QUERY SUPPORT FOR GMZ

Ayush Khandelwal [a], Dr. K. S. Rajan [a]
[a] Lab for Spatial Informatics, International Institute of Information Technology, Hyderabad, India –
ayush.khandelwal@research.iiit.ac.in, rajan@iiit.ac.in

Commission IV, WG IV/4

KEY WORDS: WFS, Lossless compression, R-tree, Partial decompression, Range query

ABSTRACT:

Generic text-based compression models are simple and fast but there are two issues that needs to be addressed. They cannot leverage the structure that exists in data to achieve better compression and there is an unnecessary decompression step before the user can actually use the data. To address these issues, we came up with GMZ, a lossless compression model aimed at achieving high compression ratios. The decision to design GMZ (Khandelwal and Rajan, 2017) exclusively for GML's Simple Features Profile (SFP) seems fair because of the high use of SFP in WFS and that it facilitates high optimisation of the compression model. This is an extension of our work on GMZ. In a typical server-client model such as Web Feature Service, the server is the primary creator and provider of GML, and therefore, requires compression and query capabilities. On the other hand, the client is the primary consumer of GML, and therefore, requires decompression and visualisation capabilities. In the first part of our work, we demonstrated compression using a python script that can be plugged in a server architecture, and decompression and visualisation in a web browser using a Firefox add-on. The focus of this work is to develop the already existing tools to provide query capability to server. Our model provides the ability to decompress individual features in isolation, which is an essential requirement for realising query in compressed state. We construct an R-Tree index for spatial data and a custom index for non-spatial data and store these in a separate index file to prevent altering the compression model. This facilitates independent use of compressed GMZ file where index can be constructed when required. The focus of this work is the bounding-box or range query commonly used in webGIS with provision for other spatial and non-spatial queries. The decrement in compression ratios due to the new index file is in the range of 1-3 percent which is trivial considering the benefits of querying in compressed state. With around 75% average compression of the original data, query support in compressed state and decompression support in the browser, GMZ can be a good alternative to GML for WFS-like services.

1. INTRODUCTION

1.1 Motivation

With the increasing number of Internet users, the demand for WebGIS is on a rise. The most popular WebGIS service Web Mapping Service (WMS) has been in place since 1999. It serves information by converting vector data into raster image tiles and then sending these tiles that can be rendered in a browser (The Geospatial Web, 2007). The issue with raster images is that the resolution of images changes due to pan and zoom. Web feature service (WFS) came later with the promise of improving WebGIS and providing transactional WebGIS. It serves information as vector data itself using any of the vector data formats such as GML (standard), JSON, KML, etc. It also lets the client edit data and send it back to the server. Vector solves the resolution issue as vector data can be zoomed and panned without change in resolution. But vector data takes up a lot of space compared to raster data for the same scene.

Compression seemed like an obvious choice to solve this issue and therefore, we came up with GMZ. But the issue with compressed data is that it cannot be used directly. These additional steps of compression and decompression everytime we need to use the data increase unnecessary overheads. Though the compression ratios are high, the compression and decompression times of GMZ are not even close to that of simple text based compressors such as zip. Query support is therefore, a major improvement in usability GMZ. Given that GML is widely used in WFS, we have designed our query subsystem to target queries that are widely used in WFS. WFS supports the bounding-box query and simple attribute queries. We have used R-tree for building spatial index from compressed data. The main feature of our query subsystem is its ability to perform partial decompression. When querying, this enables us to directly go to the target data and decompress it without touching the rest of the file. Another issue faced when reading indexed data from disk is the number of disk I/Os. Random access is slow because of the way disks are designed. We have made concrete efforts to reduce random disk accesses and increase sequential disk accesses wherever possible.

1.2 Literature Survey

Due to the verbose nature of XML, XML compressors such as XMill (Liefke and Suciu, 2000) and XMLPPM (Cheney, 2011) were developed. Later came compressors with query support such as XGrind (Tolani and Haritsa, 2002), XPRESS (Min et al, 2003) and XQzip (Cheng and Ng, 2004) (Sakr, 2009). These generic XML compressors performed worse when they were used for compressing GML. Also, there was no support for spatial queries. Initial GML compressors such as GPress (Guan and Zhou, 2007) and others (LI et al, 2008; Weiand Guan, 2010) borrowed heavily from XML compressors and added separate coordinate compressors. All these models are based on the principle of separating data and structure and compressing them independently using specialized data compressors. The compression ratios weren't good enough even though these were lossy compressors favouring delta encoding for floating-point compression. GTree (Harshita and Rajan, 2010) was a lossless compression technique that replaced delta encoding with incremental encoding by treating coordinates as strings. It achieved some of the best compression ratios but was restricted to work with polygons only. GQComp (Dai et al, 2009) was the

first GML compressor to add spatial and non-spatial query support using an R*-tree index and a feature structure tree. GQComp employs a lossless custom encoding for coordinate compression. It achieves good compression ratios but lags behind in attribute query performance. There is no way to compare its spatial query performance.

The matrices in play for establishing a comparison are compression ratio, compression and decompression times, and spatial and non-spatial query performance. Compression ratios of GMZ are better than any other GML compression model. GMZ lags behind in compression and decompression times. We expect decent query performance for spatial and non-spatial queries.

1.3 Dataset

We are using the same dataset that we used when we developed GMZ. Due to the unavailability of compiled SFP compliant GML 3 datasets, it has largely been prepared by making GML files SFP compliant or by converting shapefiles into SFP compliant GML files. QGIS has been used for the conversion process. We have prepared GML files for 2 countries – India and USA. The India files were downloaded from mapcruzin.com, a provider of region wise shapefiles, and then converted to GML. The USA GML files were downloaded from data.gov, the data portal of the government of US, and then made SFP compliant. The dataset is combination of point, line and polygon GML files. The file size ranges from 20 MB to around 1 GB with most files under 100 MB.

2. MAIN BODY

2.1 Overview of the compression model

Our compression model is based on the idea of separating spatial (coordinates) and non-spatial (attributes and structure) data and applying targeted compression on them. We create a total of 4 data containers:

1. Attribute container
2. X-coordinate container
3. Y-coordinate container
4. Index container

The attribute container stores attributes as well as GML structure. We traverse the feature tree and replace each tag by a pre-defined integer encoding. We get a list of integers representing the structure of a feature that we convert into a string. This string acts as another attribute of the feature.

The coordinate containers go through 3 pre-processing steps – duplicate removal, sorting and zero-padding. Coordinates are often repeated among features due to shared boundaries in polygons, shared ends points in linestrings or in general. Duplicate removal facilitates storage of a single copy of each coordinate. Sorting is done on coordinates to make sure we get minimum difference (delta) between subsequent coordinates. But we use a custom delta encoding instead of simple floating-point delta compression. This prevents loss of data and gives better compression per coordinate.

The pre-processing on coordinates alters the original order in which they occur in features. To reconstruct this order, we create indices that act as placeholders of the original coordinates. These indices are basically positions of coordinates in their container. Due to the high value that these indices can take, we store just the delta of these index values. The value of this delta tends to be small because adjacent coordinates in a feature tend to be close in position in the coordinate containers.

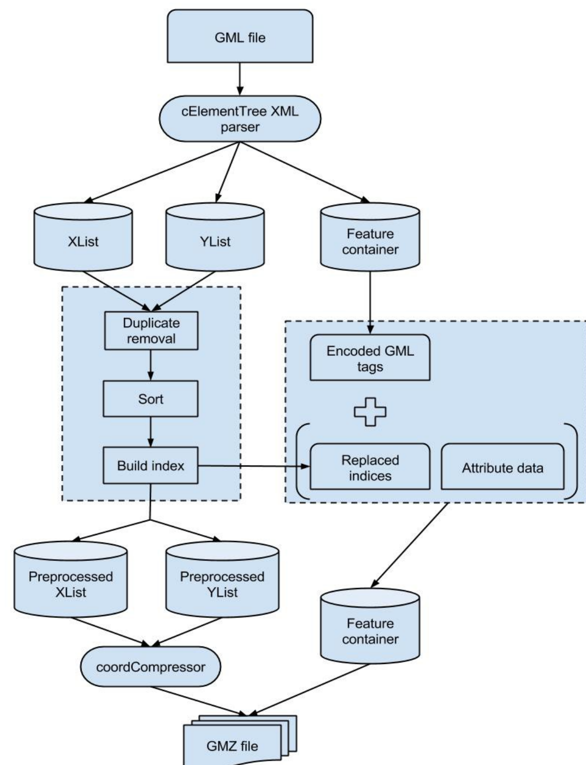

Figure 1. Compression model pipeline

2.2 Partial decompression

The ability to partially decompress individual features is the basis of our query subsystem. To ensure partial decompression, we have to provide random access to all the data containers so that data can be extracted from each container without decompressing the entire container. The smallest unit that we would ideally like to decompress is a feature. Index container inherently provides random access to a feature's coordinate indices. The modifications made to the attribute container and the coordinate containers have been discussed below.

The attribute container has been revamped to come up with the concept of attribute row. This row is loosely similar to a row in a database. The attributes of a feature are stored as a string separated by a delimiter as its attribute row. The structure encoding of the feature is being treated as another attribute of the feature and stored along with the other attributes. Though it does not provide random access to individual attributes of a feature, which is important for non-spatial queries, and demotes specialized compression on attributes by simply zipping these strings, the overall query efficiency will increase when dealing with large datasets.

Our custom encoding for coordinates does not allow random access to individual coordinates in the coordinate containers because each coordinate shares partial information about itself with its preceding coordinate. Therefore, technically a coordinate stored in the middle of a container cannot be accessed without decompressing all preceding coordinates in the container. But we don't need random access to each coordinate for any application. The smallest unit that we would

ideally like to decompress is a feature. Coordinates are stored in sorted order in containers so if the minimum and maximum X and Y coordinates or bounding-box coordinates of a feature are known, we are sure that all other coordinates of that feature will lie between this minimum and maximum. Therefore, all we need is random access to the bounding-box coordinates of a feature to decompress all the coordinates of that feature.

2.3 Optimizing disk I/Os

Random access to individual features is the first step towards achieving a query subsystem. But pure random access creates the issue of multiplying disk I/Os as the number of features to be decompressed increases. Decompression of one feature requires 4 random disk I/Os, one for each container. Similarly, 2 features would require 8, 3 would require 12 and N features would require 4N random disk I/Os. Thus, number of random disk I/Os is a function of number of features to be decompressed. There is a need to minimize the number of random disk I/Os and maximize the number of sequential disk I/Os for optimal performance. To increase sequential disk I/Os, we need our data to be sequential according the query results. The queries that we are worried about are range query, nearest neighbour query and simple non-spatial attribute query. There is no way to arrange our data sequentially for all attribute queries. But range query and nearest neighbour query results are features that are nearby geospatially. This enables us to add good amount of sequentiality to our data for range queries and nearest neighbour queries. Therefore, we have made range query and nearest neighbour query our primary queries and we have designed our query subsystem with the aim of optimizing these queries.

We have added enough sequentiality already to our coordinate containers by sorting them. This sequentiality is also relevant to our target queries as sorting brings coordinates of nearby features closer in the coordinate containers. To understand how this sequentiality helps in reducing the number of random disk I/Os, assume that a range query output is 4 nearby features. If we were to extract coordinates of these features individually using their bounding-box coordinates, we will need to 2 random disk I/Os per feature and therefore, 8 random disk I/Os. Alternately, we can merge the bounding-box coordinates simply by finding the minimum and maximum X and Y coordinates of the bounding-box coordinates. Extracting all coordinates between this new minimum and maximum ensures that the coordinates of output features will definitely lie within these coordinates. Though we are decompressing some coordinates of other features not in our output set, we are making number of random disk I/Os independent of the number of features to be decompressed. Be it any number of features, we will always need 2 random disk I/Os for decompressing all coordinates of that feature.

Unlike coordinate containers, index and attribute containers do not have any sequentiality to assist efficient nearby feature decompression. We can employ various pre-sorting methods to add significant sequentiality to index data of geospatially nearby features. A simple method would be to sort these containers based on the minimum X-coordinates or Y-coordinates of the features. Adding such sequentiality to attribute container will have a minor effect on performance because compared to the amount of data in coordinate containers and index container, attribute container is very small. Infact, decompressing the entire attribute container for each query will not affect query performance to a lot of extent. Moreover, this will add convenience for executing attribute queries, as the attribute rows will be in memory. We are also using the attribute rows for storing other crucial data about features. In addition to feature ID and attributes, we are storing 2 other feature relevant things:

1. Start byte position of a feature's coordinate indices stored in the index container.
2. Byte positions of a feature's bounding-box coordinates.

A thing to note is that this added sequentiality helps only in optimization of primary queries. Secondary queries will still perform poorly because there is essentially no way to add sequentiality in such a way that it improves efficiency for each attribute query.

2.4 R-tree indexing and querying

R-tree is a space partitioning tree data structure widely used for indexing spatial data. It is designed to be serializable on disk. They key idea of the data structure is to group nearby objects and represent them with their minimum-bounding rectangle (MBR). The R-tree is a balanced search tree that organizes data into pages; each page can have a variable number of entries. Each entry within a non-leaf node stores two pieces of data: a way of identifying a child node, and the bounding box of all entries within this child node. Leaf nodes store the data required for each child, often a point or bounding box representing the child and an external identifier for the child. Construction of R-tree is a huge optimization problem which is beyond the scope of this paper. In a classic R-tree, objects are inserted into the subtree that needs the least enlargement. When the node starts overflowing, it is broken down into 2 nodes again by minimizing the area. Better heuristics have arrived to help optimize issues associated with tree construction. Bulk-loading is one such optimization that we have used in our work. Geometries are sorted by their X-coordinate or Hilbert distance and then split into pages of desired sizes.

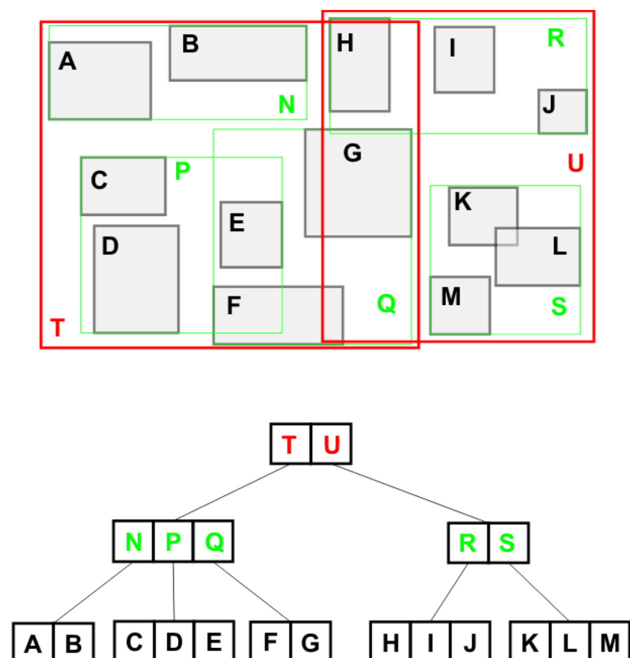

Figure 2. Set of rectangles indexed by an R-tree (top) and the corresponding R-tree structure (bottom)

Range queries and nearest neighbour queries can be performed with R-tree. In a range query, a bounding box is fed as input to the R-tree. Starting from the root node, this bounding box is

https://doi.org/10.5194/isprs-archives-XLII-4-W2-95-2017 | © Authors 2017. CC BY 4.0 License.

evaluated for overlap with the MBR of that node. If there is overlap, the child nodes are also evaluated for overlap until we reach the leaf nodes. To perform a nearest neighbour search, nodes (leaf and non-leaf) are inserted into a priority queue according to their distance from the search point. Depending on the number of nearest neighbours required, the priority queue is now used popped and the encountered leaf nodes are returned.

2.5 Query processing

We have used the R-tree python library to index spatial data. We initiate the index object and add feature ID and MBR of each feature into the index. The library provides bulk-loading that takes all feature IDs and MBRs as input, pre-sorts them to create groups or pages and inserts them into the index. When compressing a GML file, we serialize this index to a file on disk. Therefore, we now get 2 files after compression – a GMZ file and an index file. One advantage of this 2-file system is that the GMZ file can be used independently if querying is not required.

For a range query, we get a bounding-box as input and we return a list of features as output. We follow these steps:

1. The first step is reading the index file to get the python index object.
2. The library provides the intersection method of the index class for searching geometries that lie within this bounding-box. The method takes the query bounding-box and returns a list of feature IDs that lie within that box.
3. Read the GMZ file and decompress the attribute container. Filter out attribute rows of features that lie in the feature IDs list.
4. Get the byte positions of feature’s bounding-box coordinates for each feature and merge these bounding-boxes to get query level bounding-box coordinates. Decompress X and Y coordinates for the bigger bounding-box.
5. Get the start byte position of a feature’s coordinate indices stored in the index container from the attribute row of each feature. Get the minimum and maximum of these byte positions and decompress all indices that lie within minimum and maximum.
6. Reconstruct GML/GMZ for the output features.

3. CONCLUSION

3.1 Results

The size increase due to changes made in the compression model and the creation of a new index file is almost negligible. The same can be said about the compression and decompression times. This makes sense as we make minimal changes in the compression model.

We list down queries that we ran on some of the files with the times it took to execute those queries. Features decompressed is the number of features decompressed after the query, total query time is the total time it takes to decompress all the features in the query and query time per feature just total query time divided by features decompressed. The time is in milliseconds.

Features decompressed (Total number of features)	Total query time	Query time per feature
2 (37)	360	180
10 (37)	920	92
33 (37)	1180	35.75
46 (667)	810	17.3
128 (667)	900	7.03
344 (667)	1050	3.05
617 (667)	1120	1.8
25 (2340)	710	28.4
505 (2340)	1530	3.03
2015 (2340)	2140	1.06
284 (32563)	2950	10.38
6681 (32563)	3810	0.57
15122 (32563)	4330	0.29
25772 (32563)	5100	0.2

Table 1. Decompression performance

The query performance is reasonable. We can see that as features decompressed increases for the same data file, the query time per feature decreases. This was expected as disk read time per feature decreases significantly due to sequential access. These results are preliminary. Extensive testing needs to be done on various GMZ files with various spatial and non-spatial queries.

3.2 Conclusion and future work

The preliminary results presented in this work demonstrate that querying a GMZ file in compressed state is feasible. With an efficient indexing mechanism, support for partial decompression and optimized I/O operations, GMZ with query support fits well in the context of webGIS. The two bottlenecks of webGIS, bandwidth and query efficiency have been dealt with convincingly.

Future work will mainly focus on implementing GMZ support in an open-sourced WFS server. FeatureServer is an implementation of a RESTful Geographic Feature Service using standard HTTP method. It is lightweight, WFS compliant and written entirely in python. It supports usage of any of the OGR datasources, including GML, as backend data providers. It will be worthwhile to compare the performance of GMZ with GML using both as data backends. Another comparison we would like to make is pitting GMZ against PostGIS/PostgreSQL. We would like to use GMZ throughout the entire pipeline of a WFS server and test its capabilities in data storage, querying and data transfer over the Internet. We would also work on improving query performance for primary queries by coming up with better techniques of sorting data for increased sequentiality. Query performance for secondary queries can also be improved.

4. REFERENCES

Cheney, J., 2011. XMLPPM: XML-Conscious PPM Compression. *See http://www. cs. cornell. edu/People/jcheney/xmlppm/xmlppm. html*.

Cheng, J. and Ng, W., 2004, March. XQzip: Querying compressed XML using structural indexing. In *International confer-*

ence on extending database technology (pp. 219-236). Springer Berlin Heidelberg.

Dai, Q., Zhang, S. and Wang, Z., 2009, October. GQComp: A Query-Supported Compression Technique for GML. In: *Computer and Information Technology, 2009. CIT'09. Ninth IEEE International Conference on* (Vol. 1, pp. 311-317). IEEE.

En.wikipedia.org. (2017). *Web Feature Service*. [online] Available at: https://en.wikipedia.org/wiki/Web_Feature_Service [Accessed 27 May 2017].

Guan, J. and Zhou, S., 2007, April. GPress: Towards effective GML documents compresssion. In: *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on* (pp. 1473-1474). IEEE.

Harshita, N.N. & Rajan, K.S., 2010. Effective Topological and Structural Compression of GML coordinate Data. In: *Global Spatial Data Infrastructure (GSDI 12)*. Singapore. GSDI 12.

Khandelwal, A. and Rajan, K.S., 2017. GMZ: A compression model for WebGIS. In: *1st International workshop on Web Mapping, Geoprocessing and Services (Submitted)*

Li, Y., Imaizumi, T., Sakata, S., Sekiya, H. and Guan, J., 2008. Spatial Data Compression Techniques for GML. In: *Frontier of Computer Science and Technology, 2008. FCST'08. Japan-China Joint Workshop on* (pp. 79-84). IEEE.

Liefke, H. and Suciu, D., 2000, May. XMill: an efficient compressor for XML data. In *ACM Sigmod Record* (Vol. 29, No. 2, pp. 153-164). ACM.

Min, J.K., Park, M.J. and Chung, C.W., 2003, June. XPRESS: A queriable compression for XML data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (pp. 122-133). ACM.

Peng, Z.R. and Zhang, C., 2004. The roles of geography markup language (GML), scalable vector graphics (SVG), and Web feature service (WFS) specifications in the development of Internet geographic information systems (GIS). *Journal of Geographical Systems*, *6*(2), pp.95-116.

Sakr, S., 2009. XML compression techniques: A survey and comparison. *Journal of Computer and System Sciences*, *75*(5), pp.303-322.

The Geospatial Web – How Geobrowsers, Social Software and the Web 2.0 are Shaping the Network Society. (2007). *Management of Environment Quality: An International Journal*, 18(5).

Tolani, P.M. and Haritsa, J.R., 2002. XGRIND: A query-friendly XML compressor. In *Data Engineering, 2002. Proceedings. 18th International Conference on* (pp. 225-234). IEEE.

Wei, Q., & Guan, J., 2010. A GML compression approach based on on-line semantic clustering. In: *Geoinformatics, 2010 18th International Conference on*. June 2010. IEEE.