

Efficient Update and Retrieval of Objects in a Multiresolution Geospatial Database

Sham Prasher Xiaofang Zhou

School of Information Technology and Electrical Engineering
The University of Queensland, Australia
{sham, zxf}@itee.uq.edu.au

Abstract

Many emerging applications benefit from the extraction of geospatial data specified at different resolutions for viewing purposes. Data must also be topologically accurate and up-to-date as it often represents real-world changing phenomena. Current multiresolution schemes use complex opaque data types, which limit the capacity for in-database object manipulation. By using z-values and B⁺ trees to support multiresolution retrieval, objects are fragmented in such a way that updates to objects or object parts are executed using standard SQL statements as opposed to procedural functions. Our approach is compared to a current model, using complex data types indexed under a 3D R-tree, and shows better performance for retrieval over realistic window sizes and data loads. Updates with the R-tree are slower and preclude the feasibility of its use in time-critical applications whereas, predictably, projecting the issue to a 1-dimensional index allows constant updates using z-values to be implemented more efficiently.

1 Introduction

Newer applications in large data repositories such as data warehousing, data mining, and multimedia and spatial data add complexity to traditional relational data management. These applications typically entail large complex data types that require non-standard mechanisms be built into the database to satisfy organization and querying needs. Recent trends in real-time computing have also lent to the addition of a temporal dimension to existent database information. In particular, the combination of this dimension to spatial data results in a model beneficial to many applications in geographic surveying, planning and resource management. This integration also allows trends over time to be discovered through data mining techniques.

Geospatial data organized into online geographic information systems (GIS) function as repositories to remote users. With the advent of modern mobile computing technologies a large market for GIS use has

been developing. Weather and traffic information can be retrieved to serve mobile navigation, as can other spatial layers relating to the layout of streets and buildings. For many cases, spatial data (objects) reflect real-world phenomena that can often change over time. For data acquisition mobile surveying and remote sensing is a common medium through which data is initially obtained and subsequently updated to GISs.

Spatial objects can consist of hundreds to thousands of points. However the limited display real-estate offered by many portable computing devices renders much of this information retrieved unusable. Correspondingly this holds for display of detailed information on high-resolution devices but from a largely zoomed-out perspective. As a result it is desirable to provide representations of spatial objects at several abstracted levels of detail, or representations at different scales. Thus a more limited display resolution could resolve to use a low-resolution object whereas a desktop workstation could handle the object at full detail. Such differentiation spares network transmission costs and storage overhead on the client-side that would otherwise be wasted. We refer to a database that provides access to multiple representations of spatial data as multi-resolution.

Mobile devices can also be used in time-critical situations such as the monitoring of oil spills, fires, and hazardous weather effects. In these scenarios spatial information changes constantly, demanding efficient update to the underlying database so that information retrieved by data users is current and accurate. This dynamism precludes the use of static pre-generated object resolutions in a multi-resolution database because of replication overhead. Traditional opaque spatial data types adopt a ‘whole-of-the-geometry’ approach when storing complex spatial objects, thus limiting the capacity of in-database modification and forcing updates to be modeled as complete re-computation of existing objects. Current multi-resolution schemes still suffer from this problem to a degree because they still work on these traditional constructs.

A multi-resolution database should allow different representations of an object to be derived from a single stored instance efficiently using some simplification

method. To accommodate changes in objects' location or its shape, the design must also provide access to specific parts of objects to minimize the impact of change to the database. Recent focus has also been placed on preserving topological relationships between objects during the course of data manipulation. Current multi-resolution architectures use data types that are not structured to cope with constantly changing geometries due to large maintenance costs on data and index structures. Moreover they rely on simplification methods that cannot guarantee the topological correctness of multiple objects with each other, or modified or added objects with others in the database. In this paper we outline a multi-resolution scheme that can both extract multiple resolutions dynamically (as part of spatial query processing process within the database [14]), and deal with constant updates efficiently with topological relationships among all objects guaranteed. This approach is based on a simplification method that implicitly preserves topology by considering an object's visual characteristics on a digital display of any particular resolutions.

The remainder of this paper is organized as follows. In Section 2 we provide the background of our work. Sections 3 to 5 outline the two architectures we use for comparison. An experimental analysis on retrieval and updates performance is presented in section 6. We conclude this paper in section 7.

2 Background

The majority of previous work on modeling the change in spatial data has looked largely at storing snapshots of objects as they change over time. Other work based on the same principle, applies changes in shape as well as location [12]. However the stigma of opaque data types remains a major obstacle in the efficient modification of existing objects, which limits modeling changes to the storage of successive, independent snapshots. Various snapshots of the same object can then be uniquely identified by a timestamp. This value indicates when the snapshot is a valid representation of the object. During a change to an object, either a new snapshot can overwrite the older instance entirely, or it can be stored in addition to previous snapshots. If the latter is done a time-series is formed, allowing further analyses into the pattern of objects as they change over time. An overview of types of queries used on fixed-shape moving objects in spatiotemporal databases can be found in [8]. More specific work has focused on storing the time quality of moving spatial objects and developing suitable language to query data on spatiotemporal relationships [13]. More specialized research has extended this idea to incorporate information on object velocity, speed and direction over time using differential mathematics [11].

Original work in multiresolution data focused on the visualization of 3D triangular meshes representing terrain

and object models [10, 9]. More recent work on geographic and topographic data has brought the concept of using space-filling curves for multiresolution modeling to the field of digital cartography [5, 3]. Typically a spatial object's geometry is represented as a sequence of connected and ordered points. Deriving multiple representations of spatial objects is achieved by employing simplification algorithms, which by the selective elimination of individual points produce visually similar yet simplistic copies of the original. Each representation has a level of detail (LoD), which corresponds to the visualization of the object on a map of a particular scale. High LoDs are best suited for large-scaled maps (high zoom factor) and visa versa for low detail representations.

Determining a suitable LoD for data depends on the extent of the data space and that of the digital display resolution. In an online context GISs are designed to deliver complete maps to users on request. From a brute-force perspective, spatial data may be extracted in its entirety, filtered and iteratively reduced on the client-end until a desirable result is achieved. However, in an online context the limited computational capabilities and resources of mobile client-side computing make this approach infeasible. Furthermore iterative, manual refinement is time-consuming and, hence, impractical in time-critical applications. Ideally the majority of computation that contributes to the generation of a map should complete either prior to or during retrieval to minimize total I/O and data transfer costs.

Recent trends in multiresolution databases focus on retrieving simplified data directly from the database during query processing. The somewhat naive approach involves storing multiple static LoD simplifications in the database [4]. This multi-representation approach allows efficient processing of scale-oriented data requests however suffers from higher storage requirements and maintenance issues. Second, more detailed representations contain redundant data from less detailed ones. In the context of online browsing, progressive retrieval of data through pan or zoom operations results in high redundancy. These problems are addressed in [6] by assigning scale values to spatial objects, and indexing on those values, allowing efficient filtering on irrelevant data during query processing. This multi-resolution approach is extended to use simplification in a pre-computation step to fragment objects into various LoD-based representations that are then assigned scale values [15]. Hence only a single representation is stored, but is structured so that fragments can be selected and recombined as needed. Processing costs to the client are also largely reduced because only extracted fragments need to be reconstructed and displayed.

A problem when dealing with changing data is the costs of re-computing scale-specific information each time a change is effected. Due to the way in which different

representations are structured, changing one part can, in the worst case, propagate changes to all other parts of the same object. We also present a hybrid adaptation to overcome this drawback by altering the method for pre-computing object representations. This hybrid is then compared with our own model which builds on work in [14] where objects are stored as z-values and subsequent simplification is performed using standard SQL queries.

3 VER: Incremental Resolutions

In this section we overview a multiresolution model built from techniques presented in [15].

3.1 Object simplification

Multiple resolutions of one object are derived using the Douglas-Peucker simplification algorithm [2]. Initially a baseline connecting the first and last points is established. The distance of a computed line joining each point to the baseline perpendicularly is measured as the selection factor. Point a distance beyond a specified threshold are retained and used to generate future baselines.

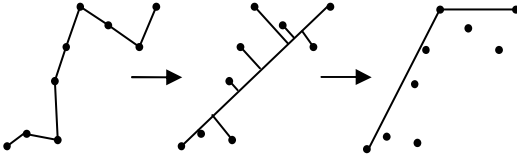


Figure 1 – Douglas-Peucker simplification: generation of a baseline and selection the first point.

An object originally represented as a sequence of points $Obj = \{p_1, p_2, p_z\}$. Each simplification may be seen as a self-contained representation of the original object, known as a resolution, which in turn can be associated with a particular scale. Thus we have a set of resolutions $Obj = \{v_1, v_2 \dots v_m\}$, each associated with an index indicating its LoD (1 being the lowest LoD, or smallest-scale representation). If the threshold used to derive a resolution v_n is given as $T(v_n)$, where $1 \leq n \leq m$, then $T(v_n) > T(v_{n+1})$. When created the final resolution will contain all points in the original. A resolution v consists of points and point indexes that denote a point's position in the original complete sequence, giving $v = \{p_i, indx_1 \dots p_k, indx_k\}$ where $indx_1 < indx_2$. Each point in the original object p_i therefore also has a range in which it is occurs given as a smaller-scale and larger-scale bound $s_{p_i} = [s_s, l_s]$. Correspondingly the scale range of a resolution v_n is given as $s_{v_n} = [s_l, s_n]$. For simplicity we restrict the scale range to a single continuous interval.

3.2 Redundancy in Resolutions

As consequence of performing several stand-alone simplifications there is an unpredictable amount of data replication between resolutions. Redundancy can run high overheads in most spatial databases where data loads are often voluminous. To remove duplication, if a point occurs in two resolutions v_a and v_b where $a < b$, then the point is removed from v_b . As a result points are only stored within the largest-scale resolution in which they occur. Following this a combination of all resolutions results in a complete reconstruction of the original object. Thus objects are incrementally reconstructed and the sorted combination of points from two resolutions v_1 and v_2 to give an object O is denoted as $O = \{v_1 \oplus v_2\}$ where $\forall p_a, p_b \mid p_a, p_b \in O \rightarrow indx_a < indx_b$. The retrieval of an object's resolution specified at any scale s_n , where $1 \leq n \leq m$, is given as $O = \{v_1 \oplus \dots \oplus v_n\}$.

3.3 Object Clipping

To allow clipping for window queries each resolution is split into a sequence of contiguous blocks. A block retains the point order and scale range of its parent resolution and records vertices alongside their index positions in the original point sequence. So a resolution $v = \{b_1, b_2, \dots, b_n\}$ where a block $b = \{p_l, indx_l \dots p_h, indx_h\}$. Given two blocks of v , b_c and b_d , $\forall p_i, p_w \mid p_i \in b_c, p_w \in b_d \rightarrow indx_i < indx_w \wedge c < d \leq n$. The length of a block, its *blocking factor*, is pre-defined by a domain user. A large *blocking factor* reduces the reconstruction costs of the object but increases the amount of extraneous data retrieval, and visa versa for smaller block lengths. A block also maintains a buffer to accommodate future modifications to the object. When the buffer is full, overflow occurs and blocks are split, distributing points between the old and new blocks equally.

3.4 Data Retrieval

Normally a minimum bounding rectangle (MBR) would be used as approximations to locate spatial objects using a spatial index. However, after fragmentation MBRs do not include information linking neighboring blocks together. As a result blocks may not be properly clipped to a query window. To solve this problem a dynamic MBR (DMBR) is computed on each block. Initially DMBRs are computed on each point of the block to contain i) the point itself, ii) the two neighboring points that have a small-scale bound equal to or less than the point and iii) all other points in between that belong to resolutions of a larger scale bound. Following this, the DMBR of a block is an MBR which encloses the DMBRs of all of its points. After fragmentation an entry for each object block in the data table is: $(object_id, resolution_id, block_id, block,$

dmb). In this schema (figure 2) the DMBR is represented as a 3D optimized rectangle. The upper and lower scale bounds attributed to each block are integrated into the rectangle's z dimension.

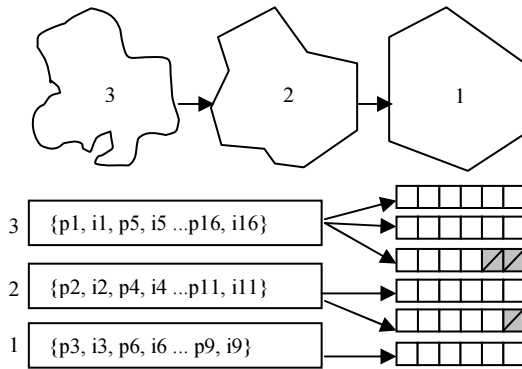


Figure 2 – Object simplified to three resolutions, and corresponding blocks (unused block space is shaded).

A 3D R-tree is imposed the *dmb* field. To retrieve objects visible in the scale range $sr = [s_a, s_b]$ and within an area of interest $window(min_x, min_y, max_x, max_y)$ a query can be formulated using the window and *sr*'s details as the x,y and z coordinates of a spatial query.

Results are ordered for efficient reconstruction of objects on the client side.

3.5 Modeling Changes to Objects

Change to an object's location denotes a uniform shift to all of its subsidiary parts. Instead of modifying the vertices of each block we maintain an offset. The offset is stored in a separate field as the difference between the object's stored location and its actual indexed location. Entries in the data tables are updated to: (*object_id*, *resolution_id*, *block_id*, *block*, *dmb*, *offset*) To keep index entries updated the coordinates of each of the object's DMBRs are changed. During reconstruction the offset is retrieved and used to adjust the object's points accordingly. Offsets and DMBRs are updated using an SQL *update* operation.

Modifications to specific points in an object, however, are difficult to perform because the Douglas-Peucker algorithm produces inflexible simplifications. Baselines generated at any given iteration of simplification are constructed from points selected in previous iterations. As a result intrinsic dependencies between selected points are formed. Therefore a single change in one resolution can cause unpredictable cascading changes in others. As a result we must recalculate all resolutions to ensure a correct set of simplifications is stored. When modifying partial geometries with this approach computational costs are synonymous to that of generating completely new

objects. Unlike the drawbacks of multi-representation storage schemes the problem arises not because the spatial object is atomic and inaccessible, but that object fragments are co-dependant in definition. The solution is to remove the element of multi-resolution object generation that causes this dependency. To this end we propose a hybrid alternative in section 5.

4 ZVAL: Z-Value Simplification

The Li-Openshaw simplification algorithm [7] considers the equivalent raster representation of given vector data during the simplification process (figure 3). The algorithm uses the concept of the Smallest Visible Object (SVO) as the finest cartographic detail visible by a human user. On a CRT or LCD display this equates to a single pixel. A list of points is first given as the polyline to be simplified. A grid of pixels is placed over the data. The pixels are defined at the resolution of the digital environment onto which the resulting simplified data is to be displayed. Pixels that contain points are retained as in the final result. This approach ensures excessive or unrealistic (self-intersecting) simplification does not occur and topological relationships are not violated.

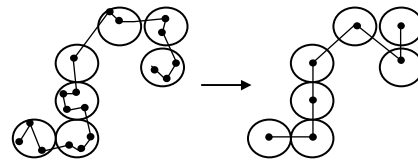


Figure 3 – Li-Openshaw simplification on data at original resolution (left) to a lesser resolution (right).

4.1 Simplification and Retrieval

An object is broken into a collection of two-line segments, each stored as an entry in a table. Points are stored as z-values. In this method object simplifications are constructed during query execution for data retrieval. The process is outlined as follows:

Step 1: establish the peano cell size for a pixel. We determine the length of a z-value necessary to represent cells (roughly) equivalent to pixels on the screen resolution. Given the area of interest in the data space and the display size in terms of pixels, the size of the area in the data space corresponding to a pixel can be determined. We call this area a *data pixel*. Because the original resolution of data is typically much higher than display resolutions the z-value representing a data pixel will be of a length, *c*, less than its original.

Step 2: choose line segments that have both endpoints in different data pixels. If a line segment's two endpoints occur in the same data pixels then its detail is lost within

that SVO. If both endpoints occur in different pixels, then the line segment is kept. The first level of decomposition where both endpoints of a line segment exist in different peano cells is called the *delta*. We can then infer that if a pixel screen resolution is at a finer level of decomposition than the *delta* of a line segment, that line spans more than 1 pixel on the display. Because it is visible at the desired display resolution the line should be retained. Additionally, since endpoints are shared between consecutive line segments, we only need to retrieve the first point of line segments, and by doing so avoid redundant data retrieval. Rows in the data table under this approach have the following structure: (*object_id*, *line_id*, *z1*, *z2*, *delta*). The *object_id* and *line_id* are used to sequence line segments. The *z1* and *z2* fields store the starting point and endpoint of a line segment. A query for simplification can thus be constructed as: “*select object_id, line_id, z1 from data_table where delta > c order by object_id, line_id*”. Results are sorted so that objects can be efficiently reconstructed on the client-side.

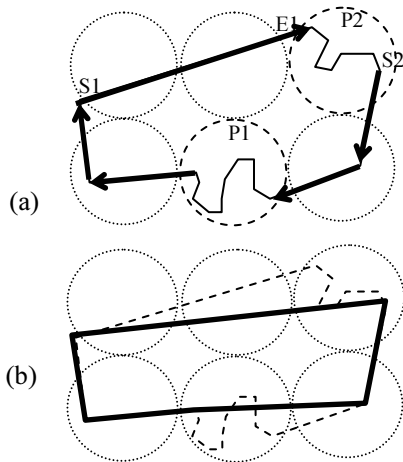


Figure 4 – Z-value simplification on a 2x3 display; (a) before and (b) after.

Figure 4 illustrates the simplification of an object using the query. Point order follows the directional arrows, the arrow-end indicating the endpoint of a line segment. Lines with *delta* > *c* are shown in bold. Lines that do not cross different pixels (contained within pixels P1 and P2) will be eliminated. Joining the starting points of retained line segments completes the final object, highlighted in figure X2B. Endpoints are discarded and are replaced by the starting points of subsequent lines that begin in the same data pixel.

Although variations are produced (i.e. joining S1 to S2 and losing E1), these are never larger than a single pixel width and hence become indistinguishable when rendered to the display resolution. Different simplifications of an object can be obtained by altering the *delta* parameter to reflect change in either the display resolution or the size of the data space. The *z1* and *delta* fields are indexed as a

composite key in a B⁺-Tree. To filter data on location for window querying we use a method adapted from [1] to translate a query window into a range query on stored *z*-values. In this conversion the window is equated to a peano cell.

4.2 Modeling Changes to Objects

To reflect a change in an object’s location every line segment is extracted, its endpoints are shifted then individually updated to the database. When modifying a point the all line segments tied to that point must also be updated. Let the point to be modified in the line segment *l_i* be *z1_i*. Given an object as a sequence of lines $O = \{l_1, l_2, \dots, l_n\}$ and $i > 1$ then we retrieve the *l_i*’s predecessor line *l_{i-1}*, or *l_n* where $i=1$. For reference this line is known as *l_p* and its starting point *z1_p*. Because lines share common endpoints $z1_i = z2_p$. After the coordinates of *z1_i* are updated, the *delta* values of *l_i* and *l_p*, *delta_i* and *delta_p*, are then recalculated on lines (*z1_i*, *z2_i*) and (*z1_p*, *z1_i*) respectively and updated to the database along with the new *z1_i* and *z2_p*. In total the operation requires two database updates. An SQL *update* operation is used to commit a point change. When new objects are created and stored the *insert* operation is used.

5 HYBR: A Hybrid Alternative

In this approach we alter aspects of VER to make the data structure more robust to change. Foremost the Li-Openshaw simplification method is used instead of the Douglas-Peucker to generate resolutions. Instead of threshold distances to generated baselines, the size of *delta* values between line segments is used to fragment objects. The *delta* of a resolution *v_n*, given as $D(v_n)$ where $1 \leq n \leq m$, implies $D(v_n) > D(v_{n+1})$. Blocks are derived as in VER. Like ZVAL, changing a point will require a recalculation of the *delta* with the point’s immediate neighbors. If the change in *delta* is large enough then the point must be pushed to a different resolution. Consequently we must know which block in to insert the point into. Since point order is ascending but not necessarily consecutive in each resolution this requires linear scanning. To reduce search time an additional field is stored for each block containing the maximum point index of that block, called the *block_max*. Rows in the hybrid schema are given as: (*object_id*, *resolution_id*, *block_id*, *block*, *dmb*, *block_max*, *offset*).

Once retrieved, a linear scan is still needed to determine the exact position within the block for insertion. Adding a new point may incur overflow in the block, in which case the next free space in a succeeding block is used. If overflow carries beyond the final point in the last block then a new block must be inserted and the Block IDs of all subsequent blocks incremented.

A change in block structure also corresponds to a change in its DMBR. By definition a DMBR can enclose points having a smaller scale bound, but scanning other resolutions for points to reconstruct DMBRs after a block change can be costly. However following the definition of object reconstruction in section 3.1.2 we can infer that neighboring points of a lower scale bound within the area of interest will be retrieved along with blocks from lower-scale resolutions. More formally let the resolution in which a point is being modified be v_n and its block b_n . The endpoint of b_n , p_{end} , is connected to another endpoint, p_{nbr} , of b_n 's neighbor b_m . Assume $m > n$. Therefore $s_{p_{end}} = [s_l, s_n]$ and $s_{p_{nbr}} = [s_l, s_n]$. So $\exists p_i \in v_a$ such that $s_{v_a} = [s_l, s_a] \wedge indx_{end} < indx_a < indx_{nbr} \rightarrow p_i \in O$ where $O = \{v_l \oplus \dots \oplus v_n\}$. This property allows us to reconstruct DMBRs using points from blocks of the same resolution. Changes to each block are committed to the database using the SQL *update* operation. For both VER and HYBR the creation and storage of completely new objects is executed using SQL *insert*.

6 Experiments

Tests were conducted to contrast the performance of VER/HYBR against ZVAL for data retrieval and updates. The latter is broken down into updating the location of objects, or their shape by direct manipulation of individual points. Additionally we test the cost of modifying objects by inserting the changed geometry as an entirely new object with an updated timestamp. The purpose of tests is to determine whether VER/HYBR and ZVAL perform consistently, and efficiently, using traditional methods for modeling object change as well as under newer methods based on direct manipulation that exploits the multiresolution data structures. Note results reflect access to largely fragmented data on disk, and hence involve generally high seek times.

6.1 Test environment

Tests were conducted on a PIII Intel system with a clock speed of 800MHz. Data was stored using Oracle's Object-Relational database with the Spatial Data Cartridge. Blocks are stored as number arrays. The DMBRs are stored as 3D optimized rectangles and indexed in a 3D R-tree. In HYBR the *block_max* field is indexed with a B⁺tree. Under ZVAL the *z1*, *z2*, *delta* and *object_id* fields are organized in B⁺-Tree indexes.

6.2 Datasets

We use a real Queensland land parcel dataset for the simulation of retrieval and update operations on large polygon objects. The data space extends 2054km×1507km. On average objects cover an area

equivalent to 93.8km², the smallest having being 7.1km² and the largest 385.4km². Data retrieval and reconstruction was performed on a sample of 102,000 points in 1000 objects. For data updates separate samples of 100 polygons containing 13,000-15,000 points were used for the object shift and point change tests. Table 1 summarizes dataset statistics.

Number of points	Retrieval + Reconstruction	Change in Location	Change in Shape
Average	147.5706	136.84	139.41
Std. Dev.	293.7027	212.8249	210.8243
Minimum	26	6	7
Maximum	5562	1114	1114

Table 1 – Data Statistics.

6.3 Retrieval and Reconstruction

Figure 5 shows a comparison of data retrieval by map size as a relative area of the total data space. This test employs both scale and location filtering on data: a range query on the composite key under ZVAL and a 3D range query under VER. Time performance shown include costs to reconstruct object on the client side after retrieval. Objects are scale-coded so that for the VER scheme at larger windows one small-scale resolution of each object is retrieved, while for smaller windows at most two resolutions are combined for reconstruction. The *blocking factor* was set to 12. When retrieving using ZVAL the *delta* was increased at each window increment to reflect the new size of the data pixel as the ratio between window and display size. The resolution for display was set to 1024×768.

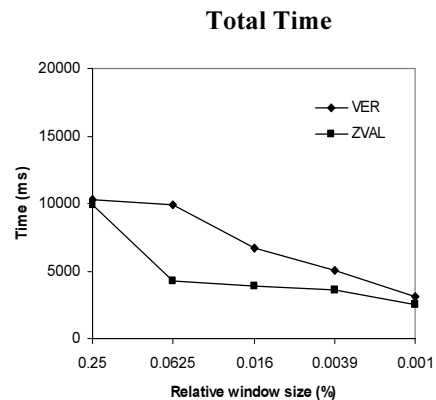


Figure 5 – Retrieval and reconstruction.

We note that data loads processed under ZVAL were larger at each window interval when compared to VER. The number of retrieved points ranged from 3.7k to 25.4k. Under VER data loads ranged from 3.1k to 23k; because

only 3 resolutions are used, and threshold values are only estimated, data distribution is not even under all scales. The results indicate that ZVAL adjusts well over a realistic set of window sizes. At 6% multiple resolutions are merged at once during reconstruction, whereas under ZVAL a single result set is processed in linear fashion. At larger (25%) zooms the 3D R-tree gains efficiency because initial filtering on the scale dimension quickly reduces potential false hits whereas ZVAL is constrained to filtering on location first. Also at 25% the data load exceeds that of the main memory sort buffer under ZVAL, adding I/O costs to reconstruction. This performance gain is lost on smaller windows where the distribution of data amongst scale and location dimensions in the index becomes more even.

6.4 Data Updates: Change in Location

Changes in objects' locations are performed identically under ZVAL and VER. Because block size affects point distribution and the regularity of overflow VER is shown under different *blocking factors* of 12, 24 and 36, denoted VER_12, VER_24, and VER_36.

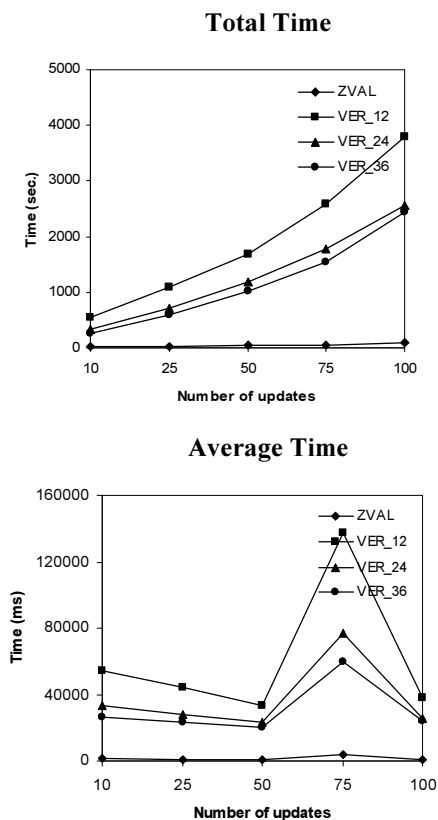


Figure 6 – Shift in objection location.

Multiple continuous changes to the 3D R-Tree amount to large reorganization costs even though ZVAL requires

more individual row changes per object. The results (figure 6) indicate a larger *blocking factor* clearly improves performance with VER by reducing the number of block updates needed, although processing times remain relatively costly. The majority of large objects (>1000 points) are located between the 50 and 100 mark, causing the spike in figure 6.

6.5 Data Updates: Change in Geometry

Second we test the time performance of changing individual random points between HYBR (*blocking factor*=12) and ZVAL. For both schemes 100 random updates to the sample are made (figure 7). We note that performance under HYBR is largely subject to block utilization, thus is shown in several different runs: the optimal case where points are not moved to other resolutions (HYBR_OPT), and using different buffer sizes where points do change resolutions (HYBR_33%, HYBR_16%). Buffer size is given as a percentage of the *blocking factor*. Figure 7 shows time to change point locations under both schemes.

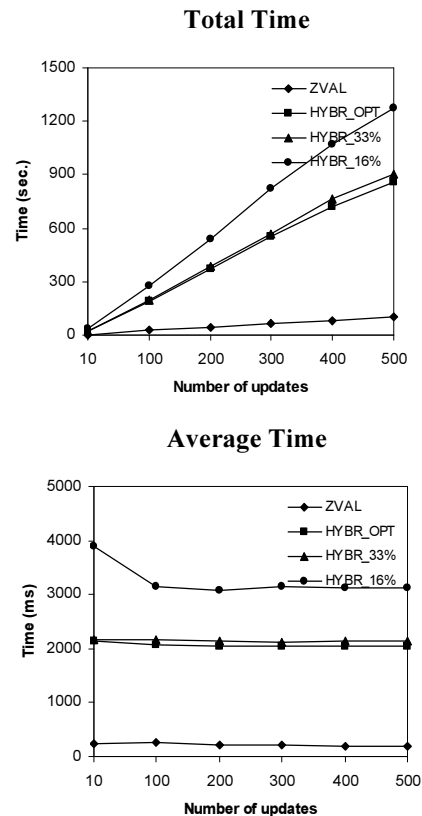


Figure 7 – Change to object geometry.

Predictably a larger buffer reduces the number of block splits and overall time. The difference between HYBR runs also indicates that splitting overhead is quite

low. However general performance in HYBR indicates inefficiency when dealing with highly dynamic data. This difference is attributable to the overall complexity of HYBR's architecture, which must restructure blocks, recompute DMBRs and reflect the updates in the 3D index.

6.6 Object Insertion

Results include the time required to generate and store multiresolution objects from a single sequence of vertices. In total 100 objects, from the second sample outlined in table 1 are inserted. Objects were not inserted in any order. For all tested runs data tables were modified to accommodate timestamp values.

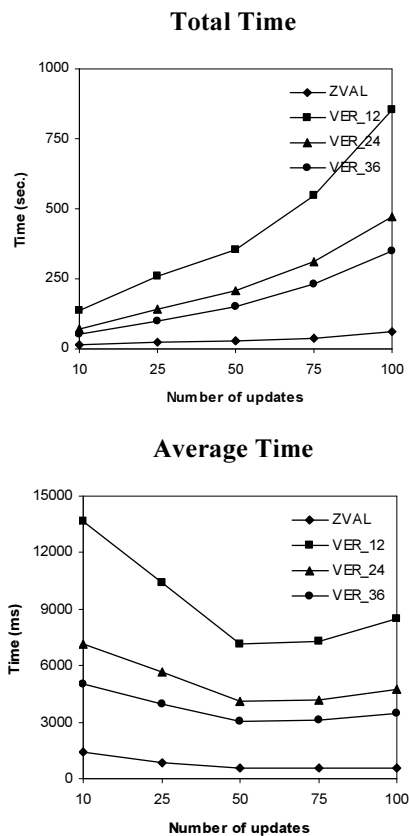


Fig 8 – Creation of new objects.

Results shown in figure 8 are consistent with previous tests carried out. The VER scheme is unable to handle large modifications. A comparison of times for individual point updates and whole object insertions suggest the latter is preferable when an object's geometry is undergoing non-trivial changes. Expectedly ZVAL maintains performance comparable to that of the previous test since the same number of atomic updates is required in both. Cumulative evidence from all test suggest ZVAL is robust and scalable to large cases of change to the data.

7 Conclusions

We presented a multiresolution spatial data model ZVAL that builds on existing primitive data types and relational indexes for fast object update in light of changes to both location and shape. Data structure and SQL syntax for update and retrieval are notably simpler than the alternate VER/HYBR model. Tests were conducted to retrieve objects with respect to location and scale, update shifts in object location, and execute changes to specific parts of objects. We compared our approach to VER, which indexes objects in a 3D R-tree. Performance from retrieval and reconstruction show that ZVAL was faster than VER. Though, with larger data loads ($\approx 30k+$) VER is markedly superior because of the number of rows to sort in ZVAL compounds. By the same token such loads typically contain too much detail to display on most common resolutions. Overall performance indicates the z-value-based model can adequately deal with numerous changes to data in succession. Conversely while VER performs well for data retrieval, it suffers from maintenance on object fragments and the R-tree index during updates.

An added advantage of ZVAL is that the range of representations derivable from a single stored instance correlated directly to the range of *delta* values. Under VER this range is limited to the number of pre-generated resolutions. Hence one resolution may be used for a large range of display resolutions whereas ZVAL will generally be able to produce simplifications tailored specifically to more resolutions. Furthermore our approach permits the efficient access and modification to specific object parts without resorting to complete re-computation. This is an advantage that previous multiresolution models have been unable to exploit, limiting their capacity to deal with malleable object types in the database.

References

- [1] A. Aboulmage, W. G. Aref (2001) *Window Query Processing in Linear Quadtrees*. Distributed and Parallel Databases 10(2).
- [2] Douglas, D. H. and T. K. Peucker (1973). *Algorithms for the Reduction of the Number of points Required to Represent a Digitized Line or its Caricature*. The Canadian Cartographer vol 10(no. 2).
- [3] Dutton, G. (1999). *A hierarchical coordinate system for geoprocessing and cartography*. Lecture Notes in Earth Sciences, Vol 79.
- [4] A. U. Frank and S Timpf (1994) *Multiple representations for cartographical objects in a multi-scale tree – an intelligent graphical zoom*, Computers and Graphics, 18(6), 1994, pp. 823-929.

- [5] Gerstner, T. (2001). *Multiresolution visualization and compression of global topographic data*. Geoinformatica.
- [6] M. F. Horhammer and M. Freeston (1999). *Spatial indexing with a scale dimension*. SSD'99.
- [7] Z. Li & S. Openshaw (1992) *Algorithms for automated line generalization based on a natural principle of objective generalization*. International J. of GIS, 6(5), 1992, pp. 373-389.
- [8] K. Porkaew, I. Lazaridis, and S. Mehrotra (2001) *Querying Mobile Objects in Spatio-temporal Databases*. SSTD'01.
- [9] E. Puppo (1998) *Variable Resolution Triangles*. Computational Geometry, Vol.11, N.3-4.
- [10] E. Puppo (1997) *Building and traversing a surface at variable resolution*. In Proceedings IEEE Visualization 97.
- [11] J. Su, H. Xu and O.H. Ibarra (2001) *Moving objects: Logical Relationships and Queries*. SSTD'01.
- [12] E. Tøssebro and R.H. Gutting (2001) *Creating Representations for Continuously Moving Regions From Observations*. SSTD'01.
- [13] M. Vazirgiannis and O. Wolfson (2001) *A Spatiotemporal Model and Language for Moving Objects on Road Networks*. SSTD'01.
- [14] X. Zhou, S. Prasher, and M. Kitsuregawa (2002) *Database Support for Spatial Generalisation for WWW and Mobile Applications*. WISE'02.
- [15] S. Zhou and C. B. Jones (2001). *Design and Implementation of Multi-Scale Databases*. SSTD'01.