Knowl Inf Syst
DOI 10.1007/s10115-008-0132-8

REGULAR PAPER

A hybrid aggregation and compression technique for road network databases

Ali Khoshgozaran · Ali Khodaei · Mehdi Sharifzadeh · Cyrus Shahabi

Received: 15 March 2007 / Revised: 4 January 2008 / Accepted: 19 January 2008

Abstract Vector data and in particular road networks are being queried, hosted and processed in many application domains such as in mobile computing. Many client systems such as PDAs would prefer to receive the query results in unrasterized format without introducing an overhead on overall system performance and result size. While several general vector data compression schemes have been studied by different communities, we propose a novel approach in vector data compression which is easily integrated within a geospatial query processing system. It uses line aggregation to reduce the number of relevant tuples and Huffman compression to achieve a multi-resolution compressed representation of a road network database. Our experiments performed on an end-to-end prototype verify that our approach exhibits fast query processing on both client and server sides as well as high compression ratio.

Keywords Multi-resolution compression · Vector data · Aggregation · Road networks · Spatial databases · GIS

1 Introduction

Current advances in GIS applications and online mapping tools have enabled navigation and browsing of geographic maps. With an online geospatial service, a remote user issues a

A. Khoshgozaran (✉) · A. Khodaei · M. Sharifzadeh · C. Shahabi
Department of Computer Science, Information Laboratory (InfoLab),
University of Southern California, Los Angeles, CA 90089-0781, USA
e-mail: jafkhosh@usc.edu

A. Khodaei
e-mail: khodaei@usc.edu

M. Sharifzadeh
e-mail: sharifza@usc.edu

C. Shahabi
e-mail: shahabi@usc.edu

window query asking for the visual representation (e.g., map) of a limited geographical neighborhood. The requested geospatial information about the queried area (e.g., road network) must be transferred to the user's machine (client) and displayed appropriately. Consisting of thousands of points and line segments representing various geographical features, this data requires a significant amount of time to be generated at the server side and is even more time-consuming to be rendered at the client side. The common practice to address the performance/storage issue in GIS applications is to send a raster image, which is a rendition of requested geospatial data at the server side, to the client. Google Maps[1] and Microsoft Live Local[2] are examples of such an approach. However, the transmitted raster image is only a visual representation of the geospatial data and hence does not include the geometric objects and corresponding metadata. Instead, the objects are rendered in the image and some of the corresponding metadata is superimposed as limited labels. Any subsequent query (e.g., nearest neighbor query) in the client results in another handshake with online server which hosts the original dataset. That is, the client application cannot manipulate the results for further processing.

A solution to this problem is sending the original query result in vector data format to the client to enable further interaction with it and to preserve the metadata. This way, server sends the vector data instead of the raster image to the client which enables users to further interact with query results (e.g., to select a specific geometric object or to issue a query based on returned results). An example of such approach is recently taken by Yahoo! Maps Beta[3] which allows users to highlight different sections of a path (i.e., different linestrings). Such level of interactivity can greatly improve user experience with the system and enable more sophisticated query analysis on the client [6,15].

A careful study of road vector databases in general and such query results in particular, however, reveals that the data returned to the client is highly repetitive in nature and shows strong correlation. Such data behavior suggests finding a way to reduce this redundancy and to apply a multi-resolution compression technique which is capable of compacting such correlated data and making it smaller based on clients' desirable level of details. However, one challenge in compressing vector data is that an ideal compression technique while being fast, must respect the spatial and topological aspects of the data. For instance, the boundary of a hexagon-shaped park must be invariant to the compression scheme used. More importantly, compressing (decompressing) the query results at server (client) while making the data smaller and the transmission time shorter, should improve both the server and the client performances to be a viable approach. While compression makes the data smaller in size, it may not improve the server or the client performance by itself.

In this paper, we propose a novel approach that integrates the application of a data compression technique (i.e., Huffman coding) and a line aggregation operator[4] on vector data within a geospatial query processing system. With our approach, we compress the entire query result that is to be sent to the client and do not remove any road segments due to its relative unimportance to the desired level of detail. Hence, our approach is orthogonal to those other approaches where only selected portions (e.g., freeways) of the original data are chosen for transmission. We, however, by adding an aggregation technique before compression, eliminate the huge amount of redundancy from the data before compressing it

[1] http://maps.google.com.

[2] http://local.live.com.

[3] http://maps.yahoo.com.

[4] By *aggregation* here we refer to the term from the database field, not to be confused with the same term used in cartography.

and therefore improve the performance. We show that this combination of aggregation and compression, reduces response time in both client and server significantly and achieves high compression ratios while preserving spatial and topological aspects of data. Obviously, our compression/aggregation scheme also reduces the amount of data transferred over the network. This said, our empirical studies show that the data transfer is not the dominant factor in response time as its duration is negligible as compared to processing time in both server and client sides (see Sect. 7). Using a road network database as our running example, our two-step approach uses spatial aggregation to group spatial (and non-spatial) attributes of related road segments together, followed by a multi-resolution compression technique which is capable of compressing/representing data up to any desired accuracy level. The ultimate result is a highly compressed data; the aggregation reduces the total number of tuples in the data and the compression decreases the number of bits by which these tuples are represented. We propose two variants of our method for different query scenarios: *cONa* offers faster client response time by performing the line aggregation operator online over the query result and highly compressing it and *cOFa* minimizes server throughput by pre-aggregating the entire database offline before the compression takes place. We argue that depending on the application, server load and query size a geospatial service can choose the appropriate variant.

A preliminary version of this work appeared in Khoshgozaran et al. [10]. This paper subsumes [10] and in addition studies the client side of the system. The focus of Khoshgozaran et al. [10] was on the server side performance and the experiments were aimed to evaluate the server performance. In this paper, we show that our proposed approaches reduce not only the server's response time and the amount of data transferred from the server but also the time it takes for the client to process and display the query results. In this paper, we also differentiate between two different types of clients namely a lightweight and a heavyweight client and study how their differences can affect the overall system performance. Finally, as a proof of concept, we implemented an end-to-end system prototype. Using real-world datasets from NAVTEQ,[5] we conducted several new experiments using this prototype system. The new client/server architecture of our prototype allows us to measure the improvements on the server side, as well as on the client side (both the lightweight and heavyweight clients). All the operators discussed in this paper have been implemented in our prototype system. We implemented our methods on top of Google Maps (as an example of heavyweight client) and our own Java application (as an example of lightweight client). Vector data (both aggregated and non-aggregated) from NAVTEQ were populated in relational tables in Oracle 10g and were stored on the server side.

Our experiments in Sect. 7 show promising results for both of our proposed methods. *cONa* achieves high compression of around 80% for almost any size of data while *cOFa* reaches up to 70% compression ratio for large enough data. Even though *cONa* and *cOFa* have an extra encryption (and decryption) module, they reduce the server response time by almost 60% of that of the naive approach on average. With the heavyweight client, both *cOFa* and *cONa* achieve more than 70% reduction on average in client response time for large enough data. Similarly, with the lightweight client, *cOFa* improves the client performance up to a factor of three while *cONa* improves it by a factor of 4.

The remainder of this paper is organized as follows. We discuss the related work in Sect. 2. Section 3 defines the preliminaries necessary for understanding our compression scheme. In Sect. 4 we study how our approach works and introduce the modules it utilizes. Sections 5 and 6 detail our query processing approach on server and client sides, respectively. Section 7

5 http://www.navteq.com

evaluates the performance of our proposed methods and finally Sect. 8 discusses the lessons learnt and future work.

2 Related work

The problem of vector data compression has been addressed by two independent communities: (1) data compression community that takes numerical approaches embedded in compression schemes to focus on the problem of compressing road segments [2,16,20]. The advantages of these methods lie in their simplicity and their ability to compress a given vector data up to a certain level. However, the drawback of using a pure data compression approach is that it ignores the important spatial characteristics of vector data inherent in their structure. A successful approach must be devised with the rendering process and user's requested level of details in mind as this is the final deliverable of the entire process. Also none of the above references study the efficiency of their approaches with regards to overall response time. (2) GIS community that uses hierarchical data structures such as trees and multi-resolution databases to represent vector data at different levels of abstraction [1,3–5,7,8,11–14,18,19]. With most of these methods, displaying data in a certain zoom level requires sending all the coarser levels. Furthermore, using generalization operators introduces the issue of choosing which objects to be displayed at each level of abstraction (although as discussed below, the approaches based on line simplification do not suffer from this issue). While both approaches have their own merits, neither of these techniques blends compression schemes with hierarchical representation of vector data. Furthermore, most of the above approaches do not perform empirical experiments with bulky real-world data to study the effect of performing such compression techniques on the client, transmission and server times. As our work bears more similarities with the second class, we discuss some of the references in more details.

One of the most well-known line generalization and simplification schemes proposed in the GIS community is the Douglas Peucker algorithm [7]. The idea behind this algorithm is to propose an effective way of generalizing a linear dataset by recursively generating long lines and discarding points that are closer than a certain distance to the lines generated. More specifically, this algorithm takes a top down approach by generating an initial simplified polyline joining the first and last polyline vertices. The vertices in between are then tested for closeness to that edge. If all vertices in between are closer than a specified tolerance, $\epsilon > 0$, to the edge, the approximation is considered satisfactory. However, if there are points located further than ϵ to the simplified polyline, the point furthest away from the polyline will be chosen to subdivide the polyline into two segments and the algorithm is recursively repeated for the two generated (shorter) polylines. Although the use of polyline simplification algorithms is orthogonal to our work, due to its relevance, a more detailed comparison between our approach and the Doughlas–Peucker based techniques is discussed next.

The Douglas–Peucker algorithm delivers the best perceptual representations of the original lines and therefore is extensively used in computer graphics and most commercial GIS systems. However, it may affect the topological consistency of the data by introducing self-intersecting simplified lines if the accepted approximation is not sufficiently fine. Several studies propose techniques to avoid the above issue known as *self-intersection property* [14, 18,19]. For instance [14] proposes two simple criteria for detecting and correcting topological inconsistencies and [18] present an alternative approach to the original Douglas-Peucker algorithm to avoid self-intersections without introducing more time complexity. Finally, based on

Saalfeld's algorithm [19] propose an improvement to detect possible self-intersections of a simplified polyline more efficiently. However, due to high cost of performing the proposed generalization in real-time [19] suggest a pre-computation of a sequence of topologically consistent representations (i.e., levels of detail) of a map which are then progressively transmitted to the client upon request. Our approach mainly differs from Zhou and Bertolotto [19] in that we do not store multiple representations of data at different levels offline and we perform the entire process of generating user's requested level of detail on the fly. Another important difference is that Zhou and Bertolotto [19] propose sending the coarsest data layer to the user initially and then progressively transmitting more detailed representations of the same data to the client. However, we only send the single desirable level of detail requested, to the user. Therefore, as the need for having more levels of detail increases, use of their proposed progressive transformation increases the amount of data being transferred redundantly. In Sect. 7 we perform extensive experiments to study the effect of different zoom levels on client/server response time and compression ratio achieved for different levels of detail and show how well the simplified data at each level of detail approximates the original data. Furthermore, as opposed to Zhou and Bertolotto [19] we do not focus extensively on topology preservation; however, it is important to note that our aggregation does not affect topology at all. Also the finest level of detail will contain the original data and thus completely preserves topology. For all other levels of detail, our visual investigation of the results show strong indication that we do preserve topology.

Another work based on the Douglas–Peucker algorithm related to this paper is the work of Buttenfield [5] in progressive vector data transmission over the web. Similar to Zhou and Bertolotto [19], Buttenfield proposes a hierarchical simplification preprocessing where each *packet* stores the representation of a complete vector at some intermediate level in a tree. Packets are constructed using the Douglas–Peucker algorithm for different levels of detail and based on the user request, the entire row of a certain height is transferred over the network. Buttenfield argues that "transmission time and disk space remain as two significant impediments to vector transmission", however, we show that transmission time accounts for a very nominal portion of the response time and the most time consuming operations occur during server processing and client rendering phases. Also with current cheap storage devices, the size of vector data is becoming a less important issue compared to the response time and processing overhead. Furthermore, as mentioned above, we tend to avoid the preprocessing required by the offline phase in storing multiple representations of the data and thus progressive transmission of refined results. Finally [5] does not report any experimental evaluation of the proposed system and it is only tested on single polyline packets contained in a small geographic database and the efficiency of this method in dealing with real-world databases is yet to be studied.

We now review the necessary preliminaries which help show how the end-to-end compression process is carried out.

3 Preliminaries

With the advent of techniques to collect geospatial data such as road networks, different vector datasets such as NAVTEQ and TIGER/Line[6] have been developed. Regardless of the accuracy and methods used to obtain such datasets, their spatial information is stored as a

[6] http://www.census.gov/geo/www/tiger.

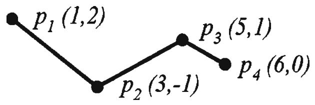


Fig. 1 A sample road segment LS_1

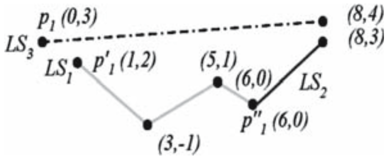


Fig. 2 Linestrings of vector database VD

linestring (to be defined later). In this section, we define terms and notations used throughout the paper and provide the background for compression technique we adopt.

3.1 Vector database

Definition 1 A *linestring* $LS = \langle p_1, \ldots, p_n \rangle$ is an ordered set of $n \geq 2$ points $p_i = (x_i, y_i)$ each representing a location at longitude x_i and latitude y_i. Figure 1 illustrates a linestring LS_1 consisting of four points representing a road segment in the city of Los Angeles.

Definition 2 A *vector entity* $VE = \langle a_1, \ldots, a_m, LS \rangle$ is a combination of one linestring attribute LS associated with $m \geq 1$ non-spatial attributes $a_1, \ldots, a_m$. We use vector entity and *entity* interchangeably throughout the paper. In Fig. 1, $VE_1 = \langle$'Vermont St', $LS_1\rangle$ is an entity including LS_1.

Definition 3 A *vector database* VD is a database storing vector entities VE. It could represent the road network of a city or the rivers of a country. An example of a vector database is $VD_1 = \langle VE_1, VE_2, VE_3 \rangle$ where $VE_1 = \langle$'Vermont St', $LS_1\rangle$, $VE_2 = \langle$'Vermont St', $LS_2\rangle$ and $VE_3 = \langle$'Slausen Ave', $LS_3\rangle$. Figure 2 depicts the corresponding linestrings.

3.2 Huffman coding

Consider a set S of words each composed of symbols from an alphabet A. Given alphabet $A = \{a_1, \ldots, a_n\}$, assume F is the set of numeric values f_i: the frequency of repetition of symbol a_i in the words of set S. The standard Huffman coding finds a function $H_{A,F}(a_i)$ that maps each symbol a_i in A to a binary code (i.e., bit string) b_i. The mapping guarantees that $\sum_k f_k \times |a_k|$ is minimized where $|a_k|$ is the size of a_k [9]. The generated mapping function is a lossless prefix-free binary encoding of A (i.e., no bit string b_i is a prefix of $b_j \neq b_i$). The more a symbol a_i is repeated in S, the shorter code b_i is assigned to a_i by Huffman coding. Hence, encoding the words in S using b_is results into an efficient compression of the original set S. Section 4 describes why this property makes Huffman coding an appropriate method for compressing a vector database.

4 Our approach

In this section, we define two operators performed on the set of vector entities of a query result. Later in Sect. 5, we describe alternative query processing schemes based on different ways these operators can be assembled together.

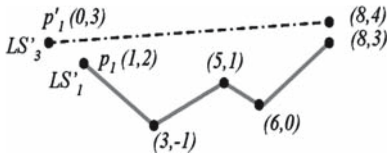


Fig. 3 Aggregation of vector database VD

4.1 Line aggregation operator *LAgg*

In a vector database VD, the information of each real-world road is stored as a set of vector entities each containing a segment of the road as a linestring attribute associated with some non-spatial attributes such as road name. Given VD a set of vector entities VE = $\langle a_1, a_2, \ldots, a_m, \text{LS}\rangle$, the Line Aggregation operator *LAgg* combines each group of VEs with a common non-spatial attribute value into one single vector entity VE′. The entity VE′ consists of a single linestring LS′ generated by concatenation of all linestrings LS of the original entities. For example, Fig. 3 illustrates the result of applying LAgg on the vector database of Fig. 2. Notice that the linestring of the generated entity consists of the same points of the original entities thus preserving the spatial aspect of the data. We use the notation of relational algebra [17] to formally define LAgg as

$$a_1,\ a_2, \ldots, a_k \varsigma f_{k+1} a_{k+1},\ f_{k+2} a_{k+2}, \ldots,\ f_m a_m,\ \text{CONCAT LS (VD)}$$

where $a_1, a_2, \ldots, a_k$ $(1 \leq k \leq m)$ are non-spatial attributes on which we group the entities. Each f_i is an aggregate function defined on non-spatial attribute a_i. CONCAT is the spatial aggregate function that returns the concatenation of a set of related linestrings as one linestring. Performing the operator *LAgg* on the vector database VD generates an aggregated vector database VD′. It consists of vector entities VE′ = $\langle a_1, a_2, \ldots, a_k, a'_{k+1}, a'_{k+2}, \ldots, a'_m$, LS′$\rangle$ where $a'_i = f_i(\langle a_i \rangle)$ and LS′ = CONCAT($\langle$LS$\rangle$) are the results of applying functions f_i on the multiset of values for attribute a_i in the group of related entities in VD and CONCAT on their corresponding linestrings LS, respectively. We apply *LAgg* on VD_1 defined in Section 3.1 to get VD'_1 = Street Name LAgg CONCAT LS (VD_1). VD'_1 will now consist of the following vector entities VE'_1 = ⟨‘Vermont St’, LS'_1⟩, VE'_3 = ⟨‘Slausen Ave’, LS'_3⟩, where LS_1 and LS_2 are concatenated to generate LS'_1 (see Fig. 3).

Notice that aggregating a real-word vector database in which each entity includes only one road segment significantly reduces the number of tuples in the database. For example, applying *LAgg* on the vector database of 250,000 entities in an area in southern California based on street names, types (e.g., road/highway) and directions generates only around 50,000 aggregated entities (about 80% reduction in number of tuples). Notice that although applying *LAgg* does not lose any spatial data in reducing number of tuples, it increases the size of each tuple. However, we will later show that both the reduction in the number of tuples and the increase in each tuple size make the data more compressible.

4.2 Compression operator *Comp*

The second operator in our scheme is a compression operator *Comp* which is used to convert the input data to a compressed format that is much smaller than the original data. *Comp* consists of four different modules to convert the input data to a compressed format sequentially. The modules illustrated in Fig. 4 are as follows:

1. The transformation module Trans Given L, a set of linestrings LS, the transformation module *Trans* computes the differential representation of each LS in L. This is an alternative

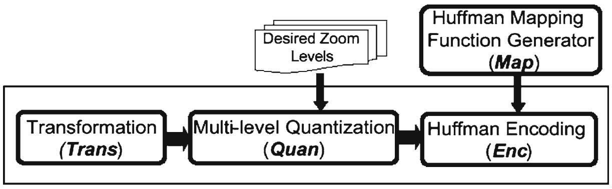

Desired Zoom Levels
Huffman Mapping Function Generator (Map)
Transformation (Trans)
Multi-level Quantization (Quan)
Huffman Encoding (Enc)

Fig. 4 The *Comp* operator

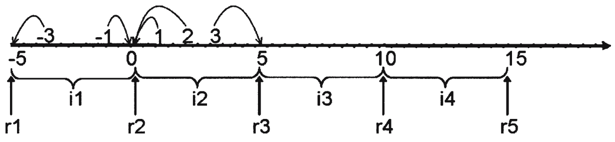


Fig. 5 Quantization of the transformed VD

way of representing a line segment. Given LS = $\langle p_1, \ldots, p_n \rangle$, the differential representation of LS is

$$Trans(LS) = \langle\langle p_1.x, \Delta x_1, \ldots, \Delta x_{n-1}\rangle, \langle p_1.y, \Delta y_1, \ldots, \Delta y_{n-1}\rangle\rangle$$

where $\Delta x_i = p_{i+1}.x - p_i.x$ and $\Delta y_i = p_{i+1}.y - p_i.y$ for $1 \leq i < n$. We refer to point p_1 as the *base point* of linestring LS. The differential representation of the linestrings illustrated in Fig. 3 is as follows:
Trans(LS'_1) = $\langle\langle 1, 2, 2, 1, 2\rangle, \langle 2, -3, 2, -1, 3\rangle\rangle$
Trans(LS'_3) = $\langle\langle 0, 8\rangle, \langle 3, 1\rangle\rangle$

In Sect. 7, we show that compressing the differential representation of real-world road segments results into high compression ratios.

2. Multi-level quantization module Quan Depending on the level of detail (LOD) desired by the application, *Quan* defines the maximum number of digits up to which the data can be rounded without losing the accuracy in its visual representation. For instance, if the user intends to view the data in an LOD where the points $p = (1.4, 2.3)$ and $p' = (1.0, 2.0)$ are displayed as a single point, p' can be used to visualize p (we round p by one digit). As a result, *Quan* increases the frequencies of the symbols used by the Huffman mapping function generator (defined later in this section) which results in high compression. Figure 6 illustrates the effect of quantization on the distribution of symbols.

Suppose that the application requires the data in several LOD's which are known a priori. For each level L_i, *Quan* processes the output of *Trans* (i.e., differential values of linestrings) to *quantize* the data with respect to L_i. This representation of the quantized data is called zoom level Z_i. The quantization process is done as follows: Based on the error tolerance for each level, *Quan* first partitions the domain of data into intervals (bins) of size $\frac{10}{K}$ units where K is a constant. Subsequently, it distributes any numeric data among these bins. That is, the number N in interval $i_q = [r_q, r_{q+1})$ will then be represented by r_q (r_{q+1}) if $|N - r_{q+1}|$ is greater (less) than $|N - r_q|$. Figure 5 shows the result of quantizing five numbers in interval $[-5, 15)$ to four bins. As seen $-3, -1, 1, 2$, and 3 are quantized to $-5, 0, 0, 0$, and 5, respectively.

Quantizing the road data for multiple LOD's enables the application to view the data in arbitrary levels of detail with a hierarchical representation. The choice of the number of required LOD's depends on the application. For each level L_i, the number of bins used by *Quan* for quantization is determined by the smoothness of transition desired from viewing the data in level L_i to the next finer/coarser level.

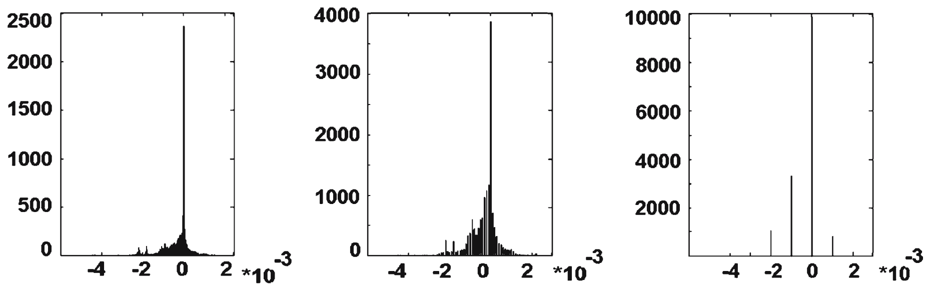


Fig. 6 Symbol distribution in the output of *Quan* for three levels of detail

Table 1 Symbol mapping for alphabet A

Symbol a_i	–	0	1	*Blank*	5	…
Bit string b_i	11	010	0001	0110	0001	…

The hierarchical nature of the output of *Quan* allows a gradual decrease in data precision as long as the error level (i.e., the displacement of points from their original locations) is tolerable to be displayed properly for that LOD. Notice that each zoom level uses the original data for quantization instead of the data from the previous levels and hence the errors will not propagate through all levels. Also in order to preserve the finest granularity of data, in level L_0 we do not quantize the data and thus perform a lossless compression of query results.

3. Huffman mapping function generator Map Our ultimate goal is to compress the query result set before sending it to the user. Discrete acquisition scheme of road vector data and its frequency suggest that the value of differences in latitude and longitude of consecutive recorded points should have high repetition and correlation (see Fig. 6). Moreover applying our transformation and quantization modules further increase this correlation. Consequently as a compression technique that significantly compresses such highly correlated data, we use Huffman coding (see Sect. 5). Having sets A, F and S (defined in Sect. 3.2), the *Map* operator first computes f_i, the frequency of occurrence of symbol a_i in S and constructs Huffman mapping function $H_{A,F}\ (a_i)$ where $F=\{f_1, \ldots, f_n\}$. We use *Dictionary* to refer to the mapping function $H_{A,F}\ ()$. In our approach, S is the set of geocoordinates of points representing linestrings in VD. As S includes only numeric values, we have $A=\{0, \ldots, 9, -, \textit{blank}, \text{eof}\}$ (in the worst case). Table 1 illustrates a part of the mapping constructed by applying *Map* on our original vector database.

The Matlab code we used[7] has a fixed small alphabet which protects us from facing the problem of gradual growth of the dictionary size which is common in the applications of Huffman coding. This is an important property in evaluating the effectiveness of our compression scheme and measuring its compression ratio. After performing the line aggregation, transformation and quantization modules on the vector database, a bulky dataset of differential values is generated. Applying the *Map* module on this data on-the-fly can sometimes take significant amount of time. Therefore, for each zoom level, in an off-line process we apply *Map* on the differential values and store the constructed dictionary once for each zoom level.

4. Huffman encoding module Enc This module performs the actual compression of the vector data. *Enc* uses the dictionary constructed by *Map* to encode the output of *Quan* and generate the compressed bit string of linestrings for each zoom level. After going through

[7] http://www.mathworks.com.

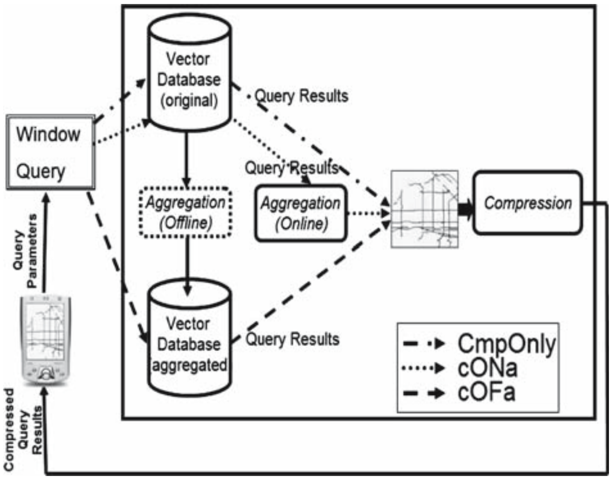


Fig. 7 Alternative query flows

phases mentioned above each original road segment of query result set (i.e., linestring LS) will be transformed into its base point (i.e., LS.p_1) followed by encoded values of its quantized latitude/longitude differences corresponding to each zoom level. For each quantized difference δ, we have $\text{Enc}(\delta) = \{H_{A,F}(a_i)|a_i \in \delta\}$ where $H_{A,F}()$ is the Huffman mapping function generated by *Map*. For example, *Enc* compresses the quantized linestrings of Fig. 5 as $\text{Enc}(\text{LS}'_1) = 0110\ldots00001$ which in combination with the base point $p_1(1,2)$ constructs the compressed data.

5 Server-side query processing

Suppose that a user issues a window query from his/her client machine. A *Naive* approach retrieves the relevant vector entities (query result) from the database and sends the *uncompressed* result back to the user. Depending on different layouts of the operators in our system, we propose two variants of our compression scheme:

Compression of off-line aggregated data (*cOFa*) This approach uses the aggregation operator *LAgg* off-line (once) to generate the aggregated vector database VD′ from the original vector database VD. It then submits the user's spatial queries to VD′ instead of VD. The retrieved query result is then fed into *Comp* whose output is a highly compressed encoded copy of the original query result which is transferred more efficiently to the client. The bottom query path in Fig. 7 shows this approach.

cOFa offers two advantages over the *Naive* approach: (1) It highly reduces the size of the data that must be transferred to the querying device. (2) It always outperforms the *Naive* approach in terms of query response time.

Compression of on-line aggregated data (*cONa*) This approach is similar to *cOFa* except that it first issues the window query against the original vector database VD. It then applies the *LAgg* operator to the query results. Finally, *cONa* sends the aggregated output to the *Comp* operator similar to *cOFa*. Notice that here the aggregation operator is performed once for every received query in contrast to *cOFa* in which the aggregation operator is performed once for the entire VD. The middle query path in Fig. 7 illustrates *cONa*. This approach

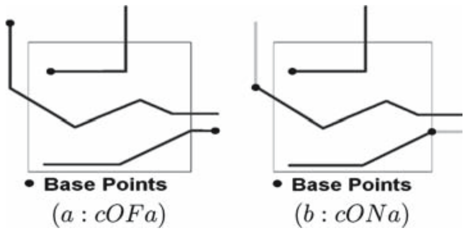


Fig. 8 Base points in *cOFa* versus *cONa*

achieves an even higher compression than that of *cOFa* while imposing an increase on the query processing time.

One important difference between *cOFa* and *cONa* is the way they treat base points. With *cOFa*, the entire vector database is aggregated offline and thus (assuming the aggregation is made on street name) each road consisting of several road segments will be represented as a single linestring with a fixed base point (i.e., the first point of its first linestring before aggregation). Therefore, while compressing the road segments that overlap with our window query, we use those fixed base points and the entire linestring representing each road as an input to our *Comp* operator. In contrast, *cONa* first finds all the road segments that overlap with the window query and performs line aggregation only on these line segments. Consequently, for each road in the query window, *cONa* treats the first point of the results' first line segment as the base point for that road. Figure 8 illustrates this difference in assigning base points to the resulting linestrings in *cOFa* and *cONa*. Although the query result retrieved in *cOFa* contains more information than what the user actually requires, its overall compression ratio is far better than the *Naive* approach and is comparable to *cONa* (see Sect. 7). This property suits the applications where the user is most likely to navigate to the adjacent areas right after his query result is displayed.

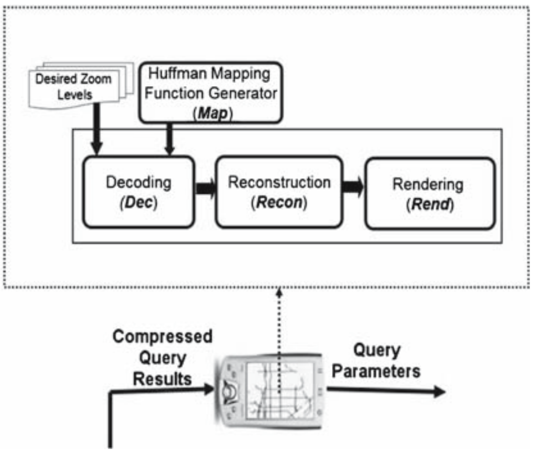
6 Client-side result processing

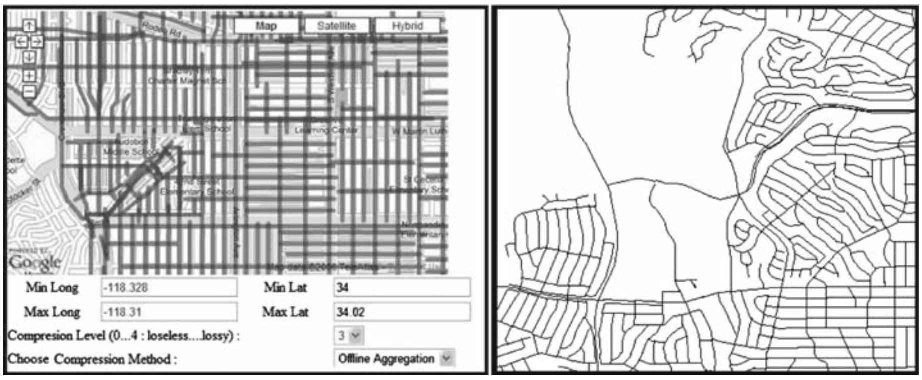
In Sect. 5 we explained how the server compresses/aggregates the query results (components inside the box in Fig. 7). This data is then ready to be sent to the client. In this section, we discuss the processing components of the client itself (outside the box in Fig. 7). A client can range from an application running on a PDA to a publicly available web-page loaded in a web browser of a PC. Figure 9 shows the enlargement of the client part of Fig. 7 including its different modules. Client in general has two main functionalities: It enables the user to specify his/her query and corresponding parameters (outward arrow from the client in Fig. 9) and more importantly processes and renders the query results (inward arrow to the client in Fig. 9) back to the user. We study three different modules on the client side that enable processing and displaying the data: Huffman decoding module *Dec*, reconstruction module *Recon* and rendering module *Rend*. The first two of these modules are created specifically for our system while the last one is generic. These modules together, convert the compressed data received from the server to a meaningful and displayable format.

Two different types of clients were used in our end-to-end system: Heavyweight client and Lightweight client. In a heavyweight or slow client, besides the resulting vector data, additional data such as raster or satellite images are sent and displayed on the client. We used Google Maps (using Google Maps API[8]) as an example of a heavyweight client. Note that

[8] http://www.google.com/apis/maps/.

Fig. 9 Client Modules





Min Long	-118.328	Min Lat	34
Max Long	-118.31	Max Lat	34.02

Compresion Level (0...4 : loseless....lossy) : 3

Choose Compression Method : Offline Aggregation

Fig. 10 Snapshots of heavyweight (*left*) versus lightweight (*right*) clients

in this case, vector data is coming from our own server while extra layers of imagery are coming from Google servers. In a lightweight or fast client, only the desired vector data and nothing more are sent and rendered on the client. As an example of lightweight client, we implemented our own java-based application (using a publicly available graphics library[9]). Figure 10 illustrates snapshots taken from our heavyweight and lightweight clients. Results of our experiments on both types of clients and their performance are discussed in Sect. 7.

6.1 Huffman decoding module *Dec*

This module performs the actual decompression of the compressed vector data. The *Dec* module is in nature the inverse of the *Enc* module. It receives from the server an encoded bit string for each linestring and the zoom level in which data was encoded as input. The zoom level is known to the client a priori, since this is the same zoom level user specifies at the beginning of his/her request to the server. Data has been encoded and quantized based on this zoom level on the server side and then returned back to the client to be decoded based on the same zoom level. *Dec* uses the same dictionary constructed by *Map* operator (for that specific zoom level) to

[9] http://www.cs.princeton.edu/introcs/15inout/.

decode the encoded bit string. *Dec* then returns an ordered list of symbols a_i as output. This list represents linestring's quantized differential values separated by blank. For example, assume that client receives the bit string '01010110010100101001010010100101001010100010' from the server (this is in fact the same bit string used in the server side processes of Sect. 5). *Dec* decompresses bit string into '0 -5 0 0 0 0 0 5' using the mapping shown in Table 1 assuming the same zoom level used by server in Sect. 3.2. The resulting numbers alternate between (quantized) longitude and latitude differences. For instance 0 is the first (quantized) longitude difference and -5 is the first (quantized) latitude difference.

6.2 Reconstruction module *Recon*

Given the output of *Dec* and a base point for each linestring LS, the reconstruction module *Recon* reconstructs a new linestring $\hat{LS}$. In other words, given a base point p_1 and differential values $\Delta x_1, \ldots, \Delta x_{n-1}$ and $\Delta y_1, \ldots, \Delta y_{n-1}$, *Recon* generates a new reconstructed linestring $\hat{LS} = \langle \hat{p_1}, \ldots, \hat{p_n} \rangle$ where $\hat{p}_{i+1}.x = \hat{p}_i.x + \Delta x_i$ and $\hat{p}_{i+1}.y = \hat{p}_i.y + \Delta y_i$ for $1 \leq$ i $<$ n. We denote the new linestring by $\hat{LS}$ (and its reconstructed points by $\hat{p}_i$) to emphasize that these values are reconstructed values and might not be identical to the original values. Since during compression, data is quantized after transformation, there will be loss of accuracy after constructing the linestring. In Sect. 7, we show that although average displacement error in reconstructed data increases with zoom level, the error is acceptable for all zoom levels.

Considering the above example, *Recon* gets a sequence of differential values '0 -5 0 0 0 0 0 5' and a base point $p_1 = (1,2)$ as input. Based on this input, *Recon* computes a new linestring $\hat{LS} = \langle (1, 2), (1, -3), (1, -3), (1, -3), (1, 2) \rangle$. Similar to *Dec* which is the inverse/opposite module of *Enc*, *Recon* is the inverse/opposite module of *Trans*. While *Trans* transforms a linestring to its base point and differential values, *Recon* generates the same linestring given corresponding base point and differential values.

6.3 Rendering module *Rend*

This module implements the last phase of the entire process. It displays the final result of user's window query. *Rend* is application-dependent, meaning that its implementation strongly depends on the visualization interface used. As we already mentioned, for our application, we have used two different types of clients for displaying query results. Depending on the client type, performance of the *Rend* module and consequently the entire client-side performance is affected (see Sect. 7).

In general, *Rend* in abstract level is a black box that receives the required data from *Recon* module and renders that information on a suitable graphical interface. Notice that this module takes the longest time among all the modules since it requires rendering all the qualified geometries on the screen (point by point). The amount of time *Rend* spends is proportional to the size of data it receives (i.e., number of points). In Sect. 7, we show that our proposed methods speed up this module because they eliminate the redundant points by aggregating the data. Notice that all the data rendering processes/preparations are performed on the client side not on the server side. That is, instead of having the server process the results and create a raster image and then send the image back to the client, client itself takes care of the rendering part. This way, instead of a raster image, client has access to the data itself for possible further query and analysis. For instance, the client can select a line segment and query its length.

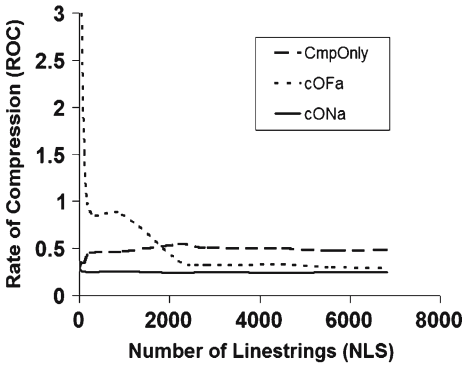


Fig. 11 Rate of compression versus number of linestrings

7 Performance evaluation

We conducted several experiments to evaluate the performance of our proposed approaches. The effectiveness of *cOFa* and *cONa* is determined in terms of (1) total query time which is a combination of the time it takes the server to prepare the query results, the time to transfer the results to the client and the time it takes the client to display the results, (2) size of the encoded query result, and (3) the displacement error resulting from lossy compression at different zoom levels. We also compared our proposed compression schemes with commonly used text compression utilities such as gzip.

In the following sets of experiments, we compared *cOFa* and *cONa* from different perspectives against two other methods: (1) *Naive* approach which simply sends the query result to the client and (2) *CmpOnly* approach (the top query path in Fig. 7) which only compresses the query result using our *Comp* operator without applying aggregation operator (*LAgg*). Our experiments are performed on a real-world dataset obtained from NAVTEQ covering a 70 mile by 100 mile area in southern California containing more than 250,000 road segments. In our experiments, we randomly generated window queries with different sizes (from 1×0.8 square mile up to 7×5.6 square mile) and compressed the results for five different zoom levels Z_0 (i.e., finest, lossless) to Z_4 (coarsest). Results are reported as the average number of linestrings returned in each query window (i.e., number of non-aggregated linestrings satisfying the query). We call this parameter NLS (*Number of LineStrings*). Experiments on the client side were run on an Intel 2.26 GHz with 2 GB of RAM while experiments on the server were run on AMD 2.19 GHz with 4 GB of RAM hosting Oracle 10g with spatial extension. Our first set of experiments focuses on the effect of increasing NLS on the size of resulting compressed data. We define *Rate Of Compression* (ROC) as the average size of the data compressed by our schemes over the size of original data being sent by the *Naive* approach. Figure 11 shows the ROC as NLS increases from 1 to 7,000 for zoom level Z_1. As seen in this figure, for all cases when the query results are not very small (NLS more than 2000), *cOFa* compresses the original data well and it outperforms the *Naive* approach. It also shows that *cONa* almost always achieves about 80% reduction in size. Our experiments show that the data behaves similarly for all other zoom levels. As the figure illustrates, for a very small NLS, *cOFa* transmits even more data than the original size of the data.[10] The reason is that the amount of excessive data being compressed is relatively large for very small window sizes/NLS (see Fig. 8). However, as the window size/NLS grows, *cOFa* reduces size very

[10] Note that for displaying purposes we have truncated the y-axis of Fig. 11 from 7 to 3. ROC of *cOFa* approaches 7 for NLS very close to zero.

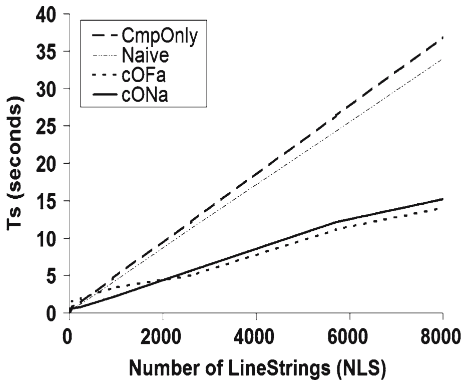


Fig. 12 Server time versus number of linestrings

fast and approaches *cONa* for the larger areas being queried. Also the figure shows that all three methods achieve better ROC as NLS grows.

The second set of experiments focuses on the effectiveness of each approach in terms of performance (i.e., time). The total query time, T consists of three times: server response time T_s, transmission time T_t and client time T_c. For each window query, the first time value, T_s is the time it takes the server to prepare the data based on the client's request. T_s itself consists of three times based on the type of operations performed. Server's first time value, t_{sq}, is the time it takes to issue the window query against our vector database VD and retrieve the results. If line aggregation is required, it is accounted for in t_{sq}. The server's second time, t_{st}, is the time that the *Trans* and *Quan* modules spend to compute the quantized differential values and finally t_{se} is the time it takes for *Enc* operator to encode its input using the dictionary. The second major time value, T_t, is the time it takes to transfer data generated by the server to the client. Based on our experiments, this time is far less than total query time even for very large query windows. To investigate this further, we performed our experiments on both a high-speed and a low-speed network connection. With a high-speed network connection (10 Mbp), the average ratio of T_t to T was around 0.2% while with a low-speed connection (56 Kbp) this ratio was about 4%. Therefore, in our evaluations we do not take this time into consideration. An interesting observation, however, is that in contrast to a common belief of compressing vector data to improve the transmission time and decrease the bandwidth requirements, we conclude that the transmission time is not a bottleneck in such approaches and thus any improvement in the client and server sides would have a more dominant effect on the overall system performance. The third major time value, T_c, is the time that client spends to prepare and display the data received from the server. For heavyweight clients, T_c is the most significant time of the entire process while in lightweight clients T_c and T_s are comparable to one another. T_c itself has three components: t_{cd}, t_{cc} and t_{cr} which are the times the *Dec*, *Recon* and *Rend* modules spend to decode, reconstruct and render the data, respectively. To summarize, T is calculated as follows: $T = T_s(= t_{sq} + t_{st} + t_{se}) + T_t(\simeq 0) + T_c(= t_{cd} + t_{cc} + t_{cr})$.

During the experiments we measured T_s, T_t and T_c for each query window. As we mentioned, T_t is negligible for all methods and for different NLS values. Figure 12 shows the server response time (T_s) versus NLS. This figure shows that both *cONa* and *cOFa* outperform the *CmpOnly* and *Naive* approaches for almost every NLS in server side. Only for very small NLS values, *cOFa* spends more time in server in comparison with *CmpOnly* and *Naive* approaches. The reason is that for very small NLS values, *cOFa* retrieves relatively large amount of excessive data and consequently spends more time compressing this excessive data. One interesting observation in Fig. 12 is the fact that *cOFa* outperforms *cONa* for larger

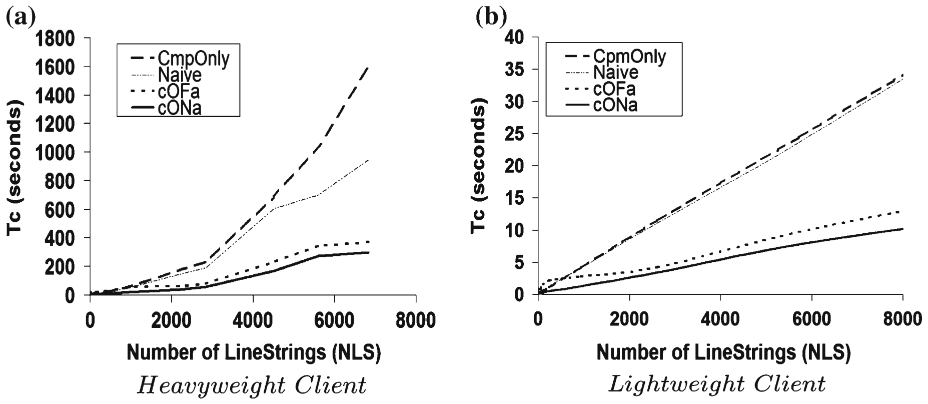


Fig. 13 Client time versus number of linestrings

NLS values (NLS > 2000). The reason is that t_{sq} for *cONa* increases with query window size because the *LAgg* operator's implementation in Oracle (the DBMS used in our experiments) takes a significant amount of time to concatenate related linestrings for bulky vector databases. However, notice that even in the worst case, *cONa* significantly outperforms *Naive* approach. Both *cOFa* and *cONa* take less transformation/quantization time t_{st} compared to *CmpOnly* approach because they operate on less number of linestrings (and hence less number of base points).

Figure 13a and b show the client time versus NLS for our heavyweight and lightweight clients, respectively. Since the performance of client side is highly affected by client type (heavyweight versus lightweight), results are shown for both client types. We first focus on the common trend in both clients. As it is depicted in both figures, both *cONa* and *cOFa* always significantly outperform *CmpOnly* and *Naive* approaches. Since *cONa* and *cOFa* remove many repetitive points by aggregating linestrings, they both spend less time in each of their client modules in comparison with *CmpOnly* and *Naive* approaches. Between *cONa* and *cOFa*, *cONa* always outperforms *cOFa* since *cONa* has less data to decode and display than *cOFa*. Now, we discuss the issue of performance difference between the two clients. As it is shown, the amount of time a heavyweight client spends is far more than the amount of time that a lightweight client spends for the same NLS value (compare the Y-axis of Fig. 13a and b). As we discussed earlier in Section 6, this is due to the large amount of extra information received and displayed in combination to the vector data by the heavyweight client while lightweight client only retrieves and displays resulting vector data. Also, by comparing client time in Fig. 13a with the server time in Fig. 12, it is clearly seen that a heavyweight client spends more time than the server for the same query and hence the total query response time T is dominated by T_c for a heavyweight client. The reason is that in a heavyweight client, rendering of the results takes a significant amount of time. Note that although the total query time is dominated by the client time in heavyweight clients, it is obviously still useful to have reduction in server time. In most applications, the server is shared among thousands or even millions of clients and hence improving server throughput is a very important factor. In these applications, even a small reduction in server response time will improve the server throughput significantly.

Figure 14a and b depict how much of the total time T, is spent at client and server sides for each of the four approaches and for three different ranges of NLS values with both the lightweight and heavyweight clients, respectively. While the total time is considerably

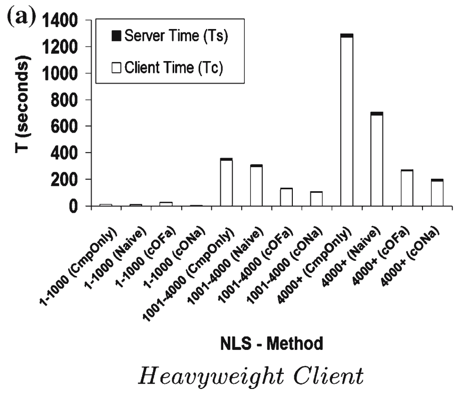
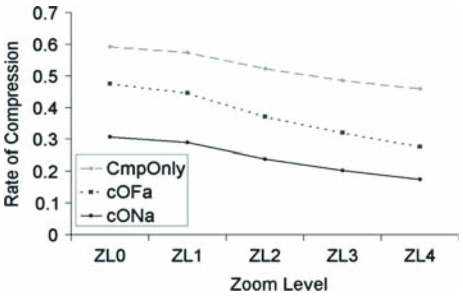


Fig. 14 Total time

Fig. 15 Rate of compression versus zoom level



dominated by the client time with our heavyweight client (Fig. 14a), it is almost distributed evenly between client and server with the lightweight client (Fig. 14b). In both diagrams, *cONa* and *cOFa* always outperform *CmpOnly* and *Naive* in total time for reasonably large values of NLS.

The third set of experiments measures the effect of different zoom levels Z_i on the rate of compression (ROC). As shown in Fig. 15, *cONa* achieves the best rate of compression. The reason is that *cONa* requires to compress less number of linestrings (compared to *CmpOnly*) and each linestring has also less excessive data (compared to *cOFa*). For example, *cONa* achieves more than 80% reduction in data size (ROC of 0.2) for Z_4.

As we discussed before, the key advantage of using *cONa* and *cOFa* compared to typical text compression utilities such as gzip lies in their use of geometrical redundancies in query results to achieve better compression. As our fourth set of experiments, we empirically evaluated the effectiveness of our proposed schemes against well-known text compression utilities such as gzip in terms of processing time and reduction in the result set size. Figure 16 shows the result of comparing the time it takes to encode the same set of random window queries used in previous experiments (with NLS varying from 1 to 7,000) with *cONa*, *cOFa* and gzip. Note that the first two components of the server time (i.e, t_{sq} and t_{st}) are always needed to generate the query result for any of the three encoding variants *cONa*, *cOFa* and gzip. Therefore, we only compare gzip time with t_{se}, the time it takes for the server to encode the query results. It is clear that both *cOFa* and *cONa* always outperform gzip in terms of the time it takes to compress the same query result. The relatively small values of time depicted in Fig. 16 leads to another observation which is the fact that the server query time t_{sq} and

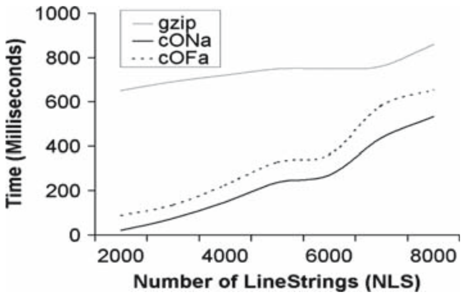


Fig. 16 *cONa* and *cOFa*'s t_{se} versus gzip

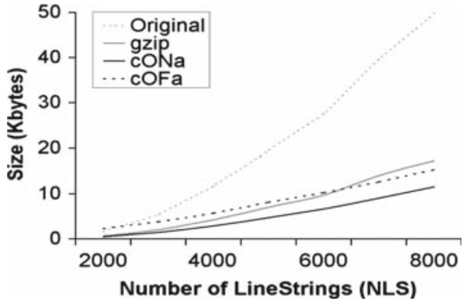


Fig. 17 *cONa* and *cOFa* size versus gzip

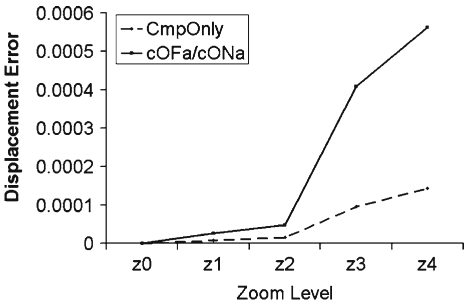


Fig. 18 Displacement error versus zoom level

server transformation/quantization time t_{st}, significantly dominate t_{se}, the time it takes for the server to encode query results. Finally, as we previously saw in Fig. 11, *cONa* produces a smaller query result compared to *cOFa* and therefore, spends less time encoding it.

We also compared the reduction in query result set size achieved while using our proposed schemes and compared it with the reduction obtained by compressing the same result set with gzip. As Fig. 17 illustrates, while *cONa* always outperforms gzip in reducing the size of the original query result, *cOFa* achieves a better reduction compared to gzip for relatively larger NLS values.

The fifth set of experiments shows the effectiveness of our scheme by measuring the displacement error as zoom level increases. As Fig. 18 illustrates, the average error resulted because of quantization is 0.00056 for the coarsest zoom level corresponding to a 0.04 mile displacement which is insignificant for this zoom level. It also shows that the displacement error in both *cOFa* and *cONa* is more than *CmpOnly*. This is because the error propagates into longer sequence of differential values after applying *LAgg* on query results.

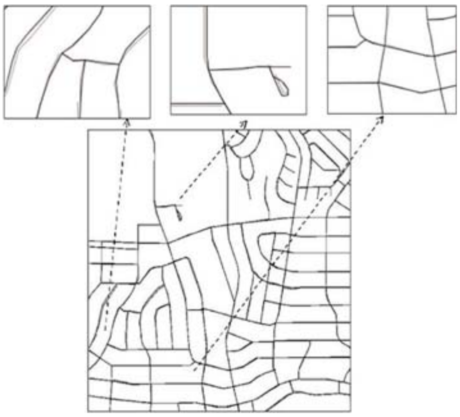


Fig. 19 Reconstructed data

With our final experiment, we compressed and then reconstructed an area in the city of Los Angeles at zoom level Z_3. This area contains different types of roads (major/minor, straight/with curves). Figure 19 shows both the original data and the reconstructed data.

8 Conclusion and future work

Querying vector data and in particular road network data is becoming more and more popular in many application domains. In most common approaches, vector data server creates a raster image of the query results and sends it back to the client. Some other approaches use general data compression schemes or hierarchical representation of vector data at different levels of abstraction. All of these approaches have their own pros and cons. In this paper, we proposed a hybrid aggregation and compression scheme which easily integrates with a spatial query processing system and compresses vector data significantly and fast. In short, our main observations were:

- our proposed methods *cOFa* and *cONa* achieve high compression ratio
- they both reduce client response time (while *cONa* outperforms *cOFa*)
- they both reduce server response time (while *cOFa* outperforms *cONa*) and hence improve server throughput
- both *cONa* and *cOFa* perform the compression in multiple levels of detail
- transmission time is negligible even for a very large number of linestrings and low communication bandwidth

In this paper we also showed the details of our end-to-end prototype and all its building blocks (both server-side and client-side modules). Our experiments verified that depending on client time, server time, and storage requirements, variants of our method can be used to guarantee high compression ratio or fast response time. We plan to address the problem of compressing other types of geometries such as polygons. We also aim at enhancing our approach to dynamically choose between the two variants at the query time. Finally, as part of our future work we also plan to compare the overall efficiency of our approach with several algorithms proposed in the class of progressive transmission schemes that are based on polyline simplification and the Douglas–Peucker algorithm.

Acknowledgments We thank anonymous reviewers for their very useful comments and suggestions. This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0238560 (PECASE), IIS-0324955

(ITR), and unrestricted cash gifts from Google and Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Ai T, Li Z, Liu Y (2003) Progressive transmission of vector data based on changes accumulation model. SDH, Leicester, Springer, Berlin, pp 85–96
2. Akimov A, Kolesnikov A, Fränti P (2004) Reference line approach for vector data compression. In: ICIP, pp 1891–1894
3. Bertolotto M, Egenhofer MJ (2001) Progressive transmission of vector map data over the world wide web. Geoinformatica 5(4):345–373 URL http://citeseer.ist.psu.edu/bertolotto01progressive.html
4. Bertolotto M, Zhou M (2007) Efficient line simplification for web-mapping. International journal of web engineering and technology special issue on web and wireless. GIS 3(2):139–156
5. Buttenfield B (2002) Transmitting vector geospatial data across the internet. In: GIScience '02: proceedings of the 2nd international conference on geographic information science, London, UK, Springer, Heidelberg, pp 51–64. ISBN 3-540-44253-7
6. Cai Y, Stumpf R, Wynne T, Tomlinson M, Chung DSH, Boutonnier X, Ihmig M, Franco R, Bauernfeind N (2007) Visual transformation for interactive spatiotemporal data mining. Knowl Inf Syst 13(2):119–142 ISSN 0219-1377. doi:10.1007/s10115-007-0075-5
7. Douglas DH, Peucker TK (1973) Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Can Cartogr 10(2):112–122
8. Han Q, Bertolotto M (2004) A multi-level data structure for vector maps. In: GIS '04: proceedings of the 12th annual ACM international workshop on geographic information systems, New York, NY, USA, ACM Press, pp 214–221 ISBN 1-58113-979-9. doi:10.1145/1032222.1032254
9. Huffman DA (1952) A method for the construction of minimum redundancy codes. Proc Inst Radio Eng 40(9):1098–1101
10. Khoshgozaran A, Khodaei A, Sharifzadeh M, Shahabi C (2006) A multi-resolution compression scheme for efficient window queries over road network databases. In: ICDM workshops. IEEE Computer Society, pp 355–360. ISBN 0-7695-2702-7.
11. Paiva AC, da Silva ED, Leite FL Jr, de Souza Baptista C (2004) A multiresolution approach for internet gis applications. In: DEXA Workshops, IEEE Computer Society, pp 809–813 ISBN 0-7695-2195-9
12. Persson J (2004) Streaming of compressed multi-resolution geographic vector data. Geoinformatics, Sweden
13. Puppo E, Dettori G (1995) Towards a formal model for multi-resolution spatial maps. In: Egenhofer MJ, Herring JR (eds) SSD, volume 951 of Lecture Notes in Computer Science, Springer, Heidelberg, pp 152–169. ISBN 3-540-60159-7
14. Saalfeld A (1999) Topologically consistent line simplification with the douglas-peucker algorithm. Cartogr Geogr Inf Sci 26(1):7–17
15. Shahabi C, Kolahdouzan MR, Safar M (2004) Alternative strategies for performing spatial joins on web sources. Knowl Inf Syst 6(3):290–314. ISSN 0219-1377. doi:10.1007/s10115-003-0104-y
16. Shekhar S, Huang Y, Djugash J, Zhou C (2002) Vector map compression: a clustering approach. In: Voisard A, Chen SC (eds) ACM-GIS, ACM, pp 74–80 ISBN 1-58113-591-2
17. Silberschatz A, Korth HF, Sudarshan S (1998) Database system concepts, 5th edn. McGraw Hill, New york. ISBN 0-07-295886-3
18. Wu ST, Márquez MRG (2003) A non-self-intersection douglas-peucker algorithm. In: SIBGRAPI, IEEE Computer Society, pp 60–66. ISBN 0-7695-2032-4
19. Zhou M, Bertolotto M (2005) Efficiently generating multiple representations for web mapping. In: Li K-J, Vangenot C (eds) W2GIS, volume 3833 of Lecture Notes in Computer Science, Springer, Heidelberg, pp 54–65. ISBN 3-540-30848-2
20. Zhu Q, Yao X, Huang D, Zhang Y (2002) An efficient data management approach for large cybercity gis. ISPRS

Author Biographies

Ali Khoshgozaran received a B.S. degree in computer engineering from Sharif University of Technology, Tehran, Iran, in 2003 and an M.S. degree in computer science from The George Washington University, Washington D.C., in 2005. He is currently working towards his Ph.D. degree in computer science at the University of Southern California where he is a research assistant working on geo-spatial databases, location-based services and location privacy at the Integrated Media Systems Center (IMSC)—Information Laboratory at the Computer Science Department of the University of Southern California.

Ali Khodaei received his B.S. degree in computer engineering from Iran National University (Shahid Beheshti University), Tehran, Iran, in 2003 and his M.S. degree in information and computer science from University of California, Irvine, in 2006. He worked as a reseracher/deveoper at the Integrated Media Systems Center (IMSC)—Information Laboratory at the Computer Science Department of the University of Southern California from 2006 to 2007. His current research interests include Geospatial Data Analysis, Spatio-temporal Databases and Geospatial Information Integration. He is currently working as a database administrator/developer at VESystems.

Mehdi Sharifzadeh received his Ph.D. degree in computer science from the University of Southern California in 2007. He is currently a senior engineer at Google. During his Ph.D. studies, Mehdi was a research assistant working on spatial databases at the Information Laboratory (InfoLab) of the University of Southern California. His research interests include spatial databases, data stream processing, computational geometry, and data mining.

Cyrus Shahabi is currently an Associate Professor and the Director of the Information Laboratory (InfoLAB) at the Computer Science Department of the University of Southern California. He received his B.S. in Computer Engineering from Sharif University of Technology in 1989 and then his M.S. and Ph.D. degrees in Computer Science from the University of Southern California in 1993 and 1996, respectively. He has two books and more than hundred articles and conference papers in the areas of databases, GIS and multimedia. He is currently an associate editor of the IEEE Transactions on Parallel and Distributed Systems. He is also a member of the steering committee of IEEE NetDB and the general co-chair of ACM-GIS 2007 and 2008. He serves on many conference program committees such as VLDB 2008, ICDE 2008 and SIGKDD 2007. Dr. Shahabi is the recipient of the 2003 Presidential Early Career Awards for Scientists and Engineers (PECASE).