Design and Implementation of Multi-scale Databases[1]

Sheng Zhou and Christopher B. Jones

Department of Computer Science
Cardiff University
Cardiff CF24 3XF, United Kingdom
{s.zhou,c.b.jones}@cs.cf.ac.uk

Abstract. The need to access spatial data at multiple levels of detail is a fundamental requirement of many applications of geographical information, yet conventional spatial database access methods are based on single resolution spatial objects. In this paper we present a design for multi-scale spatial objects in which both spatial objects and the vertices of their component geometry are labelled with scale priority values. Alternative approaches to database implementation are considered in which vertices are organised into scale-bounded layers. Access times for spatially-indexed vertex block schemes (comparable to the PR-File) were superior to a BLOB scheme where only entire multi-scale objects were spatially indexed. The use of a 3D R-tree to integrate scale and space indexing was found to improve considerably on using either R-Tree indexing of space only or B-tree indexing of scale. Techniques are also presented for client-side reconstruction of cached multi-scale geometry. Implementations are compared in a client-server environment using the Informix object relational database system.

1 Introduction

A characteristic of spatial objects in most geographical and geoscientific databases is that they represent a view of reality at some particular level of semantic and geometric abstraction. Many applications of geographical data however require access at several levels of abstraction, for purposes of information browsing as well as scale-specific analyses. These needs are met typically by storing distinct representations that refer to the same real world phenomena. Typically these representations are simply digital versions of the scale-specific map series produced by topographic mapping agencies. The approach suffers from inflexibility of scale, imposed by the source data, in combination with data duplication and with problems of integrity maintenance of the multiple versions.

In theory it is possible to envisage storing in a spatial database a single detailed representation of the phenomena of interest and then applying map generalisation algorithms online to retrieve the scale-specific representation that suits the user's interests. For applications requiring large spatial databases that may be applied across

[1] This research was funded by EPSRC Grant GR/L49314 in collaboration with the Ordnance Survey and ESRI(UK).

C.S. Jensen et al. (Eds.): SSTD 2001, LNCS 2121, pp. 365–384, 2001.
© Springer-Verlag Berlin Heidelberg 2001

a wide range of levels of detail, there appears to be little immediate prospect of such a scenario being realised. The reasons relate to the computational overheads of processing potentially very large volumes of data in order to retrieve a small subset, as well as the functional and performance limitations of existing map generalisation procedures.

A pragmatic solution is one of pre-computation that strikes a balance between the use of digitised manually generalised map data and the exploitation of existing map generalisation procedures where they are appropriate. The most widely investigated area of map generalisation is that of simplification of linear features. These procedures can be used to attach scale-related priority values to the vertices of linear features. If these values are stored, it is possible to retrieve subsets of the vertex geometry that can be re-assembled into simplified versions of the line. This principle underlies several multiresolution data structures and database storage schemes. Examples are the strip tree data structure [1] and its variants such as the arc-tree [2], the BLG-tree (employed in the reactive data structures of [3]), the binary tree structure of [4] and layered schemes such as the Multi-Scale Line Tree [5], the Multi-Scale Implicit Triangulated Irregular Network [6], which both used quadtree indexing, and the PR-File [7] which used a type of R-tree for spatial indexing.

An issue not addressed in the above schemes is that of topological consistency of the simplified spatial objects. All of these schemes employed point selection techniques that can result in simplified lines that overlap themselves and neighbouring features. An approach that maintains topological integrity of polygonal maps at multiple scales was presented in [8] while [9] have described a scheme to maintain topological integrity between complete spatial objects of different types. An important related development is the design of line generalisation procedures that are inherently topology-preserving, in that all realisations of the line that can be generated by the procedure are guaranteed topologically consistent. Examples of such procedures are those of [10, 11 and 12], the latter being employed in the progressive transmission scheme of [13]. The presence of topologically consistent generalisation techniques introduces the possibility of database storage of multi-scale spatial objects that have explicit degrees of consistency with themselves and with other objects in the database [14]. Pre-computation of topological consistency, for limited ensembles of stored objects, may then be combined with online generalisation procedures for purposes of i) graphical conflict resolution due to arbitrary selection of map symbols [15, 16] and ii) topologically-consistent generalisation of sub-sets of objects that do not have pre-computed consistency.

In this paper we focus on several multi-scale spatial database implementation issues, in particular: scale-range coding of vertices; clipping-support for multi-scale geometry; strategies for client-side reconstruction of cached multi-scale geometry; and the relationships between indexing on scale and indexing on geometric space. It should be noted that the paper deals primarily with issues concerning access to the geometry of individual multi-scale spatial objects, primarily lines and polygons. It is assumed that these objects store data that covers a specified range of scale. The geometry of an object may have been derived from larger scale objects in the course of map generalisation operations, such as amalgamation and dimensional collapse, and it may itself be a source of more generalised spatial objects through similar trans-

formations. The scale values attached to vertices are assumed to be scalars that are functionally related to map scale.

The remainder of the paper is set out as follows. In Section 2 we show how scale can be associated with spatial data. Section 3 introduces scale-coding methods that can be used to partition geometry into scale-specific intervals. In Section 4 we present alternative storage schemes for maintaining the geometry of multi-scale geometry objects. The issue of clipping multi-scale linear and polygonal features is addressed in Section 5 as a precursor to the definition in Section 6 of client-side functions for panning and zooming with multi-scale geometry. Two methods for object-reconstruction from cached geometry are presented. Section 7 introduces alternative approaches to indexing on scale. In Section 8, experimental results of spatially indexing different sized blocks of vertices in an extended relational environment are compared with object-level spatial indexing in a BLOB implementation. The results of applying the two cached-object reconstruction techniques are described, as well as the results of using both B-tree and 3D R-tree indexing on scale. The paper concludes in Section 9 by summarising the relative merits of the various implemented techniques.

2 Spatial Objects and Scale

An observation of a spatial phenomenon is made at a certain time from a certain point of view reflecting the degree of abstraction, or *scale* of the observation. In the general case a spatial object may be located in 3D space and refer to a specific time span. Here we confine ourselves to 2D space and assume that all data relate to single period of time. A *spatial object* SO can therefore be defined as a function in a 3-dimension space *SV* on $\Re$, consisting of two spatial dimensions and the scale dimension. Thus $SO = f(s, x, y)$, where f is a function to map point $p(s, x, y)$ in *SV* to *SO*.

We define a *snapshot* of a spatial object at scale s as $SO_s = (oid, R_s(G_s))$, where *oid* is an unique identity of the spatial object, $G_s = (GID, \{g_i \mid i = 1, n\})$ is a *geometry set* with identity *GID* and R_s is an operator to assemble all geometry objects g in G into the spatial form of *SO* at scale s.

Each *geometry object g* consists of one or several vertices, which may be a point; an open curve (a polyline) or a closed curve (a polygon). A vertex v_s has a unique identifier *vid,* a sequence number *vsn* indicating the position of the vertex in the parent geometry object, and spatial coordinates x, y. The scale property s of the spatial object snapshot is inherited by its geometry set and in turn by all the geometry and vertices.

2.1 The Scope of an Observation

The validity of a snapshot of a spatial object may be extended to cover a *scale interval*, i.e. $SO_s = (oid, R_s(G_s))$, $s \in [s_c, s_d]$ where $s_c \le s_d$. Following standard cartographic terminology, we regard a larger scale (e.g. $s_d > s_c$) as the scale corresponding

to a more detailed map while a smaller scale is for a map with less detail. A cartographic scale can then be any value in $(0, \infty)$.

We define a *scale range* as the collection of one or more scale intervals that may be continuous, adjacent or disjoint. In the following discussion, the term *scale range* and *scale interval* are used interchangeably when no confusion will be caused. Also, we refer to the smallest scale value in a scale range/interval as its *lower scale bound* and the largest scale value as its *higher scale bound*.

2.2 The Scale Range of a Spatial Object

A spatial object may be defined as a collection of all of its snapshots, which consist of all spatial information that we have on the object $SO = \{SOs_i \mid i = 1, n\}$. We denote the scale range of SO as S, the union of the individual scale ranges $[s_c, s_d]_i$ for $i= 1..n$.

The collection of geometry sets of all snapshots $G_g = \{g_j \mid j = 1, m\}$ forms the entire geometry set of SO. Thus a spatial object contains one geometry set only and a geometry object g may be referenced by more than one geometry set G_i of different spatial objects. It will inherit scale properties from all these geometry sets. A geometry object g is specified as $g = (gid, S_g, \{v_j \mid j = 1, m\})$ where S_g are scale properties of the geometry inherited from all geometry sets referring to this geometry. Assuming there are k geometry sets referring to this geometry, $S_g = \cup\, s_i$ for $i =1..k$. Note that s_i may be an interval.

In the general case a vertex $v(x, y)$ may be referred to by more than one geometry object belonging to the same or different spatial object. The specification of a vertex then becomes $v = (vid, S_v, vsn(s, gid), x, y)$. S_v are the vertex scale properties inherited from all geometry that refers to it. Assuming k geometry objects refer to the vertex, $S_v = \cup\, S_{gi}$ for $i =1..k$. The vertex set of a spatial object is $V = \{v_i \mid v_i \in g_j; g_j \in G_g\}$.

2.3 Continuous, Subsetting, and Non-subsetting Vertices

When scale priority values are attached to the constituent vertices of a linear or polygonal feature, several possibilities arise regarding the ranges of scales to which an individual vertex may be applicable. Here we distinguish between different types of scale-labelled vertex.

Let S' be the intersection of the scale range S_v of a vertex v_i and the scale range S of a spatial object SO referring to the vertex. If $S' = S_v \cap S = [s_a, s_b]$, $s_a \leq s_b$ is a continuous scale interval, we regard a vertex v_i as a *continuous vertex* of spatial object SO. If $s_b = s_{max} \in S$ (s_{max} is the higher scale bound of the scale range of SO), we regard v_i as a *continuous subsetting vertex* of SO, reflecting the fact that the vertex is present in the most detailed representation of SO**.** If S' is not continuous then the vertex is a *non-continuous vertex* of SO. Similar subsetting concepts apply to geometry objects.

3 Scale Coding

A multi-scale spatial database should be able to support scale change in a continuous manner across the entire scale range of the stored data. Due to the fact that the number of vertices in any dataset is limited, storage of a finite set of discrete scale intervals within the database is sufficient to support queries made at any scale. In practice the number of intervals required is normally quite small, unless an application demands very minor distinctions to be made between scale values.

A query on a multi-scale spatial dataset may be defined as: given a query scale s_q and a query window *QW*, find all spatial objects *SO* (with scale range *S*) in map *M* where $SO.MBR \cap QW \neq \emptyset$ and $s_q \in S$ and *MBR* refers to the minimum bounding rectangle of the spatial object. A group of vertices $\{v_i \mid s_q \in S_{vi} \wedge (x_i, y_i) \in QW\}$ will be retrieved to form this representation. A map is a finite set of spatial objects and hence a finite set of vertices. The scale range of a vertex is a set of scale intervals:

$$S_{vi} = \{[s_a, s_b] \mid s_b \geq s_a; s_{a+1} > s_b\}$$

Here $s_{a+1} > s_b$ is used because if $s_{a+1} = s_b$ the two intervals can be merged. Therefore the scale range for the map is:

$$S_{map} = \cup S_{vi} = \{[s_c, s_d] \mid s_d \geq s_c; s_{c+1} \geq s_d\}$$

which is a finite set of scale intervals. Note that these intervals are merged in the following manner: for $[s_1, s_2] \subseteq S_{vi}$ and $[s_3, s_4] \subseteq S_{vj}$ and $s_1 < s_3 < s_2 < s_4$, three intervals will be formed in the scale range of the map: $[s_1, s_3]$,$[s_3, s_2]$ and $[s_2, s_4]$. In the above expression $s_{c+1} \geq s_d$ is used because the intervals are created by bounding scale values coming from different vertices. If all scale values defining these intervals are unique, there are at most $r-1$ intervals for r different scale values between 0 and ∞. In this case, the scale range for the map is continuous and all scale intervals defined above can be merged to a single interval $[s_{min}, s_{max}]$.

Clearly for each of these intervals $[s_c, s_d]$ in S_{map}, the scale range of a vertex S_{v_i} either covers it or not covers it. Consequently for any $s_q \in [s_c, s_d]$, the same set of vertices (where $[s_c, s_d] \subseteq S_{v_i}$) will be retrieved. As there is no restriction on the possible value of s_q (if $s_q \notin S_{map}$, an empty representation is retrieved), we can conclude that we are able to classify vertices in a finite set of scale intervals to support queries based on continuously varying query scale values.

When all intervals in a scale range are stored, as: $SR = \{[s_i, s_j] \mid i,j=1, n; s_i \leq s_j\}$, the test for inclusion of a query scale value in a range is "If $s_q \in [s_i, s_j] \in SR$ Then *SR* covers s_q Else *SR* does not cover s_q". The scheme will work for all three vertex types of continuous, continuous subsetting and non-continuous. For the two continuous cases, the scale range has a single interval form: $SR = [S_{min}, S_{max}]$. For the continuous subsetting case, we have: $SR = [S_{min}, S_{map_{\max}}]$. Indeed, in practice only the S_{min} value is needed and the test can be further simplified to "If $s_q \geq S_{min} \in SR$ Then *SR* covers s_q Else *SR* does not cover s_q". For the non-continuous case, the resulting scale range will have a length-varying form that will certainly increase the complexity of storage and query process.

In practice, nominal variables (bit coding, C-style enumeration etc) may be used to represent bounding scale values or scale intervals to improve storage efficiency.

4 Database Access Schemes for Multi-scale Geometry Objects

In this section we introduce database storage structures that allow us to evaluate alternative approaches to implementing the geometry objects introduced in previous sections. A geometry object is denoted in the database as an MGEO (Multi-Scale Geometry Object) and it is referenced by an MSO (Multi-Scale Spatial Object). For purposes of experimentation we introduce the following constraints on the previous data model:

Constraint A: A vertex belongs to only one MGEO;

Constraint B: All vertices of the same geometry object are continuous subsetting.

The representation REP_M of a map at scale s_q and with a query window QW is:

$REP_M(s_q, \mathrm{QW}) = \{MGEO_{i,Sq} \mid s_q \in S_{mgeo};\ MGEO_i.MBR \cap QW \neq \varnothing\}$ and

$MGEO_{Sq} = \{v_i \mid s_q > s_{vmin_i}\}$.

The part of the multi-scale database that stores *MGEO*s may be represented conceptually by the relations $MGEO(MGEOID, S_{MGEO})$ and $VERTEX(MGEOID, VID, S_v, vsn, x, y)$. As the multi-scale geometry objects are the basic building blocks for implementing the whole model described above, in the remainder of this section we focus on methods for organising the storage of vertices within individual MGEOs.

4.1 Vertex Layer

If the VERTEX relation above were to be used directly in a real database implementation, the resulting vertex table would normally consist of a huge number of rows, making database queries very inefficient. It is desirable therefore to use structures larger than a single vertex as the storage unit. Hence we introduce the concept of a *vertex layer VL* as the set of all vertices which belong to the same MGEO and whose scale ranges share the same lower scale bound:

$VL = (lsn, S_{VL}, \{v_i \mid v_i \in MGEO;\ s_{vmin_i} = s_{vmin_j} = s_{VLmin},\ i \neq j;\ vsn_i < vsn_j,\ i<j\})$ where lsn is the identifier and S_{VL} is the scale range of the vertex layer. Applying the assumption of continuous subsetting within an MGEO, all vertices inside a vertex-layer have the same scale range, leading to the simplification:

$VL = (lsn, S_{VL}, \{v_i \mid v_i \in MGEO;\ S_{vi} = S_{VL},\ ;\ vsn_i < vsn_j,\ i < j\})$

4.2 Grouping Vertices in Geometry

As explained in Section 3.1, the scale range of a MGEO can be decomposed into a finite set of scale intervals $\{[s_n, s_{n-1}], ..., [s_3, s_2], [s_2, s_1]\}$, $(s_i < s_{i-1})$ where the scale range of a vertex is $[s_i, s_1]$ $(n \geq i > 1)$. Therefore, we may group vertices into several vertex layers corresponding to the following predefined set with n scale intervals $\{[s_n, s_1], ..., [s_3, s_1], [s_2, s_1]\}$. We can then define an MGEO with n vertex layers as follows:

$MGEO = (S_{MGEO} = [s_n, s_1], \{VL_i \mid i = 2, n\})$

$VL_i = (S_{VL_i} = [s_i, s_1], \{v_j \mid S_{v_j} = [s_i, s_1]\})$ and $\forall v_k (S_{v_k} = [s_i, s_1]) \Rightarrow v_k \in VL_i$. Given a query scale $s_q \leq s_1$, a subset of vertex-layers $\{VL_i \mid s_i \leq s_q\}$ forms a legible representation of the MGEO at this scale.

4.3 Storage Schemes with Explicit Vertex Sequence Numbers

Since vertex-layers are a scale-based decomposition of the vertex set of a MGEO, the ordering of vertices, relative to the original sequence, can be preserved inside each layer but not among different layers. When a subset of vertex layers is used to form a legible representation of the geometry, the original order of all selected vertices should be restored. One simple solution is to use an explicit sequence number (*vsn*) in the vertex data structure. Vertices in the subset of vertex-layers will then be sorted according to their *vsn* at the stage of object reconstruction. It should be remarked here that, alternatively, it is possible to maintain vertex ordering implicitly, without sequence numbers, by using references between vertices at different levels in combination with local ordering within layers [5]. A further option, which corresponds to most existing implementations of multiple scale data, is of course to store complete versions of the geometry at each scale-specific layer. We now introduce two approaches to storing sequence-numbered vertices that are organised in vertex layers.

4.3.1 Single Entity Geometry (BLOBSN)

As the binary large object data type (BLOB) is widely available in modern DBMS, a MGEO with many vertices may be stored as a single entity in the database. A BLOB-based representation of a MGEO has the following structure: $<(s_n, end_offset_n), ..., (s_1, end_offset_1)><VL_n, ..., VL_1>$. Thus vertex layers are stored after a head section (the length of which, fixed or varying, is known). Inside the head section, the *end offset* of each vertex layer is stored as an index used by application programs to read vertex layers from the BLOB object. Note that the starting offset of a vertex layer is the same as the ending offset of the layer preceding it. In this scheme spatial indexing is performed on the complete BLOB and hence refers to the entire geometry object represented by the MGEO.

4.3.2 Multi-segment Geometry (VBSN)

If spatial indexing is to be performed on parts of a geometry object then the vertex layers of a MGEO can be divided into multiple segments. If we set a limit to the size of each segment, they may be implemented as an extended data type in the DBMS and stored as fields of a record or row. The limit on size can either be the number of vertices in a segment or the total data volume of the segment. In the current treatment, we will use the number of vertices (*bsz*) as the size constraint. We regard such segments as *vertex blocks* (VB) where

$$VB = (bsn, S_{VB}, \{v_i \mid i=1,n;\ n \le bsz;\ v_i \in MGEO_k;\ S_{v_i} = S_{VB};\ vsn_i < vsn_j,\ i < j\})$$

and the block sequence number *bsn* is the identifier of the vertex block. Each vertex block is indexed by a minimum bounding rectangle and for two blocks VB_1 and VB_2 belonging to the same vertex-layer, if $v_i \in VB_1$ and $v_j \in VB_2$, and $bsn_1 < bsn_2$ then we have $vsn_i < vsn_j$.

5 Dynamic MBR and Object Clipping

In this section we address the problem of defining the extent of bounding rectangles that can be used for spatial indexing of vertex blocks. An important issue in querying a spatial database is to maintain topological consistency of objects overlapping the boundary of the query window. To do this it is necessary to retrieve vertices outside the query window that are neighbours of vertices of the same MGEO that are inside the query window. The associated edges can then be clipped to the window boundary. In an MGEO it important that the retrieved external vertices are at a level of detail equivalent to that of the neighbouring internal vertices

A simple solution is to retrieve the entire set of vertices of the object at the query scale. This is the approach adopted with the BLOBSN storage scheme. However, this approach may result in significant data redundancy as an object with many vertices may have only a small intersection with the query window. A solution to the problem is to associate with each vertex, or vertex block, a dynamic minimum bounding rectangle (DMBR) that includes the location of neighbouring vertices at the same or lower levels of detail, plus all intermediate vertices. The principle underlying the DMBR was introduced originally in [6] to support proximity queries.

5.1 Dynamic MBR of a Vertex

For vertex v_i=(vsn_i, S_{v_i}=[s_j, s_1], x, y) in a MGEO, the $DMBR_{v_i}$ = (x_{min}, y_{min}, x_{max}, y_{max}). It covers: 1) the vertex itself, 2) the two neighbouring vertices that have a lower scale bound equal to or smaller than the lower scale bound of the vertex (i.e. having equivalent or less detailed scales), and 3) all other vertices between these two vertices that have a greater lower scale bound (and hence represent greater detail). Note that for closed curve geometry, the search for the two neighbouring vertices should be extended beyond the beginning or the end of the vertex sequence when necessary. For the case where there is only one vertex with the scale range of the MGEO, its DMBR is the MBR of the MGEO.

5.2 Dynamic MBR of a Vertex-Block and Object Clipping

With the above definition of DMBR for an individual vertex, we can define the DMBR for a vertex-block as $VB.DMBR$ = MBR({$v_i.DMBR$ | $v_i \in$ VB}). This is the MBR of the DMBRs of all its vertices.

When a vertex v_i = (S_{v_i}=[s_k, s_1]) is retrieved by a query Q($s_q \geq s_k$, QW), its adjacent vertex v_r with equal or lower scale bound, if outside the QW, should also be retrieved. However, this can not be achieved by a conventional query of the form:

$$Q(s_q, QW) \leftarrow \{v_i = ([s_i, s_1], x, y) | \; s_i \leq s_q \wedge \text{Overlap}(QW, (x,y))$$

as v_r is outside QW. With the help of the DMBR, we can amend the query to:

$$Q_{DMBR}\,(s_q, QW) \leftarrow \{v_i = ([s_i,s_1], DMBR, x, y) \mid s_i \leq s_q \wedge (\text{ Overlap}(QW, v_i.DMBR)\}$$

In this case an externally adjacent vertex v_r will be retrieved because its DBMR will include the location of the internal neighbour and hence overlap QW. In general, most vertices outside the query window will not pass this test, thereby avoiding redundant

retrieval. Note that the definition of DMBR is simply an alternative to conventional MBR so it will not introduce any storage or performance overhead.

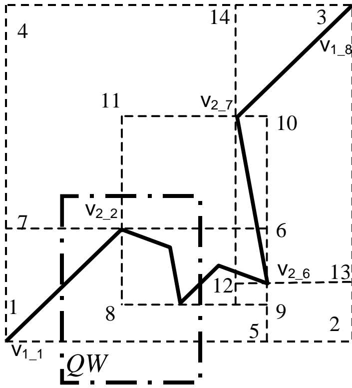

Fig. 1. DMBRs of vertices

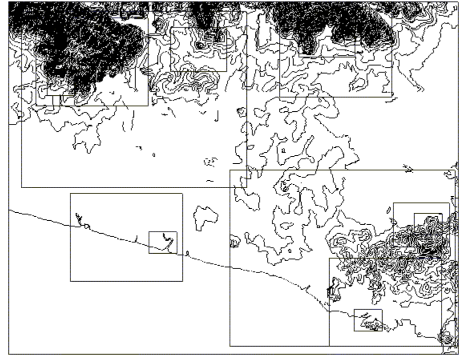

Fig. 2. The real dataset and query windows

The above convention also applies to the situation when vertex-blocks are used. If the vertex v_i mentioned above is a boundary vertex of its block VB_i (i.e. the first or last in the vertex sequence of the block), the vertex-block (say, VB_r) containing v_r should be retrieved. Since the DMBR of a vertex-block covers the DMBRs of all its vertices, if $v_r.DMBR$ overlaps QW then so will its vertex block $VB_r.DMBR$.

The query for a vertex-block based dataset is therefore:

$$Q_{\text{DMBR}}(s_q,QW)\leftarrow\{VB_i=([s_i,s_1],DMBR,\{v_j|j=1,n\})\mid s_i\leq s_q \wedge \text{Overlap}(QW, VB_i.DMBR))\}$$

For example, in Fig. 1, a vertex v_{i_j} refers to a vertex with lower scale bound i and sequence number j. According to the definition of DMBR of vertices, the DMBRs of v_{1_1} and v_{1_8} are defined by rectangles (1, 2, 3, 4), v_{2_2} (1,5,6,7), v_{2_6} (8,9,10,11) and v_{2_7} (12,13,3,14). When a query (QW, S_q <=2) is made, v_{2_2} is inside QW and retrieved directly and v_{1_1}, v_{2_6}, v_{1_8} are also retrieved because their DMBRs intersect QW. Therefore the intersection of the object with the boundary of QW is represented correctly by segment v_{1_1}-v_{2_2} and v_{2_2}-v_{2_6}. As already mentioned, there will still be some redundant vertices retrieved (v_{1_8}). However, v_{2_7} is not retrieved so that the redundancy rate is reduced. Note that the three unlabelled vertices in the figure have a larger lower scale bound and make no impact on the query result.

6 Object Caching for Zoom-In, Zoom-Out, and Panning

Querying a multi-scale spatial database is rarely a one-off operation. Users may want to zoom-in, zoom-out or pan around the entire dataset. Here we define these basic operations in the context of the vertex block storage scheme, before introducing some procedures to facilitate their implementation with cached data on the client.

In what follows each local windowing operation results in a new client-side result set RS_{m_1} of the multi-scale data that supplements the existing client-side result set

RS_{m_0}. We start by retrieving an initial representation RS_{m_0} using a window QW_0 and a query scale of s_{q_0}:

$RS_{m_0} = (Q_{DMBR}\ (s_{q_0}, QW_0) \leftarrow$
$\{VB_i=([s_i,s_1],DMBR,\{v_j\ /\ j = 1, n\ \})\ |\ s_i \leq s_{q_0} \wedge \text{Overlap}(QW_0, VB_i.DMBR))\}$

We may define a *panning operation* with a new window QW_1 as:

$RS_{m_1} = RS_{m_0} + (Q_{DMBR}\ (s_{q_0}, QW_1) \leftarrow$
$\{VB_i=([s_i,s_1],DMBR,\{v_j/j=1,n\})|s_i \leq s_{q_0} \wedge \text{Overlap}(QW_1, VB_i.DMBR) \wedge$
$\neg\text{Overlap}(QW_0, VB_i.DMBR)\ )\}$

Note that those vertex-blocks that are retrieved by the previous queries and are outside the new query window QW_1 become redundant for the current map.

A *zoom-in operation* is with a window QW_1 is defined as:

$RS_{m_1} = RS_{m_0} + (Q_{DMBR}\ (s_{q_1}, QW_1) \leftarrow$
$\{VB_i=([s_i,s_1],DMBR,\{v_j\ |\ j=1, n\})\ |\ s_{q_0} < s_i \leq s_{q_1} \wedge \text{Overlap}(QW_1, VB_i.DMBR)\}$

Implementation of this operation retrieves some additional vertex-blocks of MGEOs that have been partially retrieved in previous queries and plugs the new vertex blocks into the previous representation. Note that QW_0 is normally but not necessarily larger than QW_1.

Finally, a *zoom-out operation* to scale s_{q_1} may be defined as

$RS_{m_1}=RS_{m_0} + (Q_{DMBR}\ (s_{q_1}, QW_1) \leftarrow$
$\{VB_i=([s_i,s_1],DMBR,\{v_j/j=1,n\})|\ s_i \leq s_{q_1} \wedge \text{Overlap}(QW_1, VB_i.DMBR) \wedge$
$\neg\text{Overlap}(QW_0, VB_i.DMBR)\ \}$

In this case, some new vertex-blocks outside QW_0 may be retrieved while some existing vertex-blocks with a larger low scale bound ($s_i > s_{q_1}$) become redundant to the current map.

These three operations are defined relative to the operation immediately preceding them. In a more general situation, the initial result set is the result of a series of previous queries $\{Q_0, Q_1, \ldots, Q_n\}$. In this case, a new query Q_{n+1} will generate a new result set as $RS_{n+1} = RS_n + \{VB_i\ |\ VB_i \in (Q_{n+1} \wedge \neg(Q_0 \vee Q_1 \vee \ldots \vee Q_n))\}$. Obviously, such an expression could generate a very long and inefficient query statement. Therefore in practice a balance must be found between the increased complexity of the query statement and reduced retrieved data volume.

The purpose of supporting object caching is to make use of data that have already been retrieved and reduce the overall data communication between client and server. Although the method of supporting object caching is highly implementation dependent, we would like to address some relevant issues here.

6.1 An Enlarged Query Window

If panning operations are to be expected, then there are benefits to be gained from retrieving a larger query window than that requested by the user. We assume that in the application program there will be a mechanism to map a query window of particular size, say QW_0, to a query scale, say s_q, and that objects with $[s_i \leq s_q, s_1]$ that overlap QW_0 will be retrieved. When $s_i < s_q$, we can map s_i back to a window QW'_0 which is larger than QW_0 and shares the same geometric centre with QW_0. By using this enlarged query window QW'_0 to make the query, the retrieved dataset will remain

relevant when subsequent panning or zoom-out operations are restricted to being within QW'_0 and no new queries will need to be carried out. The size of QW'_0 will depend upon the storage resources of the system. In addition, we may use a larger query scale (say, $s'_q > s_q$) to retrieve more detailed data of objects so that zoom-in operations to scales smaller than or equal to s'_q can also be supported.

6.2 Object Reconstruction with Cached Geometry

Both the BLOB and Vertex Block storage schemes have the potential to support object caching in a graphic presentation system. For a one-off query, the normal sequence of actions is:

1) retrieve vertices in server representation;
2) reconstruct client-side representation of vertices from their server representation, put vertices into a linear data structure, which can be sorted efficiently, and delete the server representation;
3) sort vertices in the linear structure;
4) convert vertices to logical/device coordinates for graphic presentation.

If an associative data structure (such as the *map* template in C++ standard library) is used, steps 2 and 3 may be merged. In step 4, a new linear data structure which is compatible with the graphic presentation system might have to used and new vertices compatible with the graphic system have to be created. In this case, the original linear/associative structure may be discarded.

To support object caching in a multi-query situation, there are some alternatives for zoom-in/panning:

a) keep the original server representation and repeat steps 2, 3, 4 when new data are added;

b) keep the linear structure in 2 and 3, add new data directly into it and then carry out step 4 after re-sorting (for which there will be no need in the case of an associative structure).

For zoom-out, if the original server representation is not preserved, the scale range of each individual vertex has to be stored in the data structure for client-side vertices, in order to eliminate extra vertices in step 4.

7 Indexing the Scale Ranges

A technique for indexing on scale in combination with spatial indexing was introduced in [17]. We pursue a related idea here with scale values associated with individual vertices as well as complete objects. We consider several alternative approaches. At one extreme it is possible to index simply on scale. For the continuous subsetting cases, a B-tree may be used to index the scale range field in a multi-scale spatial database. One way to index and query non-subsetting data is to use two fields for the scale range and to build a B-tree on the higher scale bound in the scale range and use an extra expression to compare the query scale value and the lower scale bound of the range. Alternatively, if a generic R-tree is available, a 1-D R-tree index may be built for the scale range column by defining necessary operations on the scale

range values. An integrated approach is to combine the scale range with the spatial extent of objects (i.e. MBR) in a multi-dimensional R-tree index. For the non-continuous case, the above technique also applies but extra value testing is needed. Further design and implementation details are given in section 8.2 and in section 8.5.4 in which we perform a comparison of the three approaches of a B-tree index on scale, an R-tree index on space and a 3D R-tree that integrates indexing on scale with space.

8 Implementation and Experimental Results

8.1 Database Schemas

For purposes of evaluation four database schemas have been implemented. The first two, referred to as VBSN and VBSN_NONCLIP, are based on the vertex block design introduced in Section 4.3 and respectively include and omit clipping support on vertex blocks. The third uses blob storage (BLOBSN), as described in Section 4.3. The fourth is a blob based multi-version schema (BLOBMV) under which a complete representation for each scale interval is stored. Thus in the latter case all vertices present, at each of the multiple scales, are stored in correct order, with no need for a reconstruction step. In all implementations there are two primary tables: an MSO table and an MGEO table.

8.1.1 BLOBSN and BLOBMV

For these implementations a row in the MSO table is represented by a tuple (*mso_id*, *lsb*, *hsb*, *geomtype*, *mbr*). The value *geomtype* indicates whether the object is an open or a closed curve. *lsb* and *hsb* are the lower and higher scale bounds of the scale range of the object. A row in the MGEO table has the form of (*mso_id, lsb*, *hsb*, *mbr, blob_vertices*). Here *blob_vertices* is a handle to the blob object in the database storing all vertices of this MGEO. The internal structure of the blob object for BLOBSN has been described in section 4.3.1. The blob for BLOBMV is simply a succession of the multiple versions in ascending order of their lower scale bound.

8.1.2 VBSN_NONCLIP and VBSN

The MSO table for VBSN_NONCLIP has the same structure as for BLOBSN. The form of its MGEO table is (*mso_id, lsb*, *hsb*, *bsn, mbr, opaque_vertices*). *bsn* is a sequence number assigned to a vertex-block and *opaque_vertices* is an Informix opaque data type containing vertices of this vertex-block.

The MSO and MGEO tables of VBSN have the same structure as those for VBSN_NONCLIP except that dynamic MBRs of vertex-blocks instead of conventional MBRs are stored in the *mbr* column.

8.2 Spatial-Scale Indexing

8.2.1 Indexing on Scale

Using the generic B-tree and R-tree utilities in Informix, we tested three methods to create indices for MSO/MGEO tables on their scale/spatial extent columns to support efficient query executions. The first method is to generate a B-tree index, referred to here as *mso_scale_index*, on the lower scale bound column *lsb*. A query statement on the *mso_table* generated from a query $Q(s_q, QW)$ has the following form:

Select {+index(*mso_table mso_scale_index*)} *mso_id*, *geomtype* from *mso_table*
where *lsb* <= s_q AND s_q <= *hsb* AND overlap(*mbr*, *QW*);

where the directive {+index()} forces the query parser to use the specified index whenever possible.

The query above refers to data that are encoded with continuous vertices. To handle scale bounds of non-continuous scale ranges, we can create an extended data type, *ScaleRange*{Scale *lsb*, *hsb*;}, to contain the two bounds and define a group of operations (such as *Overlap*, *Contains*, *Within*, *Equal*) which are, in Informix terms, R-tree strategy and support functions on this data type. Creating a 1-d R-tree index on columns *sr* (scale range) using this data type, the query becomes:

Select {+index(*mso_table mso_scale_index*)} *mso_id, geomtype* from *mso_table*
where overlap(*sr*, 's_q, s_q'::ScaleRange) AND overlap(*mbr*, QW);

As the overlap operation will take two parameters of the type *ScaleRange*, we have to convert the query scale s_q to a zero-extent scale range in the query statement.

8.2.2 Spatial Indexing

All *mbr/dmbr* columns have the type *GeoRect*, which is an extended data type representing a rectangle with R-tree strategy/support functions defined for it. With an R-tree index (mso_spatial_index) created on the *mbr* column the query statement is:

Select {+index(*mso_table mso_spatial_index*)} *mso_id, geomtype* from *mso_table*
where *lsb* <= s_q AND overlap(*mbr*, QW)

8.2.3 Integrated Spatial-Scale Indexing

In order to index spatial and scale dimensions integrally, we defined an extended data type *GeoCube* {*Scale lsb, hsb; Coord minx, miny, maxx, maxy*;}. R-tree strategy/support functions, including *overlap*, are also defined for this type. With this data type, we can merge the scale range column(s) and the spatial extent column into a single column (say, *mbc*, the minimum bounding cube). The schema of the MSO table then becomes (*mso_id*, *geomtype*, *mbc*) and we can create a 3-D R-tree index (*mso_integrated_index*) on the *mbc* column and the query statement becomes:

Select {+index(*mso_table*, *mso_integrated_index*)} *mso_id, geomtype*
from *mso_table*
where overlap(mbc, 's_q, s_q, QW.minx, QW.miny, QW.maxx, QW.maxy'::GeoCube);

8.3 Generation of Sample Datasets

We regard the process of generating a multi-scale map dataset from a source dataset at a single large scale as one of calculating scale ranges for map objects and vertices and then, if necessary, classifying them according to a set of predefined scale intervals. From the viewpoint of map generalisation, this requires a mapping between various parameters of generalisation algorithms and the predefined scale interval set. It should be remarked that in theory the process of creating a multi-scale dataset could be based on several source datasets at different base scales, but for the sake of simplicity, we do not consider that possibility further.

As the generalisation of a map dataset with multiple geometric feature types remains a challenging task, that has yet to be fully automated, we restrict ourselves to handling map datasets with a single geometric feature category, namely open or closed polylines. One real map dataset and two simulated map datasets were used to test various multi-scale schemes presented in this paper.

The real map dataset (Fig. 2) consists of contours extracted from a 1:5000 topographical map which covers a region of about 50 km^2 with 627 contour lines and 48478 vertices. An amended Douglas-Peucker algorithm was used to generate scale range values of vertices taking account of the "extending beyond endpoint" situation [2]. Also, four selection levels decided by experiment were set to eliminate entire contours with certain elevation values as scale decreases. A base-2 scale interval set was used (i.e. $\{1:2^i \times 1000 | i = 0, n\}$) in this process and the values of tolerance in the Douglas-Peucker algorithm were mapped to this scale interval set.

Since the size of the real dataset is fairly small for the purpose of performance testing and the data points/lines inside this dataset are distributed unevenly, we used a Koch curve (order = 3) along with the Douglas-Peucker vertex selection criterion to create two simulated map datasets (see extracts in Fig. 4) in a "central place" style (and hence with some degree of geographical reality). A base-3 scale interval set was used (i.e. $\{1:3^i \times 1000 | i = 0, n\}$). Datasets created in this way result in a similar density of retrieved objects on the presentation medium for various scales, which conforms to the geometric presentation of a real map series. In addition, using this method, a dataset of arbitrarily large size can be generated easily.

The first simulated dataset covers a 10km by 8km region at 1:5000 base scale with 17,187 objects and 1,335,534 vertices. The second set is much larger, at 40km by 32km, with 273,996 objects and 21,073,087 vertices. In order to eliminate objects of very small size, curves with less than three levels are not created. Consequently, the smallest object has an extent of about 25m with 48 vertices.

8.4 Test Environment and Query Generation

Our experiments were carried out on a PIII-600MHz PC with 128MB RDRAM running Windows NT Workstation 4.0. The extensible object-relational DBMS used here was Informix Dynamic Server 2000 ver.9.20.TC3. All application programs to generate and process test datasets, load data to database server and make queries on the server were written in C++ with Informix Object Interface for C++ v.2.6.

The location of query windows on the real dataset was determined arbitrarily as covering a "region of interest". For the two simulated map datasets, a series of query windows was generated automatically with each window having the size of one quarter of the area of the window at a higher level. The query scale value was calculated by the query program according to the size of the query windows and an input value for "*screen resolution*", assuming a fixed display device area. In the results presented here, we used resolutions of 0.2mm, 1mm and 5mm respectively. Fine screen resolutions result in retrieving more detailed representations, while coarser resolutions lead to a smaller query scale value and hence less detail.

In our application programs, a query to the map database is carried out in two steps. First a set of MSO objects is retrieved and then in the second step their associated geometry objects are retrieved and reassembled. The results shown are the average of several runs when applicable.

8.5 Results

We carried out various measurements on different aspects of the query and object reconstruction process. In the following discussions, we refer to "retrieval time" (*RT*) as the interval between when the query is issued and the result set is returned (for blob object based schemes, the time to read data from the blob object should be added). "Process time" (*PT*) is the interval between the return of the result set and completion of reconstruction of the representation of the map dataset (for the blob object based scheme, the time to read data from blob objects should be excluded). "Client-side data redundancy"(*CDR*) is the ratio of the number of retrieved vertices outside the query window (V_{out}) to the number of vertices inside the query window (V_{in}).

8.5.1 Multi-version *vs.* Multi-scale

Our results (Table 1, 10k by 8k simulated dataset, where R_{scr} is the "screen resolution" used and Q_Res is the actual field resolution) shows that the multi-version scheme, understandably, usually has a better process time than the sequence number based scheme, which needs to carry out a sorting operation after retrieval in order to perform object reconstruction. However, its advantage on this aspect is typically no better than about 10%.

Table 1. Process time of BLOBSN and BLOBMV schemes (in seconds)

Relative *QW* Area	R_{scr} = 0.2mm			R_{scr} = 1.0mm			R_{scr} = 5.0mm		
	SN PT	MV PT	Q_Res	SN PT	MV PT	Q_Res	SN PT	MV PT	Q_Res
0.39%				0.431	0.37	1	0.28	0.23	9
1.56%				0.952	0.942	3	1.162	0.921	9
6.25%	3.742	3.497	1	3.496	3.234	9	1.422	1.274	27
25%	16.004	15.92	3	15.763	15.833	9	0.741	0.822	81
100%	89.701	89.501	3	20.998	19.625	27	3.025	3.357	81

The main disadvantage of the multi-version scheme appears to be the server side data redundancy (SDR). Theoretically, we would expect the data redundancy to be:

$\lim_{n \to \infty} R_{rud} = \frac{1}{N}$ where N is the average number of vertices inserted between two vertices at a higher level and n is the number of vertex layers (proof is omitted here). In practice we encountered much higher redundancy rates which implies a very small N. Table 2 shows the data volumes of the real map dataset using the two schemes:

Table 2. Data volume of blob schemes in bytes(the real dataset)

	BLOBSN	BLOBMV	SDR
No-selection	969,560	4,131,664	3.261
Selection	969,560	2,624,688	1.707

When the selection generalisation operator is used (to eliminate some entire contours as scale decreases), the number of vertex layers is relatively small and the redundancy is lower. However, a redundancy rate of 1.7 is still significant. Indeed, the result shown above is based on a predefined scale interval set with only a few intervals (similar to an ordinary topographical map series). If the multi-version method is used to support continuous scale change, the number of versions will be much higher. Another difficulty associated with the multi-version scheme is the preservation of information on scale range of individual vertices. If this information is to be stored for each vertex, the redundancy rate will be even higher. It is also difficult to maintain efficiently the identity of a vertex that is present in multiple vertex layers, which may be important for some analytical purposes. This may be solved by introducing an explicit identification number (or indeed, a sequence number) for each vertex, which, however, will turn the scheme into a sequence number based one with multiple versions. Finally, the multi-version scheme does not provide support for efficient client-side object caching operations other than panning.

8.5.2 BLOB *vs.* Vertex-Block

The main difference between the two implementation strategies of blob based and vertex-block based schemes is that vertices are stored in-row under the vertex-block scheme (VBSN) while vertices in blob objects are stored separately in a different “dbspace”. To open a blob object and read data from it is a relatively expensive operation. On the other hand, a MGEO table based on a blob scheme will have fewer rows which will result in a better query performance (reflected in our “retrieval time”). However, under our software/hardware environment, this advantage on query execution did not result in a better overall performance, as the operation of opening blob objects and reading data from blob objects is the bottleneck of the whole process. Table 3 below shows the average timing results of 4 runs on the first simulated dataset with a query window of 10km by 8km (i.e. the extent of the dataset) and a query scale 1:15000 (the second largest in the bounding scales of the scale interval set). R_{blob} is to the time to retrieve the content of blobs. Note that the application programs have been optimised using customised memory management schemes.

Table 3 (The 10k by 8k simulated dataset)

BLOBSN (in seconds)				VBSN (in seconds)		
RT' + R_{blob}	PT'-R_{blob}	R_{blob}	Overall	RT	PT	Overall
55.541	61.701	50.556	117.242	60.828	14.805	75.633

8.5.3 Non-clipping *vs.* Clipping

When object clipping is not supported, all vertices in a geometry object that are visible at the query scale have to be retrieved in order to maintain topological consistency with the boundary of the query window. In some cases, this can result in large client-side data redundancy. Although the approach of sending all vertices may be useful for a client-side object caching scheme, it may generate an unstable query response performance. With DMBR-based clipping support, we are able to reduce the data redundancy significantly. Fig. 3 shows the graphic presentations of some of the datasets retrieved from queries without clipping support and queries with clipping support and different block sizes. As shown in the figures, when the vertex block size increases the data redundancy of the retrieved dataset will increase as well.

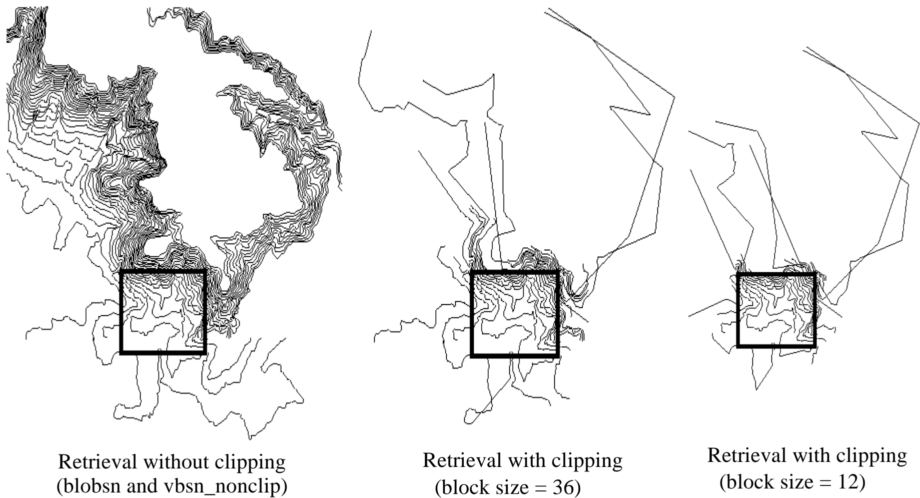

Fig. 3. Data retrieved without and with vertex-level clipping. The spurious line cross-overs outside the query window are due to arbitrary variations in detail of redundant data beyond the immediate vicinity of the window. Data inside the window are at consistent levels of detail.

The reduction of data redundancy results in a much better performance. In our experiment with 14 queries on the real dataset (block size = 1, not optimised), the average query time with clipping was 87% (SD=25%) of that of non-clipping, while the vertex process time was only 33.4% (SD=22.4%) of the non-clipping scheme.

8.5.4 Scale Indexing, Spatial Indexing, and Integrated Indexing

Here we compare experimental query response times using B-tree indexing on scale, R-tree indexing on space and integrated 3D R-tree indexing on space and scale. Fig. 5 shows the results of queries for all three types of index across a wide range of sizes of spatial window, with corresponding change in the computed scale value (retrieval resolution), for a screen resolution of 5 mm. The results show that indexing on scale alone gives very poor results for small window sizes while the R-tree indexing on space alone gives poor results for large window sizes. The reason is that in both cases the intermediate result sets returned by the indexed query directive in the composite

query statements (described in Section 8.2) are very large and subsequently the other query directives are executed sequentially on these large result sets. The integrated (3D R-tree) indexing method strikes a balance between space and scale resulting in overall superior performance (which applies for all screen resolutions though the other results are not presented here).

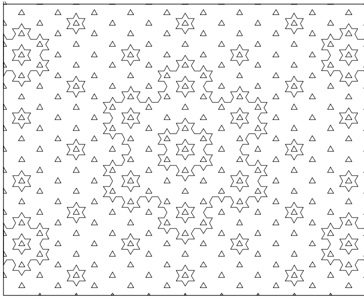

Fig. 4. Part of a simulated map dataset based on Koch curve

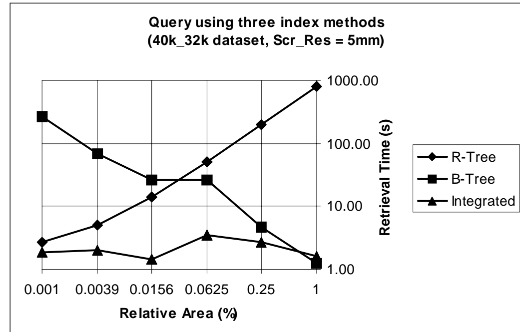

Fig. 5. Comparison of indexing with R-Tree (space), B-Tree (scale) and space-scale 3D R-tree

The results indicate that when the query window is very large, the B-tree generates a slightly better performance than the 3D R-tree. We believe this is due to the uneven distribution of spatial objects on the scale dimension (conforming here to Topfer's "radical law"). Thus, when the query window is large, the B-tree index filter retrieves a very small result set and a subsequent spatial extent check can then be done quickly. On the other hand, the 3D R-tree treats the three dimensions equally which may result in a decomposition of space whose configuration is not optimal when the query window is large and the query scale is small. It appears that if at the top level (or a few levels at the top) of the 3D R-tree the decomposition were to be carried out with a scale priority instead of a spatial priority, the overall performance of the integrated 3D R-tree could be further improved.

8.5.5 Client-Side Object Caching

So far the results presented have been based on single queries. We implemented the two client-side object caching schemes described in Section 6.1.2, with the VBSN storage scheme, to test the performance of a series of zoom-in, zoom-out and panning queries, with and without object caching. The first scheme maintains an indexed data structure (we used C++ *map* template) for vertices. When new vertices are retrieved, they are added into this data structure and sorted. At the stage of linear feature reconstruction, prior to display, all vertices in the indexed structure are examined and those with a lower scale bound smaller than the query scale are selected. The second scheme keeps a list of retrieved vertex-blocks for each object. When a linear feature is to be reconstructed, the block list is scanned. Selected vertices are then put into a linear data structure and sorted according to their sequence number.

For each action of zoom-in, zoom-out and pan, an initial query is carried out to construct a client-side result set. Subsequently two zoom-in/zoom-out/panning operations are carried out and new objects/vertices are merged into the initial result set. Table 4 illustrates the data volume reduction of caching in comparison to non-caching using the 10k by 8k dataset. *Ver_Num* is the number of vertices added into the result set after two operations and *Data_valume* is the total volume of vertex data retrieved from the server over the two operations. In all these operations, one level of caching is used (i.e. the query statement of the previous query is used to optimise the new query statement). The percentage is simply the cached data divided by the non-cached data. Our results show that the vertex-block list based scheme provides a better performance for on-line reconstruction for the retrieved objects. In addition, this scheme has the potential of supporting client-side clipping at the stage of constructing the graphic presentation if we store the DMBR of vertex-blocks on the client-side. We may do the same for the other scheme but that would involve storing the DMBR of each vertex.

Table 4. Caching vs. non-caching after two operations

Operation	Scheme	Ver_Num		Data_Volume (bytes)	
Zoom-In	Caching	141,449	74.07%	2,828,980	74.82%
	Non-caching	190,966		3,781,200	
Zoom-out	Caching	162,574	76.67%	3,253,920	76.96%
	Non-caching	212,483		4,227,900	
Panning	Caching	23,044	69.83%	482,640	70.49%
	Non-caching	32,998		684,720	

9 Conclusions

The need to access spatial data at multiple scales provides a strong motivation to develop efficient strategies for implementation of and access to multi-scale spatial objects. In this paper we have addressed several practical issues in implementing multi-scale data access schemes in which vertices of geometric objects are associated with scale-priority values and explicit sequence numbers. In the context of scale range-coding of geometry, we have shown that, because of the discrete nature of scale ranges attached to vertices, scale-specific layers of vertices can be used to maintain scale-range attributes of the vertices, and hence support queries on arbitrary scale values. The implemented multi-scale storage schemes were based on blob storage of scale-specific layers, in which only the entire geometric object was spatially indexed, and on extended relational storage of spatially-indexed vertex blocks (in the manner of the PR-File) respectively. These were compared with a conventional multi-version approach to storage of geometry in blobs at multiple levels of detail. In our implementation, the multi-version storage scheme had significant storage overheads compared to the multi-scale geometric object schemes, while providing no more than about 10% improvement in timings for reconstruction of scale-specific geometric objects. In a comparison of the spatially-indexed vertex block based scheme with the blob multi-scale scheme, the former was found to be clearly superior, demonstrating the merits of spatial indexing of subsets of vertices. Varying the resolution of this spatial indexing

(with different block sizes), the results confirm the expectation that larger block sizes combine faster retrieval with greater redundancy of retrieved data.

Following on from the work of [17] we have compared R-tree spatial indexing, B-tree scale-indexing and a 3D generic R-tree that integrates space and scale. The integrated method was found to outperform the other two methods.

Two approaches to client-side reconstruction of linear features from cached geometry have been presented. The advantages in object reconstruction of both multi-resolution caching strategies were demonstrated relative to a non-caching scheme. The best performance was obtained with client-side maintenance of geometry in the form of vertex blocks, as opposed to insertion into an index list of scale-coded vertices.

References

1. Ballard, D., Strip trees: a hierarchical representation for curves. *Communications of the ACM*, 24, 1981, 310-321.
2. Gunther, O., *Efficient Structures for Geometric data Management.* Lecture Notes in Computer Science, Vol. 337, 1988, Berlin, Springer-Verlag.
3. van Oosterom, P., *Reactive Data Structures for Geographic Information Systems*, 1994, Oxford, OUP.
4. Cromley, R.G., Hierarchical methods of line simplification. *Cartography and Geographic Information Systems*, 18(2), 1991, 125-131.
5. Jones, C.B. and I.M. Abraham. Design considerations for a scale-independent database. In *Second International Symposium on Spatial Data Handling*, 1986, Seattle, 384-398.
6. Jones C.B., D.B. Kidner and J.M. Ware. The Implicit TIN and multiscale spatial databases, *The Computer Journal* 37(1), 1994, 43-57.
7. Becker, B., H.-W. Six, and P. Widmayer. Spatial priority search: an access technique for scaleless maps. In *ACM SIGMOD*, 1991, Denver, Colorado, 128-137.
8. van Putten, J. and P. van Oosterom. New results with generalised area partitionings. In *8th International Symposium on Spatial Data Handling*, 1998, Vancouver, 485-495.
9. Bertolotto, M. and M.J. Egenhofer. Progressive vector transmission. in *7th ACM Symposium on Advances in Geographic Information Systems*, 1999, Kansas City, ACM Press, 152-157.
10. de Berg, M., M. van Kreveld, and S. Schirra, Topologically correct subdivision simplification using the bandwidth criterion. *Cartography and Geographic Information Systems*, 25(4), 1998, 243-257.
11. van der Poorten, P. and C.B. Jones. Customisable line generalisation using Delaunay triangulation. In *19th Conf. of Int. Cartographic Association*, 1999, Ottawa, CDROM.
12. Saalfield, A., Topologically consistent line simplification with the Douglas-Peucker algorithm. *Cartography and Geographic Information Science*, 26(1), 1999, 7-18.
13. Buttenfield, B.P. Sharing Vector Geospatial Data on the Internet. in *19th Conference of International Cartographic Association*, 1999, Vancouver, CDROM.
14. Jones, C.B., *et al*. Multi-Scale Spatial Database Design for Online Generalisation. in *9th International Symposium on Spatial Data Handling*, 2000, Beijing, IGU, 7b.34-44.
15. Ware, J.M. and C.B. Jones Conflict Reduction in Map Generalisation Using Iterative Improvement. *GeoInformatica,* 2(4), 1998, 383-407.
16. Harrie L. and T. Sarjakoski. Simultaneous Graphic Generalisation of Vector Data Sets, submitted to GeoInformatica.
17. Horhammer, M. and M. Freeston. Spatial indexing with a scale dimension. In *6th International Symposium on Spatial Databases, SSD'99*, 1999, Hong Kong, Springer, 52-71.