3.2 Buffer Management

We develop a state-estimation based motion prediction method to determine the probability distribution of the data that will be accessed by a client. Our buffer management technique is different from existing approaches in two aspects. First, it is the first probabilistic buffer management scheme that models non-uniform motion of a client in a multi-dimensional setting. Second, it exploits the multi-resolution nature of 3D data to further optimize the buffer management.

In the following subsections, we first describe different components of our motion-aware buffer management method, and then summarize the complete process.

3.2.1 Cost Model

We propose a cost model for a multi-dimensional non-uniform motion of a client. Our approach generalizes a similar model that was designed for one-dimensional non-uniform motion [22].

We first divide the two-dimensional data space, where the client can move in, into equal sized rectangular blocks, also known as sub-regions. The client's location is a point in a block within the space. We assume that the client device has an associated buffer where it can store the data. When the query window of the client moves into a new block that is not found in its buffer, the client retrieves a number of neighboring blocks from the server and stores them in its local buffer. Thus the client does not need to contact the server as long as it remains inside the buffered blocks. The latency in our system can be reduced by lowering the cache misses (i.e., when the data is not found in the local buffer) to a small value. Furthermore, the client has a limited buffer, and some of the pre-fetched data may not be actually accessed by the client. Thus the buffer management scheme for a client should also avoid the retrieval of unnecessary data to minimize the wasted bandwidth. We develop a cost model that allows a client to decide which portion of the data should be kept in the buffer such that the average cache misses is minimized for a fixed buffer size. The cache misses will be minimized when the average residence time of the client in the buffered region is maximized.

Let T_q be the total duration of a continuous query, M be the average number of local cache misses during the total duration of that query, and $N(j)$ be the number of blocks needed to be retrieved at jth local miss. Let C_c and C_t be the connection establishment and the data transfer costs, respectively, for a unit block of data with size B from the server to the client. Then the total data transfer costs for a continuous query from a mobile client can be determined as

$$C = \sum_{j=0}^{M}(C_c + C_t \times B \times N(j)). \tag{1}$$

The data transfer costs will be less for smaller values of M.

In [22], a pre-fetching model is proposed for a one-dimensional setting (see Figure 4). For this model, the data

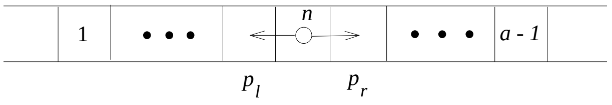
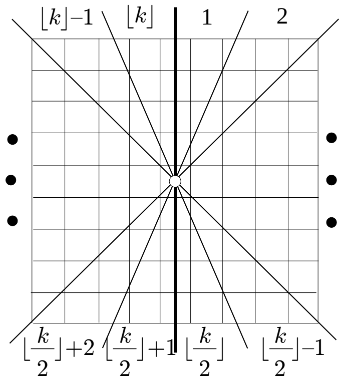


Fig. 4 One dimensional buffer.


$\lfloor k \rfloor - 1$
$\lfloor k \rfloor$
1
2
$\lfloor \frac{k}{2} \rfloor + 2$
$\lfloor \frac{k}{2} \rfloor + 1$
$\lfloor \frac{k}{2} \rfloor$
$\lfloor \frac{k}{2} \rfloor - 1$
Fig. 5 Two dimensional buffer.

space is divided into rectangular blocks where a client can move to the left block with probability p_l or to the right block with probability p_r. In this figure, the current position of the client is shown using a white circle. Assuming the client can buffer $a-1$ blocks, where $a-1>1$, then the client should buffer $(n-1)$ blocks on the left and $(a-n-1)$ on the right so that the average residence time $T_{a,n}$ that the client spends in the pre-fetched blocks is maximized. Since $M = T_q/T_{a,n}$, the average cache misses M will be minimal when the average residence time is maximized. For this, a position n_{opt} in the data space that maximizes $T_{a,n}$ can be obtained according to [22] as follows:

$$n_{opt} = \frac{\log(\frac{(\frac{p_l}{p_r})^a - 1}{a.\log\frac{p_l}{p_r}})}{\log\frac{p_l}{p_r}}. \tag{2}$$

Since the movement of a client cannot be restricted to one dimensional space, in our propose model, we partition the plane around the client into equally sized sectors. Figure 5 shows the current position of the client using a white circle, and the partition of the surrounding data space into different sectors with respect to the client's position. Each sector represents one of the k possible directions of the client. We have shown that for a one-dimensional buffer, using Equation 2 we can estimate the optimum number of blocks for each direction (left or right) when the probabilities of the client to move into each direction (p_l and p_r) and the available buffer size are given. Equation 2 uses *the ratio* of p_l and p_r to estimate the required number of blocks for each direction, thus the equation can be used to assign the blocks for any two possible directions. For a two dimensional case, we first assume a logical partition of the space that divides the data space around the client into two parts (as shown in Figure 5 where the dark solid vertical line divide the data space into two parts) and then use Equation 2 to find the number of blocks for each part. This process is *recursively* applied to find the required blocks for each of the k directions.

Let p_i be the probability that the client will move in direction i. The client needs to determine the optimal assignment of the available buffer such that the client's average time spent inside the buffered regions is maximized. Let n_i be the number of blocks that need to be assigned in direction i, and $n_{(i,i')}$ be the summation of all the blocks that need to be assigned for the directions from i to i'. Thus we have $n_{(1,k)} = a - 1$, where $a - 1$ is the buffer size. Then the buffer assignment process for different directions can be described as follows.

First, we partition all the direction probabilities into two groups such that $p_l = \sum_1^{\lfloor \frac{k}{2} \rfloor}(p_i)$, and $p_r = \sum_{\lfloor \frac{k}{2} \rfloor+1}^{k}(p_i)$. Then using Equation 2, we compute the number of blocks $n_{(1,\lfloor \frac{k}{2} \rfloor)}$ that need to be assigned for the directions from 1 to $\lfloor \frac{k}{2} \rfloor$. Hence, we have $n_{(\lfloor \frac{k}{2} \rfloor+1,k)} = n_{(1,k)} - n_{(1,\lfloor \frac{k}{2} \rfloor)}$. Similarly, we can calculate $n_{(1,\lfloor \frac{k}{4} \rfloor)}$ by using Equation 2, where $p_l = \sum_1^{\lfloor \frac{k}{4} \rfloor}(p_i)$, $p_r = \sum_{\lfloor \frac{k}{4} \rfloor+1}^{\lfloor \frac{k}{2} \rfloor}(p_i)$ and $a - 1 = n_{(1,\lfloor \frac{k}{2} \rfloor)}$. This partitioning process continues until each partition contains a single direction. After computing $n_{(1,2)}$, Equation 2 is used to calculate number of blocks n_1 for direction 1, where $p_l = p_1$ and $p_r = p_2$. Then we can have number of blocks $n_2 = n_{(1,2)} - n_1$ for direction 2. Similarly, the values $n_3, n_4, ..., n_k$ can be calculated for other directions that maximize the average residence time $T_{a,n}$. Therefore, we have $n_1, n_2, ..., n_k$ portions of the total buffer assigned for the directions with probabilities $p_1, p_2, ..., p_k$, respectively.

Using the above process, we find a set of values $n_1, n_2, ..., n_k$ for a particular ordering of k directions. There can be $k!$ possible orderings of the directions and different orderings of directions may result in different values for $n_1, n_2, ..., n_k$. Among all possible orderings, we select the result with the maximum average residence time. However, our results show that this step can be omitted as the ordering only slightly affects the average residence time.

Similarly, we can extend the model described in [22] and calculate the average number of blocks $N(j)$ to be retrieved at jth local cache miss for k directions.

Based on the proposed cost model, we can now decide what portion of a given buffer should be allocated for which direction. In the next section, we determine the probabilities that a client can take for different directions.

3.2.2 Motion Prediction

We use the *Kalman filter* [23] to predict future positions of a client from its recent locations and compute the future error covariances to determine the confidence level of those predictions. Based on the covariances and predictions, we calculate the probabilities of the client to visit neighboring regions.

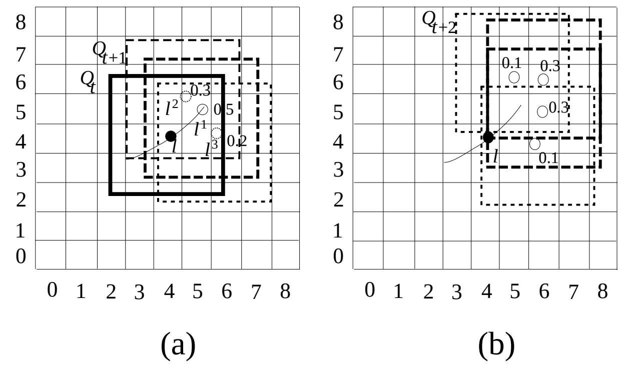


Fig. 6 Motion prediction and visiting probabilities of a client at (a) time $t + 1$ and (b) time $t + 2$.

The state S_t of a moving client at time t is defined by using the positions of this client from the h most recent timestamps, i.e., $S_t = [p(t), p(t-1), ..., p(t-h)]^T$, where $p(t)$ is a position vector. Thus the prediction of a future state at time $t+1$ is $S_{t+1} = AS_t$, where A is the transition matrix, called a one-step predictor. A can also be used for multi-step predictions. For example, a state at time $t + i$ can be calculated as $S_{t+i} = A^i S_t$. The transition matrix A can be calculated by using the recursive least-squares estimation method [40] as well as other methods [41] from recent states.

After predicting the future state, we estimate the expected confidence of a predicted state by calculating the error covariances of that state. If $\hat{S}_t$ is the predicted state and S_t be the true state, the error of the prediction is $e_t = S_t - \hat{S}_t$. Similarly, the prediction error for the state $t + i$ is $e_{t+i} = A^i(S_t - \hat{S}_t)$. From this, the covariance matrix P_t, which is a measure for the uncertainty of the predicted state $\hat{S}_t$, can be defined as $P_t = E[e_t e_t^T]$. Then we estimate the probability of the predicted state $\hat{S}_t$ conditioned on all prior estimates. Hence, the probability of a state can be estimated using a normal probability distribution [23], which can be expressed as

$$P(S_t) \sim N(E(S_t), P_t) = N(\hat{S}_t, P_t). \tag{3}$$

The predicted value $\hat{S}_t$ is the mean for the probability distribution and the variance for this prediction is obtained from the singular values of P_t.

Since the probability function is continuous, each point in the data space can have a distinct probability value. Rather than calculating the probability of each possible point location, which is a very costly operation, we divide the total space into equal sized rectangular blocks and then calculate the probabilities for different blocks to be visited by a client.

Figure 6(a) shows the current position l and query window Q_t at time t of a client. Suppose the next query window

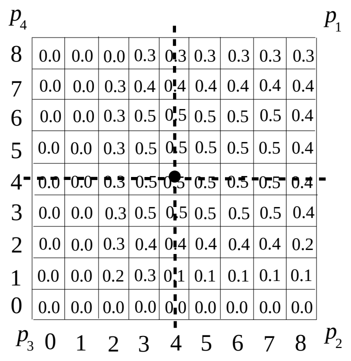


Fig. 7 Probabilities of visiting different block by a client.

Q_{t+1} can be predicted at l^1, l^2, and l^3, with probabilities 0.5, 0.3, and 0.2, respectively, by using Equation 3. Similarly, the probabilities of different blocks to be visited by the query window Q_{t+2} at time $t + 2$ is shown in Figure 6(b). By iterating this process, the surrounding blocks of the client's current position are assigned different weights by summing up these probabilities. Figure 7 shows the weights of different neighboring blocks to be visited by the client. By normalizing the weight of a block with respect to the total weight of the surrounding blocks, we can estimate the probability of a client to visit that block. Based on these probabilities, we can approximate the probabilities of a client moving in different directions. A block that intersects a partition line is assigned to one of the two partitions that owns the maximum portion of that block. If a block is equally owned by two partitions, we resolve the dispute by assigning alternating blocks to different partitions. For example, blocks (4,5), (4,6), (4,7), and (4,8) intersect the partition line between direction 1 and 2. In this case if the blocks (4,5) and (4,7) are assigned for the direction 1, then the blocks (4,6) and (4,8) are assigned for the direction 2; or vice versa. Finally, the probability of the client going in a specific direction can be obtained by summing up the probabilities of all the blocks for that direction.

After obtaining the probabilities in different directions, we apply the proposed cost model (Section 3.2.1) to determine what portion of the available buffer needs to be assigned for each direction.

3.2.3 Buffering Multi-resolution Data

The basic idea of buffering for multi-resolution data is that when a client moving at a higher speed it can buffer more objects with lower resolutions than that of a slowly moving client. Since the required buffer space for the data in a given query window varies with the speed of the client, we first estimate the buffer space for the query window as follows.

The server maintains a two-dimensional histogram H of the data space, where the data space is divided into equal size rectangular blocks. Each histogram block (i, j) contains a list of values for the corresponding block in the data space. Each entry n_s in the list corresponds to the number of wavelet coefficients required to represent the objects of that block with a particular resolution. For example, n_0 corresponds to the number of coefficients needed to visualize objects with the highest resolution (i.e., when the speed is $s \approx 0$) for the block (i, j). Similarly, n_1 represents the number of coefficients needed for the visualization of objects within the block (i, j) for the client moving at speed $s \approx 1$. To determine the required buffer space of a given query, the client acquires the data histogram of surrounding blocks from the server based on the current position and speed of the client. Then the client can estimate an approximate buffer space required for the query window Q_t at time t while moving at speed s. Let this buffer space be represented by a function $BS(Q_t, s)$, which can be defined by

$$BS(Q_t, s) = \sum_{i,j} (f_{ij} \times H_{ij}[s]), \text{where } H_{ij} \cap Q_t \neq \phi. \quad (4)$$

In this equation, i and j are the indices of a block in the data histogram H, $H_{ij}[s]$ is the number of wavelet coefficients n_s required for the visualization of objects in the block (i, j) for speed s, and f_{ij} is the fraction of the block H_{ij} that overlaps with Q_t. Since objects with lower resolution need less space in the buffer than that of higher resolution objects, $BS(Q_t, s_1) > BS(Q_t, s_2)$, where speed s_1 is smaller than speed s_2.

By using the above formulation, the client can estimate the number of blocks that can be stored in the client's available buffer based on its speed. Thus the client can reduce the cache misses by buffering data for a larger region when moving at a higher speed.

3.2.4 Buffer Management Process

So far we have discussed various parts of the motion-aware buffer management scheme. In this section we summarize the complete process and describe the necessary data structures for buffer management at a client.

The buffer management process can be summarized as follows. First, we estimate the client's path and probabilities of surrounding blocks to be visited by using the state-estimation based motion prediction model (Section 3.2.2). Then we partition the surrounding regions of the client's current position into k possible directions, and select the list of blocks to be put into the buffer from each of the directions based on the probabilities of the client to move in that direction (Section 3.2.1). For a multi-resolution buffer, when the client selects the number of blocks in each direction, it estimates the number of blocks based on the speed of client as described in Section 3.2.3. Finally, we retrieve objects from

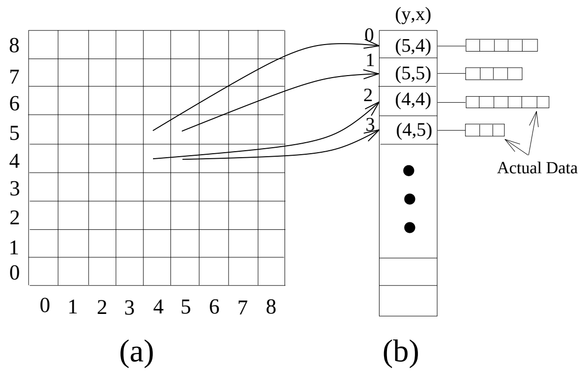


Fig. 8 (a) Status buffer and (b) data buffer at the client.

the server for the predicted blocks that are currently not in the client's buffer.

Figure 8 shows the data structures maintained at a client for buffer management. The client has a two-dimensional *status buffer* and a *data buffer*. In the status buffer, for each block the client maintains four variables: *status* represents whether the required data for the block is already in the client's buffer, *data resolution* indicates the resolution of the available data for the block, *visiting probability* stores the probability of the block being visited by the client, and *buffer pointer* points to the corresponding entry in the data buffer. The client maintains a list for the data buffer that organizes the available data in the client's buffer in increasing order of visiting probabilities. Each entry in the list is pointed by the corresponding block of the status buffer (shown by the links from Figure 8(a) to Figure 8(b)), and each entry in the list also points to a data array as shown in Figure 8(b). Figure 8 shows that the first, second, third, and fourth entries (labeled as 0, 1, 2, 3, respectively) of the list contain the data for the blocks (5,4), (5,5), (4,4), and (4,5), respectively. Since data blocks in the list are kept in increasing order of probabilities, the probability of the block (5,4) to be visited by the client is greater than that of the block (5,5). Similarly visiting probability of (5,5) is greater than the visiting probability of the block (4,4). The size of the data on each entry may vary (Figure 8(b)) and this variation depends upon the density and the required resolution of the data for the corresponding block.

In summary, the continuous data retrieval scheme (Section 3.1) depicts the incremental retrieval of 3D objects in a basic setting where we assume that there is no buffering technique at the client. The server keeps track of the previous query window to filter out the results that avoids the transmission of already available data at the client. Next, the motion-aware buffer management technique (Section 3.2) capitalizes the client's movement patterns and its buffering capacity, where the client pre-fetches new data or caches the already retrieved data in the client's buffer based on the likelihood of the client to visit the data in subsequent timestamps. Using this scheme, the data retrieved for past queries (e.g., Q_{t-3}) will still be stored in the buffer if the data has the high probability of re-use in future. In this setting, the client first decides the region of the data space that should be put in the buffer, called the *client's buffer region*. Then the client sends its buffer region to the server where the blocks of the region for which it already has the necessary data are marked. From this information, the server avoids transmitting the same data multiple times. Since the client buffer is limited, the client dynamically updates the buffer region based on its movements. In addition to overcoming the problem of high latency and unstable connection of wireless links, this buffer management technique can further reduce the data transmission overhead when the client keeps going back and forth multiple times in the same region.

4 Motion-Aware Processing on the Server

In cases like bus tours, it is common that a group of tourists will visit places in a city together. In such scenarios, the views or query windows of co-located clients (i.e., tourists) might overlap each other as more than one person may be interested in the same object at the same time. For this, to optimize the data retrieval, instead of executing each of these queries independently, we propose a suit of shared-execution techniques in the server to optimize the processing of these queries. In our current system, the server has the responsibility of combining query windows. A dedicated client in the tour could also have this task that we plan to incorporate to our future work.

Continuous query processing incurs high I/O costs for retrieving 3D objects, especially when the number of queries is large. In such cases, high I/O costs also add to the overall latency of the data retrieval system. To alleviate this problem, we develop server-side techniques introducing server-side motion-aware group query processing, buffer management, and indexing techniques, to expedite the query processing by reducing the I/O accesses.

4.1 Group Query Processing

A server often needs to handle a large number of simultaneous continuous queries from clients, possibly from overlapping regions of interest (e.g., tourists in a bus, co-located visitors inside a museum). A naive approach of executing each of these queries independently incurs high I/O costs as it needs to retrieve the same data multiple times. Therefore, in this paper, we propose a group query processing technique that overcomes this limitation. Novelty of our group query processing technique is that it combines overlapping and the multi-resolution nature of queries for efficient processing of simultaneous queries from multiple clients.

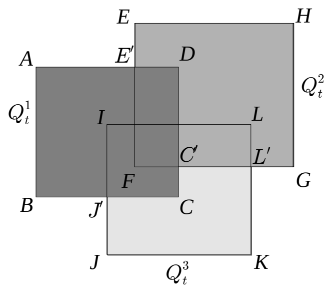


Fig. 9 Three overlapping queries Q_t^1, Q_t^2, and Q_t^3 of different resolutions from three different clients at time t.

As we have already discussed, the resolution of a query is determined by the speed of the query issuing client. Furthermore, the speed can vary largely among clients. The overall performance of group query processing can be improved by a combined execution plan that exploits the overlapping and multi-resolution characteristics of queries. The key idea of our approach is to retrieve the requested objects only once with appropriate resolutions such that it meets the requirements of all participating clients.

For example, Figure 9 shows three window queries Q_t^1, Q_t^2, and Q_t^3 received by the server from three different clients at time t. Let us assume that the resolutions of Q_t^1, Q_t^2, and Q_t^3 are r_1, r_2, and r_3, respectively, where $r_1 > r_2 > r_3$. For the required visualization of objects for all clients, the data for the overlapping regions of a group of queries is needed to be retrieved with the highest resolution of all the participating queries. Figure 9 shows the partitioning of query regions for Q_t^1, Q_t^2, and Q_t^3 into different segments based on the resolutions of these queries. In this case, all objects for the query segment (A, B, C, D) needs to be retrieved with the highest resolution r_1, i.e., Q_t^1 is executed with resolution r_1. After that, since a part of Q_t^2, (E', F, C, D), is already retrieved with resolution r_1, which is higher than the required resolution r_2, the server only needs to retrieve objects for the query segment (E, E', D, C, G, H) with resolution r_2, i.e., $(Q_t^2 - Q_t^1)$ is executed with resolution r_2 (the resolution of Q_t^2). Finally, the server retrieves objects for the remaining un-retrieved query segment (J', J, K, L', C', C) of Q_t^3, i.e., $(Q_t^3 - (Q_t^1 \cup Q_t^2))$ is executed with the lowest resolution r_3 (the resolution of Q_t^3).

Since in this method, data for each of the query segments is retrieved once with the required resolution, the I/O costs are largely reduced. In general, a lower resolution query requires less I/O than that of a higher resolution query. Based on the above theme, we propose our motion-aware sequential group query processing (SGQP) algorithm.

Algorithm 2 shows the steps of SGQP. Let $Q_t^1, Q_t^2, ..., Q_t^n$ be a group of n queries in the server at time t from different clients. After receiving these queries, the algorithm first sorts the query list in descending order of resolutions using the function *SortDescOrderOfResolution*. Then for each query Q_t^i in this sorted list, the algorithm finds the portion of the region of Q_t^i that needs to be retrieved with the resolution of Q_t^i. This can be done by removing the overlapping region of Q_t^i and the queries that require higher resolutions than that of Q_t^i. The obtained query segment for each of the query is added to a new list called NL. We use the function *FindNonOverlappingQuerySegment* to obtain NL from L. For the example shown in Figure 9, L contains Q_t^1, Q_t^2, and Q_t^3. Thus, NL will have the query segments (A, B, C, D), (E, E', D, C, G, H), and (J', J, K, L', C', C) representing Q_t^1,$(Q_t^2 - Q_t^1)$, and $(Q_t^3 - (Q_t^1 \cup Q_t^2))$, respectively.

Algorithm 2: SGQP

```
2.1 $L \leftarrow \{Q_t^1, Q_t^2, ..., Q_t^n\}$;
2.2 $SL \leftarrow SortDescOrderOfResolution(L)$;
2.3 $NL \leftarrow FindNonOverlappingQuerySegment(SL)$;
2.4 $FL \leftarrow MergeAdjacentSegmentOfSameResolution(NL)$;
2.5 $Result \leftarrow ExecuteAndFilter(FL)$;
```

For a large group of queries, since NL can contain a large number of query segments with same resolution, executing each of these query segments separately may incur large overhead. For this, the function *MergeAdjacentSegmentOfSameResolution* merges adjacent query segments having the same resolution, and inserts the segment into a new list FL. To merge adjacent query segments, we simply use a minimum bounding box that encloses all overlapping or adjacent query segments having the same resolution. Finally, the obtained query segment list FL is executed and the results are filtered out based on the actual query window list L to obtain the desired results for each client. This method can reduce I/O costs by only retrieving the data for a region once with appropriate resolution.

Though SGQP utilizes the overlapping and multi-resolution behavior of queries, all the query segments are executed sequentially. As we use a hierarchical R-tree based index structure (see Section 4.3) to access 3D objects in the server, SGQP may need to access the same node multiple times when the data of adjacent query segments is stored in the same index node. This process results in extra I/O costs. Thus, to further reduce the I/O costs, we execute all simultaneous queries in parallel to retrieve the results in a single pass over the existing index (see Section 4.3). We name this approach as motion-aware parallel group query processing (PGQP). In this approach, all queries are executed in parallel, and in general, queries with higher resolutions need to explore more nodes in the index than that of the lower resolution queries. For example, when the server executes three queries (shown in Figure 9) using PGQP, it accesses nodes in the index if the MBRs of the nodes intersect any of the three queries Q_t^1, Q_t^2, and Q_t^3. Since the data for the region covered by Q_t^1 requires to be retrieved with high resolution, more nodes may need to be accessed for Q_t^1. On the other hand, the data for $(Q_t^2 - Q_t^1)$, and $(Q_t^3 - (Q_t^1 \cup Q_t^2))$ regions are required with medium and low resolutions, respectively, and these may require less number of nodes to be accessed than that of higher resolution counterparts.

In summary, our proposed approach retrieves objects for an overlapping region of interest only once with the maximum resolution. This shared query processing approach reduces the processing load on the server. After retrieving the data for an overlapping region, the server sends the retrieved objects to each client with the resolution requested by that particular client. For example, suppose an object o is requested by three clients with resolutions r_1, r_2 and r_3, where $r_1 > r_2 > r_3$. Then the server retrieves the object o with resolution r_1, but it sends the object to three clients with three different resolutions r_1, r_2, and r_3, respectively. This is easily doable due to the properties of wavelets that allow us to represent a lower resolution of an object as a sub-set of the higher resolution object.

It is noted that in our approach, the server sends the result objects to a client with the requested resolution, and thus the quality of service is assumed to be guaranteed for all clients over time. However, there may be situations, where the server can be overloaded with large numbers of simultaneous queries from multiple clients. In these situations, the multi-resolution based shared execution scheme can be useful to reduce the workload of the server by making a tradeoff with the quality of the retrieved data. For example, in an overloaded situation the server can execute a query for a lower resolution than the requested one, and return the objects with a lower resolution to the client. The detail tradeoff analysis between the quality of service and the workload of the server will be a subject of future study.

4.2 Buffer Management

The purpose of the client-side buffer management was to reduce the high latency and the consumption of the bandwidth of wireless links. In this section, we adopt a buffer management scheme for the server that aims at reducing the I/O costs. Our buffer management scheme models the group movement of clients and buffers the data that has higher probabilities of being accessed by multiple clients.

In many scenarios, clients generally move in a group (e.g., a tourist bus) and follow predictable motion patterns (e.g., moving through roads). We observe that the data retrieved for a client in current time may need to be accessed by the same client or other clients in successive time units. Thus I/O costs can be reduced by buffering the data for the regions that have higher probabilities to be visited in successive time units by clients. This buffering results in lower I/O costs. The server does not need to access the disk again and again for the same data for many clients.

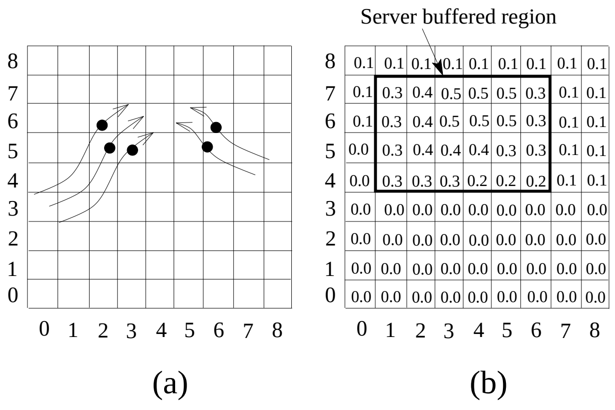


Fig. 10 (a) The paths for five different clients, and (b) combined weights of visiting probabilities of different data blocks.

For example, Figure 10(a) shows the paths of five clients as two different groups, where the centers of the query windows at time t for clients are shown using black dots. In our buffer management technique, for each client, the server estimates visiting probabilities of different data blocks by that client using the motion prediction scheme described in Section 3.2.2 that uses the movement patterns of the client. After estimating the visiting probabilities of the surrounding blocks for each client, for each block the server sums up the visiting probabilities of all clients. This process results in the combined likelihood (or the combined probabilities in normalized form) of visiting a block by clients. Finally, the server selects the blocks that have higher combined likelihood to fill its buffer. Figure 10(b) shows the different weights representing the combined likelihood of visiting different blocks by clients. In Figure 10(b), the blocks inside the highlighted rectangle are the candidates to fill the server buffer as these blocks have higher probabilities to be visited than the other blocks in the data space.

Visiting probabilities of data in different blocks continuously change with the movement of clients. The data that has higher probabilities to be visited by clients at time t may not remain the same at time $t + 1$. Therefore, the probabilities of the data to be visited by clients needs to be continuously re-evaluated as clients continue to move. Such continuous re-evaluation of the probabilities at each time instant is highly inefficient. We propose to re-evaluate the probabilities and pre-fetch the data in the server-side buffer based on these probabilities in regular intervals, named as the *pre-fetch interval*. When the pre-fetch interval is too short, the data in the buffer reflects the clients access patterns more accurately. However, in this case the buffer management cost will be high as it is necessary to re-evaluate the probabilities very frequently. On the other hand, if the pre-fetch interval is too long, as the time elapses the data in the buffer becomes stale and may not reflect the clients' access patterns accurately. Therefore, the pre-fetch interval should be carefully chosen to establish the tradeoff between these two extreme points. In this work we argue that, the system can choose a suitable value for this parameter based on the clients' behavior in the environment. It is important to note that for the client-side buffer management, we do not require any pre-fetch interval as the client starts pre-fetching spontaneously as soon as it encounters some cache misses in the client's buffer.

Our server-side buffer management procedure is quite similar to our proposed client-side buffer management technique. However, the key difference of our server-side buffer management is that it takes group movements into account. The server-side buffer management can be summarized as follows. (i) Predict possible future positions of each of the clients from their motion patterns. (ii) Calculate probabilities of each block (i.e., total data space is divided into equally sized rectangular logical blocks) to be visited by each client, and then combine the probabilities of a block to find the likeliness of all clients to visit that block. (iii) Select the candidate blocks with higher probabilities for the buffer. (iv) Pre-fetch the data for the selected blocks from the disk if it is not already available in the buffer. For a multi-resolution buffer, when multiple clients with different speeds visit a block, the data for that block is buffered with resolutions that is necessary for the lowest speed client.

The data structures for this buffer management is similar to the client-side buffer management, therefore, we omit the details for the brevity of this presentation.

Next, we propose an indexing technique for 3D objects represented using wavelets that offers improved I/O accesses.

4.3 Indexing Wavelets of 3D Objects

To the best of our knowledge, we are the first to propose an index for 3D objects represented by wavelets. We exploit one of the most important properties of wavelets known as support regions that ensure optimal data retrieval for a window query. We show how a support-region based index compared to a simple index for wavelet point data, allows us to retrieve only the necessary coefficients with the required resolution.

An R-tree [29] can be used to index the positions of wavelet coefficients and the associated values resulted from the decomposition process described in Section 2.3. The position of a wavelet coefficient is represented by a vertex (x, y, z) in a three-dimensional space and the value of the wavelet coefficient is a numerical value w. A 4D-R-tree can index a wavelet coefficient, where the first three dimensions represent the position and the fourth dimension stands for the value.

Suppose a window query $Q(R, w_{max}, w_{min})$ with the region of interest R (i.e., the view window) is submitted from a client, where w_{max} and w_{min} are the upper and lower bounds of the coefficient values (w) for the required level of objects' details within the region of interest, respectively, and $0.0 \leq w_{min} \leq w_{max} \leq 1.0$. For this, all the coefficients (vertices) that fall inside the query window are retrieved first. However, these coefficients are not sufficient for the required visualization inside the query window, because the coefficients that are associated with the neighboring vertices of these already retrieved coefficients also contribute to the visualization of objects for the region of interest R (as described in Section 2.2). Therefore, after retrieving initial sets of coefficients, we compute a bounding region that encloses all the neighboring vertices and re-execute the query for the extended region. The problem with this access method is as follows. This access method is not optimized for the retrieval costs because it cannot avoid multiple retrievals of wavelet coefficients and hence incurs high I/O costs. Moreover, additional information, neighboring vertices, are also needed to be stored for each of the vertices. To overcome these limitations and to facilitate optimal data retrieval, we utilize the support regions of wavelets.

In this section, we first give the formal definition of support regions of wavelets and discuss their important properties. Then we propose our technique to efficiently index the wavelets that can reduce the I/O costs significantly for a window query. Finally we discuss the Linked-R-tree that further reduces the I/O costs for continuous queries, especially in the presence of multiple clients.

4.3.1 Support Regions of Wavelets

The support region of a wavelet represents a region of the object to which the wavelet contributes during reconstruction of the 3D surface. For example, the wavelet coefficient associated with the vertex v_4 in Figure 1(c) has the value $v_4 - v_{4'}$ and the support region as the polygon $(1, 4, 2, 5, 6)$. If w_i^j is an ith wavelet coefficient of level j mesh M^j, then the support region r_i^j of this wavelet coefficient is the region of M^j to which the wavelet coefficient contributes during the reconstruction of the next level finer resolution mesh M^{j+1} from M^j.

It is important to note the following property for support regions. Let $W_1 = \{w_1, w_2, ..., w_n\}$ be a set of n wavelet coefficients that covers the region R_1 and let $W_2 = \{w_1, w_2, ..., w_m\}$ be another set of m wavelet coefficients that covers the region R_2, where $m \leq n$. Here, $W_2 \subseteq W_1$ and $R_2 \subseteq R_1$. If a new wavelet coefficient w_k, $k \geq n$, is added to both W_1 and W_2, and the regions affected in R_1 and R_2 by the support region of w_k are $R_1^{'}$ and $R_2^{'}$, respectively, then $R_2^{'}$ is also a subset of $R_1^{'}$ that is $R_2^{'} \subseteq R_1^{'}$. This observation is trivial from the following set of simpler obser-

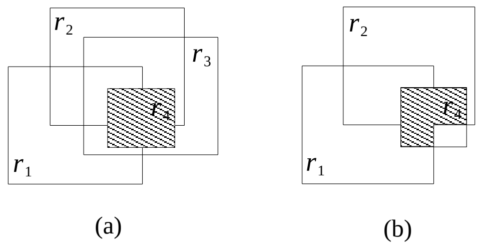


Fig. 11 Regions affected by the support region r_4 of a wavelet

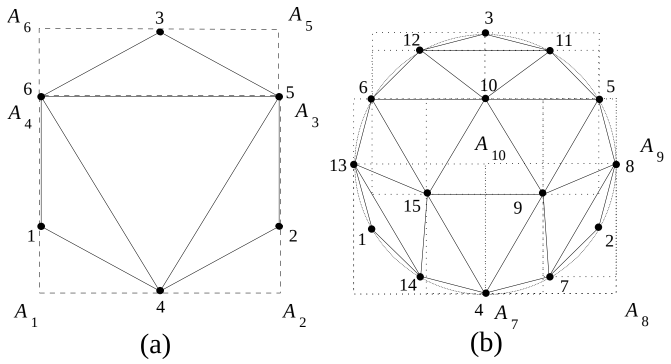


Fig. 12 Minimum bounding rectangles of support regions of (a) level 1 wavelets of Figure 1(c), and (b) level 2 wavelets of Figure 1(d).

vations. Let each wavelet coefficient w_i represent a region r_i. Then we have $R_1 = \cup_{i=1}^{n} r_i$ and $R_2 = \cup_{i=1}^{m} r_i$. When w_k is added to the set W_1, the affected region is $R_1^{'} = R_1 \cap r_k$. Similarly, we have $R_2^{'} = R_2 \cap r_k$. Since $R_2 \subseteq R_1$, then $R_2^{'} \subseteq R_1^{'}$.

For example, Figure 11(a) shows the support regions r_1, r_2, r_3 for the wavelet coefficients w_1, w_2, w_3, respectively, and the region affected by the support region r_4 of w_4 is shown as a filled rectangle, while Figure 11(b) shows the support regions r_1, r_2 of wavelets w_1, w_2, respectively, and the portion of original region affected by the support region of w_4 is shown as filled polygons. This property of a support region of a wavelet helps us design an efficient index that facilitates the efficient retrieval of data for a given query window.

4.3.2 Indexing Wavelets Based on the Support Region

We use a 4D R-tree to index wavelet-based representations of 3D objects. We utilize coefficient values and support regions of wavelets for indexing the data. The first three dimensions represent the three axes x, y, z that are used to index the Minimum Bounding Box (MBB) of the support region of each wavelet, and the fourth dimension represents the value of the wavelet coefficients w. Hence, each wavelet coefficient represents a region in the x, y, z, w plane. For simplicity of the presentation, we show the examples for a 2D object. For instance, in Figure 12(a), for a 2D object the Minimum Bounding Rectangles (MBRs) for the support regions of wavelets representing the vertices 4, 5, and 6 are shown as $[A_1, A_2, A_3, A_4]$, $[A_1, A_2, A_5, A_6]$, and $[A_1, A_2, A_5, A_6]$, respectively. Similarly, MBRs of level two wavelets are shown in Figure 12(b), where $[A_7, A_8, A_9, A_{10}]$ is the MBR for the support region of vertex 7. These examples show that the bounding rectangles can enclose two-dimensional support regions. Let the value of the coefficients (w) for level one wavelets representing vertices 4, 5, and 6 are 0.7, 0.6, 0.8, respectively, and the value of the coefficients for level two wavelets representing vertices 7-15 fall within the range [0.0-0.3]. Then we can index these wavelets using a 3D R-tree, where a part of the index is shown in Figure 13. Here, leaf node B contains all wavelets coefficients whose values fall within the range [0.0-0.6) and support regions are enclosed within the rectangle ($x_1 \times y_1$). Thus all wavelets that represent vertices 7-15 are stored in node B. Similarly, leaf node C contains all wavelet coefficients whose values fall within the range [0.6-1.0]. In this case wavelets representing vertices 4,5, and 6 are stored in leaf node C. Then leaf nodes B and C are grouped together to form an index node A.

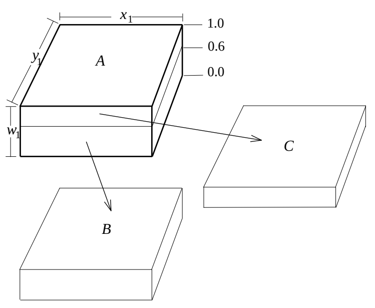


Fig. 13 A part of the R-tree (3D) based index.

Figure 14 shows an example for representing wavelet coefficients indexed by a 3D R-tree for 2D objects. In this example, x and y dimensions are used to represent MBRs of wavelets, and w corresponds to the coefficient value for the wavelet. Figure 14 shows MBRs with dotted boundary lines for three different w values 0.0, 0.3, and 0.7.

To retrieve objects with the highest resolution, a client requests a window query $Q(R, w_{max}, w_{min})$, setting $w_{max} = 1.0$ and $w_{min} = 0.0$. These values of w_{min} and w_{max} results in retrieving all wavelet coefficients irrespective of their values, such that their MBRs intersect with the query window R. On the other hand, if a client needs to retrieve objects with the lowest resolution it sets $w_{max} = 1.0$ and $w_{min} = 1.0$. In this case, it only retrieves the wavelet coefficients having value 1.0 as these coefficients are necessary for representing the overall shape of objects with the lowest resolution. In addition, the client can set any appropriate values for w_{max} and w_{min} to support progressive retrieval of objects. Let us assume a scenario where a client has all the coefficients having values greater than 0.7 for a given query window R. If the client requires objects with the

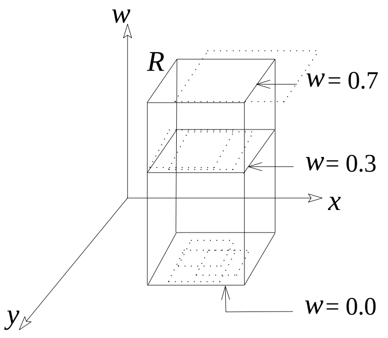


Fig. 14 Wavelet coefficients and support regions in our index

finest resolution for the same query, it only needs to send a query $Q(R, 0.7, 0.0)$ as shown in Figure 14.

In general, our motion-aware access method gives the minimum number of wavelet coefficients necessary for the query Q. Let $R_1 = \{r_1, r_2, ..., r_n\}$ be the set of support regions of n wavelet coefficients retrieved using our motion-aware index for the query Q. The set contains all the support regions that fall inside or intersect with Q. Assume that another method gives a set of support regions $R_2 = \{r_1, r_2, ..., r_n\} - \{r_k\}$ of $n - 1$ wavelet coefficients for the query Q. Here, $1 \leq k \leq n$, and hence, $R_2 \subset R_1$. In this case, the wavelet coefficient w_k associated with the support region r_k is absent. Therefore, objects inside the region $R \cap r_k$ lack some details that would otherwise be contributed by the wavelet coefficient w_k. Hence, any method that retrieves less data than that of our method is not sufficient for visualization.

We only retrieve wavelet coefficients whose support regions' MBRs intersect with the query window. There may be only a few coefficients whose MBRs intersect with the query window but the actual support regions do not intersect with the query window (e.g., when the query window intersect one corner of an MBR). In such cases, these coefficients can be easily discarded by looking into the actual support regions of these coefficients. In our implementation, we ignore these coefficients as the number of such coefficients is negligible in comparison to the total number of coefficients retrieved and these coefficients do not affect the results. Thus in our case, post-filtering of the data is not CPU-intensive.

Though the proposed index offers efficient retrieval of objects in multiple resolutions for a static window query, it is not optimized for a continuous query. Furthermore, the proposed index cannot utilize the behavior of grouped movement for multiple co-located clients. In the next subsection, we introduce pathways into the index to accelerate data retrieval for continuous queries, especially in the presence of multiple clients.

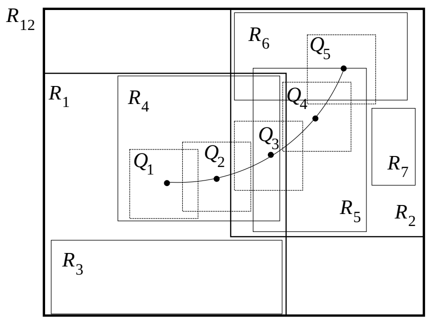


Fig. 15 Pathways and *R*-tree nodes.

4.3.3 The Linked R-tree: an Improvement to the Index

Using the above R-tree based motion-aware index, we optimize the I/O costs while retrieving objects with necessary resolutions based on the speed of a client. However, a continuous query can be seen as a sequence of snapshot queries and *each* snapshot query needs to traverse from the *root* to the leaf nodes of the index for the results. *Repeated* and *long* traversal of the index results in high I/O costs, especially when there is a large of number clients in the system. As we have mentioned before, in many applications it is common that the clients move in a group and follow common pathways. In such cases, we observe that a client can take the advantage of the traversal of the index performed by previous snapshot queries posed by itself or other clients on the same pathways. Since R-tree based index (Section 4.3.2) does not support any traversal among nodes, the query processing cannot take the advantage of continuous exploration of the data space by clients. Therefore, we propose the Linked R-tree (LR-tree) by introducing pathways to the index structure. These pathways can facilitate continuous traversal of data nodes in an R-tree. The LR-tree allows a single client or a group of clients to exploit the knowledge of the most recently accessed nodes to explore their adjacent nodes with significantly reduced I/O access.

A simple motivating scenario for the LR-tree is shown in Figure 15. In this figure, a continuous query from a moving client is represented by a sequence of five snapshot queries Q_1, Q_2, Q_3, Q_4, and Q_5. The pathway of the client is shown by an arc within the data space represented by the region R_{12}. In this example, objects are grouped into leaf nodes (level 0) R_3, R_4, R_5, R_6,and R_7. These leaf nodes are recursively grouped to form a sub-tree rooted at node R_{12}. A portion of the R-tree that indexes the data for region R_{12} is shown in Figure 16 (without dotted links among leaf nodes).

In this example, at each instant of time (e.g., t_1), the server receives a snapshot query (e.g., Q_1) and traverses the index tree from the root to one or more leaf nodes to retrieve the result. Since the index structure does not allow to visit spatially adjacent leaf nodes from the client's previous

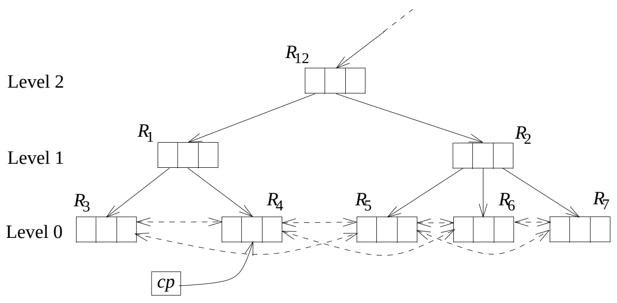


Fig. 16 *LR*-tree with pathways at level 0.

position, it cannot capitalize on the traversal done by this previous snapshot query. Thus a large I/O cost is incurred. Moreover, the same pathway can also be used by a group of clients. In such cases, for each of the snapshot queries from each individual client, the server needs to traverse the index tree from the root to leaf nodes for the results.

In our proposed LR-tree, we modify the R-tree structure such that each leaf node maintains links to its spatially adjacent leaf nodes in different directions (as shown by dotted lines in Figure 16). In addition, a pointer cp to the most recently accessed leaf node is maintained. In this way subsequent queries can follow the path by directly accessing the leaf node accessed by the previous snapshot query. In this example, at time t_1, query Q_1 needs to traverse from the root to the leaf node R_4 to retrieve objects that fall inside Q_1. cp is then updated with the address of R_4. Next query Q_2 can use this pointer to access the most recently accessed leaf block R_4. From this node, subsequent queries can then explore adjacent leaf nodes according to the moving direction of the client. In this way, for all consecutive queries Q_3-Q_5, the server can avoid the traversal from the root to leaf nodes, and can access the right block of data using less number of I/Os.

The above index structure results in reduced I/O costs. However, the reduction in I/O costs comes at the price of maintaining a large number of links at each of the leaf nodes. For example, in Figure 16, all level 0 nodes (i.e., leaf nodes) maintain links to their corresponding neighbors. In this simple two dimensional tree, leaf nodes R_3, R_4, R_5, R_6,and R_7 need to maintain 2, 3, 4, 3, and 2 links, respectively.

Considering the trade-off between required I/Os and the number of links, we propose a heuristic to reduce the required number of links while providing efficient traversal of the index for a continuous query. Before going to the details, we first define the constructs of the heuristic.

Definition: (Visibility) Let MBR_{N_1} and MBR_{N_2} be the minimum bounding rectangles of two nodes N_1 and N_2 of an R-tree, respectively. For any node x, $MBR_x.i_{min}$ and $MBR_x.i_{max}$ are the lower and upper bounds, respectively, of the MBR_x for any dimension i, $1 \leq i \leq d$. Then node N_1 is visible to node N_2 if and only if there is an overlap between $[MBR_{N_1}.i_{min}, MBR_{N_1}.i_{max}]$ and $[MBR_{N_2}.i_{min}, MBR_{N_2}.i_{max}]$ for any dimension i, or there is no other set of k nodes, such that $\bigcup_{1\leq J\leq k}\{[MBR_J.i_{min}, MBR_J.i_{max}]\}$, completely obstructs the view of $[MBR_{N_1}.i_{min}, MBR_{N_1}.i_{max}]$ from $[MBR_{N_2}.i_{min}, MBR_{N_2}.i_{max}]$.

For example (Figure 15), R_3 and R_4 are visible to each other, whereas, R_3 and R_6 are not visible to each other because nodes R_4 and R_5 completely hide R_3 from R_6.

Definition: (Neighbors) Node N_1 is a neighbor of node N_2 if two nodes are at the same level of an R-tree index and N_1 is *visible* from N_2.

For example, in Figure 15, node R_5 has four neighbors R_3, R_4, R_6, and R_7.

Based on the above definitions, we propose the following heuristic.

Heuristic 1: Only nodes of one level maintain links to their *neighbors* of the same level.

In our work, we use level 1 nodes to apply this heuristic. In a practical application of the R-tree, each level 1 node covers a larger region and thus has a smaller number of neighboring nodes than a level 0 node. Furthermore, each level 1 node can directly access the data/leaf nodes that fall inside that node. Thus links at level 1 nodes can reduce the number of links as well as offer direct access to the data nodes and thus result in reduced I/O costs.

According to the above heuristic, if the tree (in Figure 16) maintains links at level 1 nodes (R_1 and R_2) instead of level 0 nodes, the number of links are reduced from 14 to 2. The reduction of links is more prominent for a higher dimensional index with a large database.

Index nodes at level 1 maintain links to their neighbors in all directions (x, y, z, w). The links in spatial dimension (x, y, z) facilitate faster traversal of the index when clients move in the space. Since the nodes at level 1 also maintain links with the neighbors in w dimension, it also offers the client to retrieve objects in multiple resolutions progressively even if the client does not move in any spatial dimension. For example, this occurs when a client's view is swept through a scene and then focused steadily at a fixed position. For the region in focus, the client needs to progressively retrieve more and more details of the objects with higher resolutions. In this case, the links in w dimension allow the system to drill down the index for retrieving wavelet coefficients of lower w values for objects with higher resolutions. Therefore, our proposed LR-tree based index structure supports the traversal of data nodes based on the overall movement characteristics (e.g., spatial or speed) of clients and thus can reduce the data retrieval costs by optimizing I/O access. In our proposed index, a node at level 1 maintains both the *id*, and the *MBR* of a neighbor as a link to that neighbor. So when a query visits a level 1 node, it can check whether the query intersects the MBRs of neighbors before accessing these neighboring nodes and thus save I/O costs.

We summarize the data retrieval process in the following algorithm.

Algorithm 3: QueryProcessingUsingLRtree

```
3.1 $rvnl \leftarrow MRVNodes()$;
3.2 if $(Intersects(Q_t, rvnl)$ then
3.3     Recursively visit intersecting neighbors;
3.4 else
3.5     Recursively visit intersecting nodes from the root node;
```

Algorithm 3 shows the steps of the query processing for the LR-tree . Let Q_t be a query from a client at time t in the server. Let $rvnl$ be the list of most recently visited nodes of level 1 of the index tree. This list contains the level 1 nodes visited by the clients at time $t-1$ or by the preceding clients at time t. This allows not only a single client but also a group of clients to utilize common pathways. Initially, at time 0, $rvnl$ is an empty list. If the query Q_t intersects with any of the nodes in $rvnl$, it recursively visits all of the neighboring nodes and their leaf nodes intersected by Q_t to retrieve the data. If there is no intersecting node in $rvnl$, then the Q_t recursively visits the intersected nodes starting from the root to the leaf nodes, and returns the results.

The proposed LR-tree allows the continuous exploration of objects using the links based on spatial adjacency, and also links the data blocks having adjacent resolutions of the same object. It is noted that, in many database applications, the top two levels of an index are generally cached to reduce the I/O costs. Though the caching of top two levels of an index reduces the cost of searching for a particular region, processing of continuous queries in a large database still requires the scanning of some portion of the disk-resident index. Therefore, in our proposed linked-index, instead of maintaining links on top levels, we prefer links at lower levels of the index. We argue that our approach of linking the R-tree nodes that are disk-resident reduces the I/Os by allowing a continuous query to sequentially scan the index for 3D objects. We see in our related experiments that the main benefits of our approach is visible when a group of clients (e.g., a tram tour) visit the same nodes sequentially following these links. In our experiments, the linked level is regularly the fourth level from the root.

For fast moving clients, our approach still retrieves all the objects in the traversed space but with a very low resolution. Thus, we only maintain links with the neighboring visible nodes of a node. However, there can be other application scenarios, where the client can jump far away between two snapshots. In those scenarios, in addition to the links with neighboring nodes, a node can maintain links with nodes at far away locations in the space. For example, a node can maintain *successor links* with nodes which are at 1, 2, 4, 8,... unit distances apart from the node (eventually covering the whole data space), and the number of nodes for a particular distance can be selected as inversely proportional to the distance of those nodes. Thus, when a client jumps from its current position, based on its travel distance, the algorithm can choose an appropriate successor link to a distant node and progressively look for objects starting from that distant node. The detailed analysis of this technique will be a topic of future study.

5 Experimental Study

We have separated the experimental evaluation into two parts. In the first part, we evaluate the system for a single client, and in the second part, we evaluate our multi-query optimization techniques in the presence of multiple clients. For the first part, we present the evaluation of client-side continuous data retrieval and buffer management techniques, and server-side motion-aware index only assuming a single client in the system at a particular time. We then combine these three motion-aware techniques and compare it with a naive system to show the overall performance. For the second part, we first independently evaluate each of the proposed server-side multi-query optimization techniques: group query processing, buffer management, and the LR-tree based indexing. Then we combine these three components and compare the performance of our system with a naive system for multiple simultaneous queries. We conclude the experimental section with a discussion showing that similar to speed, other characteristics of a client such as distance of objects from the client can be utilized to further optimize the performance.

5.1 Experimental Setup

Our experiments are set up based on a realistic augmented-reality city tour. We create tours augmented with 3D objects (e.g., representing old buildings in cities) that are uniformly distributed throughout the data space. We vary the data set sizes as 20MB, 40MB, 60MB, and 80MB by placing 100, 200, 300, and 400 objects in the data space. The default data set size in our experiments is 60MB. In addition, we have collected and approximated the head movements of tourists in two different settings: (i) tram tours, (ii) pedestrian tours. When a client moves from a given starting point towards a destination, it connects to the server through wireless links to retrieve the 3D objects. We also vary the length and the width of the query window by taking 5%, 10%, 15%, and 20% of the length and the width of the total data space, where 10% is the default for our experiments. A client issues a 3D window query with depth representing its view to retrieve 3D objects from the server. In our implementation we place all 3D objects at altitude zero. Since in our scenario z dimension of 3D objects is small comparing to other two spatial dimensions x and y, the range for the third-dimension (z) of the view is set to 0.0-1.0 that covers the entire range of that dimension. All objects that fall inside the query window are retrieved. We retrieve all objects inside a given query window with the same resolution in our experiments presented in Sections 5.2 and 5.3 regardless of the depth of objects inside a query window. In the last set of experiments (Section 5.4), we also consider the distance of the objects from the client inside a query window and retrieve objects with different resolutions depending upon the objects' positions inside a given query window.

We assume that the speed of the client reveals the detail of information that the client is willing to consume; thus, in our experiments the speed is expected to be inversely proportional to the value of the wavelet coefficients retrieved. All coefficient values are normalized to the range [0.0,1.0]. When the speed is at a normalized maximum (i.e., 1.0), only the coefficients that have the highest geometric influence need to be retrieved. Since all the vertices in the lowest resolution version of an object have coefficient values 1.0, these vertices are retrieved for the clients with the highest speed. If the speed is very slow (i.e., close to 0.0), all the coefficients between 0.0-1.0 are retrieved, leading to all the objects being retrieved with the highest resolution.

5.2 Part I: Evaluation for a Single Client

For this part of experiments we evaluate our system performance from a single client point of view. We first evaluate our proposed client-side motion-aware continuous retrieval technique. Then we evaluate motion-aware buffer management that reduces the high latency of wireless links. After that we present the evaluation of our proposed motion-aware index and measure the required I/Os for retrieving data at varying speed. We finally compare the average query response time of a client that uses above mentioned motion-aware techniques with that of a naive system.

For a single client setting, we assume that there is only one client in the system at a particular instant of time. So, we take 10 different tours, and run the experiments for each of these clients' tours independently. In each tour, a client moves from a given starting point towards a destination, and retrieves 3D objects from the server through a wireless link. Parameters for a single client setting are summarized in Table 1, where default parameters are shown in bold.

Parameter	**Value**
Bandwidth of Wireless Links	256Kbps
Latency of Wireless Links	200ms
Buffer Size	16K, **32K**, 64K, 128K
Data Distribution	**Uniform**, Skewed

Table 1 Parameters and their values for a single client setting

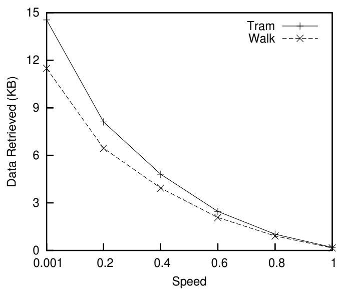


Fig. 17 Effect of speed on data retrieval

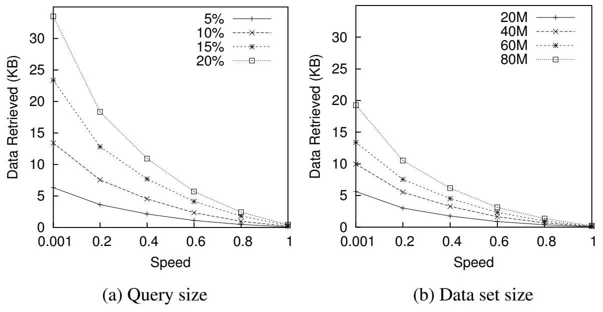


Fig. 18 Effect of query size and data set size on data retrieval

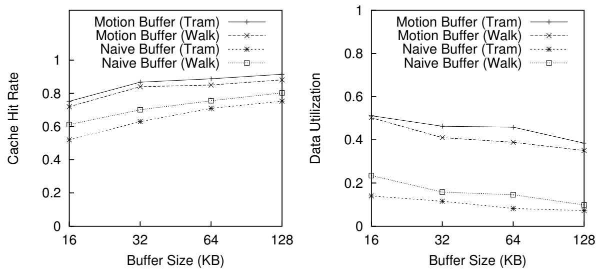
5.2.1 Evaluation of Motion-aware Continuous Retrieval

In the first set of experiments, we show the effect of the speed of the clients on continuous data retrieval. The costs of data retrieval can be reduced significantly by selecting objects with appropriate resolutions according to the speed of clients. We measure the amount of data retrieved by clients traveling similar distances at varying speeds. Figure 17 shows the average amount of data retrieved by clients on trams and on foot at different speeds (normalized to 0.001-1.0). Since the size of the objects with the highest resolution is higher than that of the objects with the lowest resolution, the data size required for clients moving with the highest speed should be significantly less than that of clients that move slowly. Figure 17 also validates this.

In Figure 18(a), we measure the average amount of data retrieved for tram tours by varying the length and the width of the query size 5%, 10%, 15%, and 20% of the length and the width of the total data space. In Figure 18(b), we vary the data set size as 20MB, 40MB, 60MB, and 80MB and measure the average amount of data retrieved for tram tours at varying speeds. These figures show that the amount of retrieved data decreases significantly with the increase of speed for different query and data set sizes. We see that for large query windows and data sizes, absolute benefits of our multi-resolution technique are more pronounced.

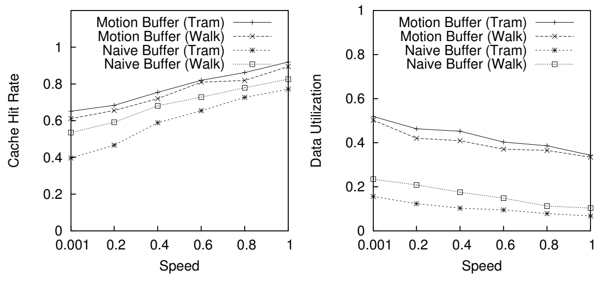
5.2.2 Evaluation of Client-side Buffer Management

In the second set of experiments, we compare our client-side motion-aware buffer management technique with a naive approach where all the surrounding regions of a query window



(a) Cache hit rate (b) Data utilization

Fig. 19 Effect of buffer size



(a) Cache hit rate (b) Data utilization

Fig. 20 Effect of speed

are buffered with equal probabilities. We compare the cache hit rate (which is a measure of reduction in latency), and the data utilization (which is a measure of data transfer overheads due to pre-fetching) of the motion-aware buffer management scheme to the naive buffer management scheme.

Effect of Buffer Size: The more buffer space a client has, the more data it can put into the buffer. However, to fill a large buffer, a client pre-fetches more data by predicting positions of the query window far into the future. Hence, there is a chance that unnecessary data may be pre-fetched with long term motion predictions. In this experiment, we vary the buffer size from 16KB to 128KB (Figure 19). The speed of the clients may also slightly vary at different parts of a tour for this experiment as the data uses a seed travel pattern collected from movements of real-world clients.

Figure 19(a) shows that the cache hit rate increases with the increase of buffer size because a larger buffer can hold more data. For a 16K buffer, the cache hit rate for tram tours and pedestrian tours are 75% and 72% for our motion-aware technique, whereas, for a 128K buffer, the cache hit rates are 92% (tram) and 88% (walk). Tram tours give superior cache hit rates because these tours can be predicted more accurately than the pedestrian tours. We also observe that the cache hit rate for the motion-aware approach is always better than that of the naive approach, i.e., on average 32% and 15% better for tram tours and pedestrian tours, respectively.

An efficient buffer management scheme should avoid pre-fetching data that will not be used by the client. From this point of view, the used portion of the total pre-fetched data is one of the major metrics for a good buffer management scheme. Figure 19(b) compares the data utilization of the motion-aware scheme and the naive scheme for both tram and pedestrian tours. The data utilization in our motion-aware scheme is 51% for trams and 50% for walking with a 16K buffer. The utilization drops to 38% (tram) and 35% (walk) for a 128K buffer. Hence, with the increase in buffer size the data utilization decreases as the client cannot make accurate predictions far into the future. The data utilization in the naive buffer management scheme is 14% (tram) and 23% (walk) for a 16K buffer, whereas, the utilization is 7% (tram) and 9% (walk) for a 128K buffer. Hence, the data utilization in our approach is always better than the naive approach (on average 4 times and 2 times for tram tours and pedestrian tours, respectively).

Effect of Varying Speed: Our buffer management scheme also uses a multi-resolution representation of objects. Figure 20 shows that, the cache hit rate increases from 64% to 91% (tram) and 61% to 89% (walk) with the increase of speed as more data can be buffered with lower resolutions. However, due to long distance predictions, we see that the data utilization is less at higher speeds than that of lower speeds. Our motion-aware approach achieves higher (on average 33% for tram tours and 10% for pedestrian tours) cache hit rates. We also have a higher data utilization (35%-51%) in comparison with that of the naive approach (7%-23%).

5.2.3 Evaluation of Motion-aware Indexing

In the third set of experiments, we evaluate our advanced indexing and access strategy on wavelet-based multi-resolution objects in comparison to the simpler index (naive approach) proposed in Section 4.3. We have implemented R^*-tree [30], which is a variant of R-tree and shows superior performance over other R-tree variants. In our implementation, we use a $3D$ (x, y, w) R^*-tree, and omit the z dimension, because in our scenario z dimension is small comparing to other two spatial dimensions x and y. The page size and the node capacity of the R^*-tree are set to 4K and 20, respectively.

Effect of Varying Speed: First, we observe the effect of speed while retrieving multi-resolution objects using our motion-aware index. Figure 21 shows that when the speed is high, i.e., in the range of 0.9-1.0, we require approximately 8-11 times less I/O costs than the costs for clients moving at the lower speeds (i.e., 0.001). This is because most of the wavelet coefficients have very small values and have almost insignificant geometric influence on the geometry of objects, i.e., leading to these coefficients not being retrieved for higher speeds. Our index structure avoids the retrieval of any extra data than required, whereas the simplistic approach requires a larger amount of retrievals. Fig-

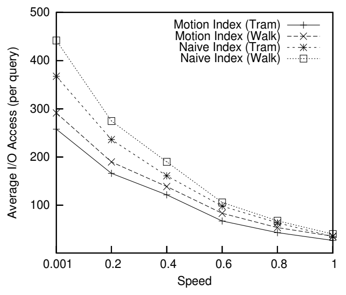
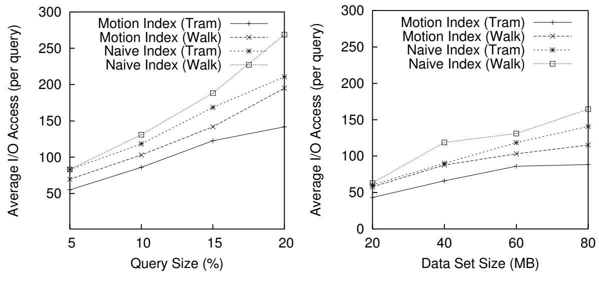
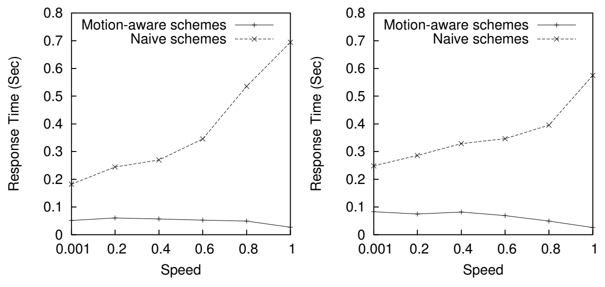


Fig. 21 Effect of speed



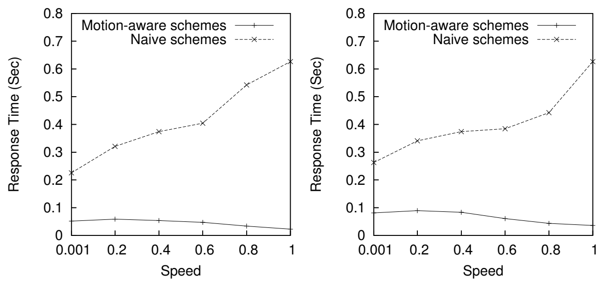
(a) Query size (b) Data set size

Fig. 22 Effect of query and data set sizes



(a) Tram (b) Walk

Fig. 23 Query response time (uniform)



(a) Tram (b) Walk

Fig. 24 Query response time (zipf)

ure 21 shows that our motion-aware access method incurs much less (21%-52%) I/O costs than the naive approach.

Effect of Query and Data Set Sizes: In this experiment, we vary the query size and keep the data set size at 60MB and the speed at 0.5 for each of the tours. Figure 22(a) shows that the data retrieval costs increase with the size of the query. Furthermore, our motion-aware access strategy incurs on average 36% less I/O costs than the naive approach. The improvement is more prominent for larger query size, which is up to 49%, because the server needs to access higher number of nodes for a larger query than that of a smaller query.

We also show the scalability of our index and access strategy by varying the data set size from 20M to 80M (Figure 22(b)). In this case the query size is fixed at 10% and the speed at 0.5. The results show that as the data size increases, the cost difference is more pronounced between our approach and the naive method. The improvement is 59% for the largest data set size of 80MB.

5.2.4 System Performance

In this set of experiments, we compare the overall performance improvement of our motion-aware approach with a naive non-multi-resolution technique from a single client's perspective.

Our motion-aware approach consists of client-side continuous data retrieval and buffer management techniques, and server-side motion-aware R^*-tree based index. To obtain a naive system, we always retrieve objects with the highest resolution and we use an R^*-tree [30] to index objects without using multiple resolutions. We also use a simple Least Recently Used (LRU) scheme for caching. In these experiments, each client travels for the same duration of time at varying speeds. Hence, when a client is traveling at a higher speed, it covers a larger area in the city than that of a slowly moving client. We measure the average response time for queries (with size 5%) from the clients moving at different speeds. Experimental results in Figure 23 and Figure 24 show that, for both uniform and Zipfian data sets, the query response time for both tram and pedestrian tours in our motion-aware approach is on average 23 times less costly than that of the naive approach at high speeds (i.e., 1.0). The improvement is on average 4 times when the speed of the client is low (i.e., 0.001). Thus these results reveal that the performance of the naive system degrades with the increase of speed, because a large number of objects need to be retrieved in a short period of time. However, the motion-aware approach can cope with the speed by retrieving lower resolution objects. The tram tours show a slightly lower response time than that of pedestrian tours because tram tours can be predicted more accurately.

5.3 Part II: Evaluation for Multiple Clients

In this second part, we evaluate our server-side multi-query optimization techniques, where the goal is to reduce I/O costs while processing simultaneous queries in the presence of multiple clients. We use the number of disk pages accessed per query as our performance metric. If we need to evaluate the overall cost in terms of time, we can charge 10ms for each page access as in [12, 13].

In this section, we first present the evaluation of motion-aware group query processing techniques that reduce I/O costs significantly. Then, we evaluate our server-side buffer management scheme that also reduces the I/O costs by pre-fetching and caching the data for the available buffer. Next, we evaluate the LR^*-tree based index (we use R^* variant of the R-tree family), and it shows significantly improved I/O access over the R^*-tree based motion-aware index. Finally, we combine all these three components and compare this combined approach with a naive approach by varying the number of clients in the system.

In the following sets of experiments, for multiple clients, we use the collected pathways of tram and pedestrian tours as described in the single client setting. Since the movement path of each tour is collected from a single client (also called the seed client), to simulate the group behavior in realistic grouped movements, we use a uniform random number generator to place a number of clients in the vicinity of the seed client (i.e., in a circular area centered at the position of the seed client with a radius of 10 units) to form a group that follows the same path of the tour. For each client in the group, we randomly vary the speed within the confined region of the group as it moves along the movement path. Though all the clients in a group use the same movement path, the direction of the movement within a group slightly varies among clients. For example, one client may look straight, whereas another client may look at one side of the path, and a third client may look at the other side of the path.

We vary the number of clients in a group to generate groups with different sizes for each tour. We vary the number of clients in a group from 5 to 25. Also, we generate a number of groups in the system by randomly placing the pathways of each tour in the data space. Since we have collected the pathways for 10 different tours in the single client setting, we have also simulated 10 different groups in the following experiments.

Parameters for the following experiments are summarized in Table 2, where the default parameters are shown in bold.

Parameter	Value
Group size	5, 10, **15**, 20, 25
Buffer Size	128K, **256K**, 384K, 512K
Pre-fetch Interval	**5s**, 10s, 15s, 20s

Table 2 Parameters and their values in multiple clients setting

5.3.1 Evaluation of Group Query Processing

In this set of experiments, we evaluate our proposed group query processing techniques: sequential group query processing (SGQP) and parallel group query processing (PGQP), and compare these techniques with sequential query processing (SQP). We expect that the average I/O access can be reduced by grouping co-located clients. Since in this experiment, our main purpose is to show the performance improvement for the grouped behavior of clients, we run the experiments for a single group in the system at a particular time. If there are multiple simultaneous groups in the system, each individual group can be identified from the overlapping behavior of clients and then we can execute each group separately using our proposed group query processing techniques.

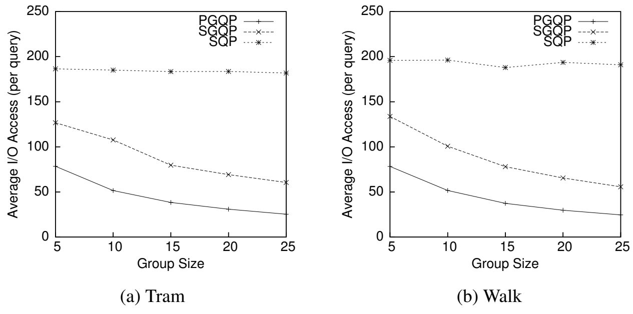
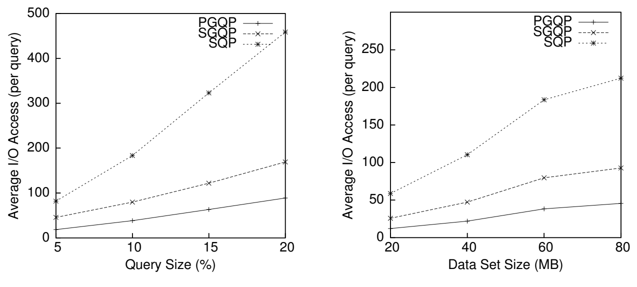


Fig. 25 Effect of varying group size

Effect of Varying Group Size: In this experiment, we vary the size of the group from 5 to 25. Figures 25(a) and (b) show that the average I/O access per query decreases with the increase of group size. Since there are more overlaps among queries for a larger group than that of a smaller group, our group processing techniques outperform SQP by a larger margin for a larger group. Experimental results show that the performance of PGQP is the best among three techniques, and SGQP performs slightly worse than PGQP because each query is executed independently for SGQP. For both tram and pedestrian tours, PGQP outperforms SQP by 5 times on average, and SGQP takes approximately on overage 2 times less I/Os than that of SQP.

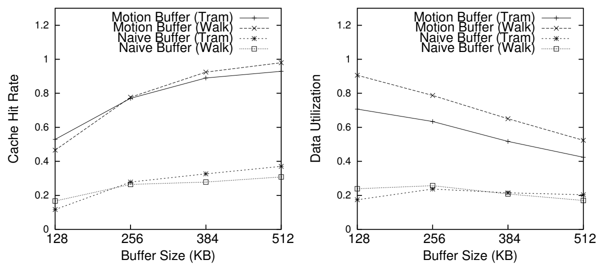
Effect of Varying Query Size: In Figure 26(a), we vary the query size from 5% to 20%, and the group size is fixed at the average value of 15. Experimental results show that the average I/O access per query increases with the increase of the query size. We also observe that both group query processing techniques (i.e., PGQP and SGQP) outperform the sequential approach with greater margins for larger query sizes as overlapping among queries increases. Figure 26(a) shows that PGQP constantly outperforms SQP by 5 times on average, and SGQP performs on average 2 times better than SQP.

Effect of Varying Data Set Size: We vary the data set size from 20MB to 80MB and the group size is again set to the average value 15. Figure 26(b) shows that the average I/O access per query increases with the increase of data set size. The results also show that PGQP performs the best,



(a) Query Size (b) Data Set Size

Fig. 26 Effect of varying (a) query size, and (b) data set size



(a) Cache hit rate (b) Data utilization

Fig. 27 Effect of buffer size

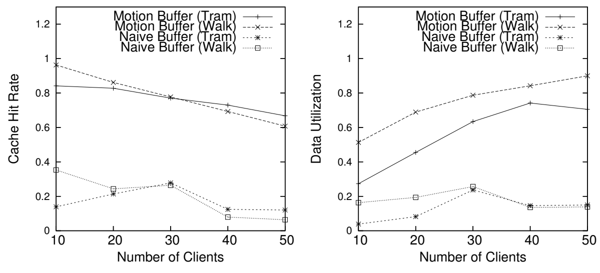
and PGQP requires on average 5 times less I/Os than that of SQP for varying data sizes. Moreover, SGQP performs approximately 2 times better than SQP.

5.3.2 Evaluation of Server-side Buffer Management

In this set of experiments, we compare our server-side motion-aware buffer management technique with a naive approach. In the naive approach, the server allocates buffer for a random set of clients, where for each of these clients surrounding regions of the corresponding query window are buffered with equal probabilities. We compare the cache hit rate (which is a measure of reduction in latency for I/Os), and the data utilization (which is a measure of data retrieval overheads due to pre-fetching) of our motion-aware buffer management scheme to the naive buffer management scheme.

We vary the buffer size, number of clients, speed, and pre-fetching cycle for different sets of experiments to show the effectiveness of our approach. Since the clients' movements are taken from different groups of tram and pedestrian tours, we select a random number of clients from each group to form a combined tour.

Effect of Buffer Size: In this experiment, we vary the buffer size from 128KB to 512KB (Figure 27). Experimental results show that the increase of buffer size increases the cache hit rate. The increased cache hit rate results in the reduction of I/O costs as well as access time because more data can be found in the server buffer. Figure 27(a) shows that, in our motion-aware technique, the cache hit rate for tram tours and pedestrian tours are 53% and 47% for a 128K buffer,



(a) Cache hit rate (b) Data utilization

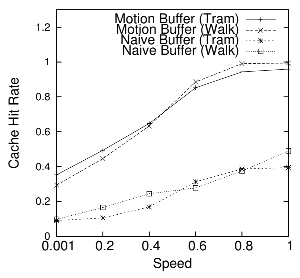
Fig. 28 Effect of number of clients

whereas, 93% (tram) and 98% (walk) for a 512K buffer. We also observe that the cache hit rate for the motion-aware approach is on average 3 times better than that of the naive approach for both tram tours and pedestrian tours. The reason is that the naive approach does not consider the grouped movements and buffers the data for clients independent of each other.

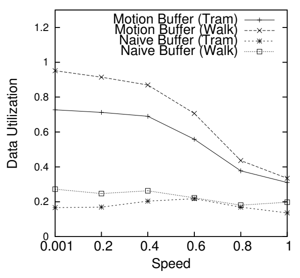
Figure 27(b) compares the data utilization of the motion-aware scheme and the naive scheme for both tram and pedestrian tours. The data utilization in our approach is always better than that of the naive approach (on average 3 times better for both tram tours and pedestrian tours). We also observe that in our approach with the increase of the buffer size the data utilization decreases as the server cannot make accurate predictions far into the future. The data utilization in our motion-aware scheme is 70% for trams and 91% for walking with a 128K buffer. The utilization drops to 42% (tram) and 52% (walk) for a 512K buffer. We have already observed that for a single client setting, the data utilization is higher for tram tours than that of pedestrian tours because the movement of a client can be predicted more precisely for tram tours. However, in this experiment for multiple clients, we observe that the data utilization in pedestrian tours is higher than that of tram tours. This is because in our setting the combined view of clients for tram tours cover a larger area and there are less overlaps among the queries than those of pedestrian tours.

Effect of Number of Clients: In Figure 28, we vary the number of clients from 10 to 50, and measure the cache hit rate and the data utilization. As the number of clients grows, there are more regions to visit by the clients at a particular instant of time. However, for a fixed size buffer the server can only buffer the region with higher probabilities. Thus the higher the number of clients, the more the clients need to fetch data directly from the disk. Therefore the cache hit rate decreases with the increase of the number of clients. On the other hand, the data utilization increases with the increase of the number of clients as there are more overlaps among clients.

Figure 28(a) shows that for our motion buffer, the cache hit rates are 84% (tram) and 96% (walk) for a small num-

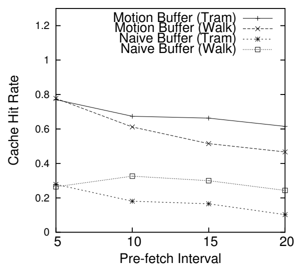


(a) Cache hit rate

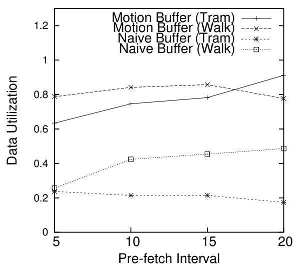


(b) Data utilization

Fig. 29 Effect of varying speed



(a) Cache hit rate



(b) Data utilization

Fig. 30 Effect of pre-fetching cycle

ber of clients 10, whereas the cache hit rates are approximately 67% (tram) and 61% (walk) for a large number of clients 50. Furthermore, our motion-aware buffer constantly outperforms the naive approach (on average 5 times). Figure 28(b) shows the data utilization of the buffer for varying number of clients. It shows that the data utilization increases from 27% to 74% for tram tours, whereas from 51% to 90% for pedestrian tours. Moreover, the data utilization of our motion-aware approach is on average 5 times better than that of the naive approach.

Effect of Varying Speed: In Figure 29, we measure the cache hit rate and the data utilization while varying the speed. Results show that the cache hit rate in our approach increases with the increase of speed as more data can be buffered with lower resolutions. However, due to long distance predictions, we see that the data utilization is less at higher speeds than that of lower speeds. Our motion-aware approach achieves higher (on average 3 times) cache hit rates. We also have a higher data utilization (35%-96%) in comparison with that of the naive approach (12%-27%).

Effect of Pre-fetching Cycle: In Figure 30, we vary the pre-fetching period at the server and measure the cache hit rate and the data utilization. Figure 30(a) shows that the cache hit rate decreases from 78% to 62% for tram tours and from 78% to 47% for pedestrian tours with the increase of pre-fetching cycle as data becomes stale for a large pre-fetching cycle. Our motion-aware buffer has always higher cache hit rate than the naive approach. Moreover, Figure 30(b) shows that the data utilization of our motion-aware buffer is on average 4 and 2 times better than

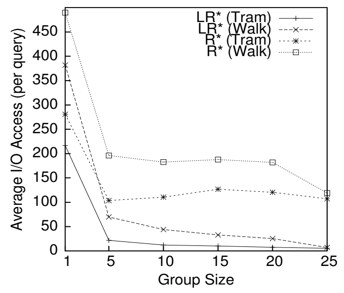


Fig. 31 Effect of varying group size

that of the naive buffer for tram and pedestrian tours, respectively.

5.3.3 Evaluation of the Linked R-tree (LR-tree)

In this set of experiments, we evaluate our proposed LR-tree and show the effect of pathways. In the following experiments we have implemented the R^* variants of the LR-tree. We vary different parameters like group size, speed, query size, data set size for evaluating the LR^*-tree based motion-aware index and compare the results with the R^*-tree based motion-index.

We have run experiments for both tram and pedestrian tours. Since in this set of experiments we want to show the effect of grouped movements of clients for efficient processing of continuous queries using the LR^*-tree, we evaluate each group of clients separately.

Effect of Group Size: We expect that the benefit of introducing the pathways in the index is more prominent for a large group of clients. This is because, if there are more clients that follow the same pathway, then each of these clients can access neighboring nodes from the index. In Figure 31, we can see that in our LR^*-tree based index, the average I/O access per query decreases with the increase of group size (or number of clients). For example, the average I/O access for a larger group of size 25 is on average 4 times (tram) and 10 times (walk) less than that of a smaller group of size 5. We also observe that the average I/O access for the R^*-tree based index is approximately 17 times (tram) and 19 times (walk) higher than that of our proposed LR^*-tree based index for a large group of size 25. The improvement is approximately 4 times for a small group of size 5. Even for a single query (i.e., group size 1), the LR^*-tree based index achieves on average 29% improvement over the R^*-tree based index, because each subsequent queries from the client can directly access the desired leaf nodes using the LR^*-tree.

Effect of Group Distributions: We observe that the size of the area covered by a group can vary while the number of clients in the group remains same. For example, a bus (e.g., a small or large) can represent the size of the covered area by the group. In this experiment we vary the window size

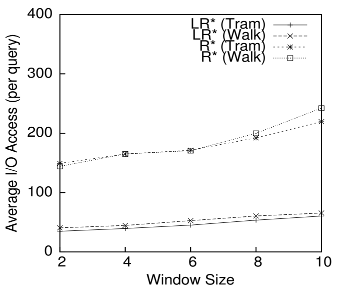


Fig. 32 Effect of varying window size

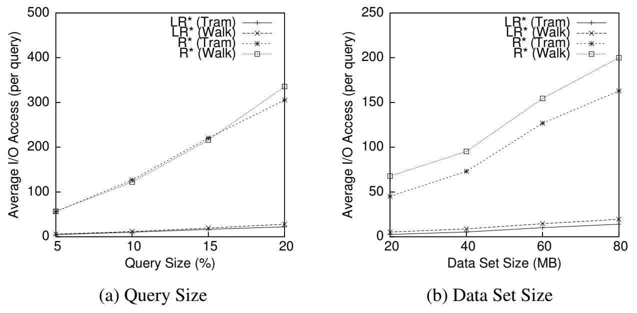


Fig. 33 Effect of varying (a) query size, and (b) data set size

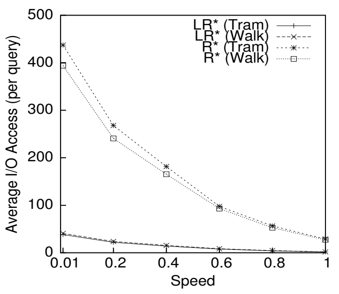


Fig. 34 Effect of varying speed

that represents the covered area of a group. Figure 32, we see that the average I/O access slightly increases for a large window size, because more nodes need to be accessed for sparsely distributed clients. We observe that the average I/O access for the LR^*-tree based index is always 3-4 times less than that of the R^*-tree based index.

Effect of Varying Query Size: We also vary the query size from 5% to 20%. Figure 33(a) shows that the average I/O access per query increases with the increase of query size. From the figure, we can see that the LR^*-tree based index always performs on average 13 times (tram) and 11 times (walk) better than the R^*-tree based index in all cases for both tram and pedestrian tours.

Effect of Varying Data Set Size: We vary the data set size from 20MB to 80MB. Figure 33(b) shows that the average I/O access per query increases with the increase of data set size. The reason for this is that for a larger data set a query needs to explore more nodes in the index than that of a smaller data set. Experimental results also show that the average I/O access for the LR^*-tree based index is on average 12 times less than that of the R^*-tree based index.

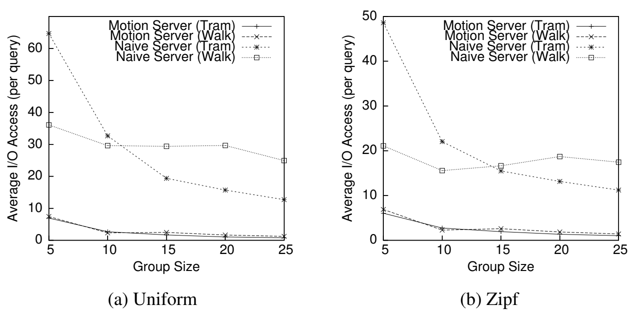


Fig. 35 Motion-aware server (a) uniform, and (b) zipf

Effect of Varying Speed: In Figure 34, we measure the I/O access while varying the speed. This experiment shows that with the increase of speed the average I/O access per query decreases as less data needs to be retrieved for lower resolutions. Figure 34 also shows that the average I/O access in the LR^*-tree based index is on average 12 times less than that of the R^*-tree based index.

5.3.4 Overall Performance

In this final set of experiments, we combine three server-side techniques for multi-query optimization, and show the overall improvement of I/O costs for processing queries with different group size for both uniform and Zipfian distributions of data (Figure 35). We compare our motion-aware server with a naive server. The motion-aware server consists of motion-aware parallel group query processing, motion-aware buffer management, and the LR^*-tree based index. On the other hand, the naive server consists of sequential query processing, LRU buffer management, and the R^*-tree based motion-aware index without pathways.

Figures 35(a) and (b) show that the motion-aware server requires on average 7 I/O accesses per query for a small group size of 5, and on average 1 I/O access per query for a large group size of 25 for both uniform and Zipfian data sets. Figure 35(a) shows that for uniform data sets the motion-aware server requires 9 (for a small group size of 5) to 16 (for a large group size of 25) times less I/Os than that of a naive server for tram tours, and 5 (for a small group size of 5) to 20 (for a large group size of 25) times less I/Os than that of a naive server for pedestrian tours. On the other hand, Figure 35(b) shows that for Zipfian data sets our approach performs 8 (for a small group size of 5) to 11 (for a large group size of 25) times better than that of the naive approach for tram tours, and 3 (for a small group size of 5) to 12 (for a large group size of 25) times better than that of the naive approach for pedestrian tours. We also observe that the average I/O accesses for tram tours in the naive server decreases with the increase of group size. This is because LRU performs reasonable well for tram tours and reduces I/O costs for a large group. However, our results reveal that the motion-aware approach can always cope with a large group size for both tram and pedestrian tours.

It is important to note that each of the above components can independently optimize the processing of continuous queries. Thus it is not mandatory to have all these components in a server at the same time.

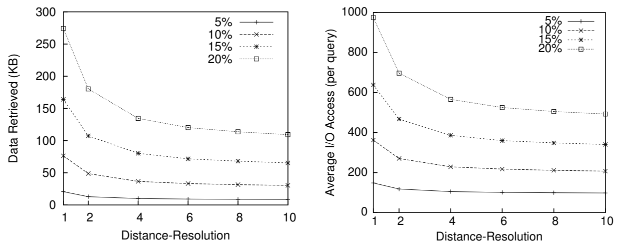
5.4 Discussion

We have shown that multi-resolution retrieval of 3D objects based on the speed of clients can significantly reduce the cost of data retrieval. Similarly, in many applications (e.g., digital battleground), other characteristics of clients such as the distance of objects from clients can also be utilized to further improve the performance. For example, a client can retrieve a closer object with a higher resolution, whereas a lower resolution might be sufficient for a distant object. We present experimental evidence that shows this effect. We use a similar experimental setup as described in Section 5.1. As the client moves along the path in a tram tour, for each of the query windows it retrieves objects inside the query window in multiple resolutions based on the distance of the objects from the client.

Similar to speed, we assume that the distance of a client from a 3D object determines the detail of information of the object that the client needs to retrieve. Thus, in our case the distance is expected to be inversely proportional to the value of the wavelet coefficients retrieved. For this, a query window is divided into multiple blocks based on the level of depth from the client, and for each block, we retrieve data with an associated resolution.

To show the effect of increasing depth (or distance), in our experiments, we vary the number of blocks from 1 to 10, where 1 represents the entire query window with the same depth and 10 represents that the query window is divided into 10 equal blocks with different depth. The resolution of the objects of different blocks decreases with the increase of the distance from the client. For example, when the number of blocks is 10, for the nearest block we retrieve all coefficients between 0.0-1.0, and then for each successive block we decrease the resolution of objects, and finally for the farthest block we retrieve objects with the lowest resolution, i.e., coefficients between 0.9-1.0. Since the number of blocks of a query window determines the effect of distance on resolutions of objects, we call this parameter *distance-resolution*.

We show the effect of the distance on continuous data retrieval in Figures 36(a) and (b). We also vary the length and the width of the query window size 5%, 10%, 15%, and 20% of the length and the width of the total data space. Figure 36(a) shows the average amount of data retrieved by clients for varying distance-resolution. The figure shows that



(a) Data Retrieved (b) I/Os

Fig. 36 Effect of distance on data retrieval

the amount of retrieved data decreases with the increase of distance-resolution for different query window sizes, and the amount of data retrieved is on average 2.5 times less for the distance-resolution of 10 than that of the case when the distance is not considered (i.e., distance-resolution of 1). Figure 36(b) shows the required number of I/Os decreases with the increase of distance-resolution, and the I/O cost is on average half of when the distance-resolution is 10 comparing to that of the distance-resolution of 1.

An extension of our work that covers other parameters of clients such as the priority of objects (e.g., objects with higher priorities can be retrieved with higher resolutions than that of lower priority objects) that can be used for efficient multi-resolution retrieval, will have significant benefits for future applications.

6 Conclusion

In this paper, we have introduced a motion-aware approach for efficient processing of continuous queries on a 3D object database. We have exploited the motion of clients to optimize query processing for both single and multiple queries in a client-server model. As a part of the complete solution, we have proposed motion-aware continuous data retrieval and buffer management techniques on the client. On the server-side, we have developed a suit of motion-aware techniques that include group query processing, buffer management, and indexing for 3D objects.

Through an extensive set of experiments, we have shown that the system that adopts our motion-aware schemes observe significantly reduced query response time as well as I/O costs than that of a system that does not consider the motion of clients. We have observed that the query response time in our motion-aware approach is 23 times less than that of a naive system when the speed of the client is high, and 4 times less when the speed of the client is low. For multiple simultaneous queries, in terms of I/O costs, our motion-aware approach outperforms a naive approach by 20 times for a larger group size of 25 and by 9 times for a smaller group size of 5.

Acknowledgment

We would like to thank the anonymous reviewers for their comments that improved our paper. This work is partially supported under the Australian Research Councils' Discovery funding scheme (project number DP0880215).

References

1. LifeClipper: http://www.torpus.com/lifeclipper/ (2005)
2. Ofcom: http://www.ofcom.org.uk/static/archive/Oftel/-publications/research/2002/benchint1202_56.htm (2002)
3. Ali, M.E., Zhang, R., Tanin, E., Kulik, L.: A motion-aware approach to continuous retrieval of 3D objects. In: ICDE, pp. 843–852 (2008)
4. Gurtov, A., Floyd, S.: Modeling wireless links for transport protocols. SIGCOMM Computer Communication Review **34**(2), 85–96 (2004)
5. Walke, B.H.: Mobile Radio Networks: Networking and Protocols. John Wiley & Sons, Inc. (2001)
6. Qualcomm: http://www.qualcomm.com/common/documents/white_papers/HSPAPlus_MobileBroadband_021-309.pdf (2009)
7. Rohde: http://www2.rohde-schwarz.com/en/technologies/cellular_standards/3GPP_HSPA/information/ (2009)
8. Chou, C.T., Misra, A., Qadir, J.: Low-latency broadcast in multirate wireless mesh networks. IEEE Journal on Selected Areas in Communications **24**(11), 2081–2091 (2006)
9. Schlosser, S.W., Schindler, J., Papadomanolakis, S., Shao, M., Ailamaki, A., Faloutsos, C., Ganger, G.R.: On multidimensional data and modern disks. In: FAST, pp. 17–17 (2005)
10. Yu, H., Ma, K.L., Welling, J.: A parallel visualization pipeline for terascale earthquake simulations. In: ACM/IEEE Supercomputing, p. 49 (2004)
11. Freitas, R.F.: Storage class memory: technology, systems and applications. In: SIGMOD, pp. 985–986 (2009)
12. Hu, H., Lee, D.L.: Range nearest-neighbor query. IEEE TKDE **18**(1), 78–91 (2006)
13. Tao, Y., Papadias, D., Shen, Q.: Continuous nearest neighbor search. In: VLDB, pp. 287–298 (2002)
14. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based spatial queries. In: SIGMOD, pp. 443–454 (2003)
15. Gedik, B., Wu, K.L., Yu, P., Liu, L.: Motion adaptive indexing for moving continual queries over moving objects. In: CIKM, pp. 427–436 (2004)
16. Lazaridis, I., Porkaew, K., Mehrotra, S.: Dynamic queries over mobile objects. In: EDBT, pp. 269–286 (2002)
17. Mokbel, M.F., Xiong, X., Aref, W.G.: SINA: scalable incremental processing of continuous queries in spatio-temporal databases. In: SIGMOD, pp. 623–634 (2004)
18. Nutanong, S., Zhang, R., Tanin, E., Kulik, L.: The V*-diagram: a query-dependent approach to moving knn queries. VLDB **1**(1), 1095–1106 (2008)
19. Tao, Y., Papadias, D.: Time-parameterized queries in spatio-temporal databases. In: In SIGMOD, pp. 334–345 (2002)
20. Prabhakar, S., Xia, Y., Kalashnikov, D.V., Aref, W.G., Hambrusch, S.E.: Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. IEEE Transactions on Computers **51**(10), 1124–1140 (2002)
21. Cho, G.: Using predictive prefetching to improve location awareness of mobile information service. In: ICCS, pp. 1128–1136 (2002)
22. de Nitto Person, V., Grassi, V., Morlupi, A.: Modeling and evaluation of pre-fetching policies for context-aware information services. In: MobiCom, pp. 55–65 (1998)
23. Welch, G., Bishop, G.: An introduction to the Kalman filter. SIGGRAPH 2001 Course (2001)
24. Luebke, D.P.: Level of Detail for 3D Graphics: Application and Theory. Morgan Kaufmann, CA (2003)
25. Hoppe, H.: Progressive meshes. In: SIGGRAPH, pp. 30–99 (1996)
26. Stollnitz, E.J., DeRose, T.D., Salesin, D.H.: Wavelets for Computer Graphics: Theory and Applications. Morgan Kaufmann, CA (1996)
27. Moran, F., Garcia, N.: Comparison of wavelet-based three-dimensional model coding techniques. IEEE Transactions on Circuits and Systems for Video Technology **14**(7), 937–949 (2004)
28. Patrick, G., Olivier, A., Christian, B.: Real-time reconstruction of wavelet-encoded meshes for view-dependent transmission and visualization. IEEE Transactions on Circuits and Systems for Video Technology **14**(7), 1009–1020 (2004)
29. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD, pp. 47–57 (1984)
30. Beckmann, N., Kriegel, H., Schneider, R., Seeger, B.: The R*-Tree: an efficient and robust access method for points and rectangles. In: SIGMOD, pp. 322–331 (1990)
31. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
32. Samet, H.: The Design and Analysis of Spatial Data Structures. Addison-Wesley, MA (1990)
33. Kofler, M., Gervautz, M., Gruber, M.: R-trees for organizing and visualizing 3D GIS database. Journal of Visualization and Computer Animation **11**(3), 129–143

(2000)
34. Hoppe, H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. In: IEEE Visualization, pp. 35–42 (1998)
35. Shou, L., Huang, Z., Tan, K.L.: HDoV-tree: The structure, the storage, the speed. In: ICDE, pp. 557–568 (2003)
36. Xu, K., Zhou, X., Lin, X.: Database support for multiresolution terrain visualization. In: ADC, pp. 153–160 (2003)
37. Xu, K., Zhou, X., Lin, X.: Direct mesh: a multiresolution approach to terrain visualization. In: ICDE, pp. 766–772 (2004)
38. Havran, V., Bittner, J., Sára, J.: Ray tracing with rope trees. In: Spring Conference on Computer Graphics, pp. 130–140 (1998)
39. Ousterhout, J.K.: Corner stitching: a data structuring technique for VLSI layout tools. Tech. rep., EECS Department, University of California, Berkeley (1982)
40. Yi, B.K., Sidiropoulos, N., Johnson, T., Jagadish, H.V., Faloutsos, C., Biliris, A.: Online data mining for co-evolving time sequences. Tech. rep., CS Department, Carnegie Mellon University (1999)
41. Tao, Y., Faloutsos, C., Papadias, D., Liu, B.: Prediction and indexing of moving objects with unknown motion patterns. In: SIGMOD, pp. 611–622 (2004)