Scalable Geospatial Object Database Systems

By Adrian Marriott

Introduction

The purpose of this paper is to describe the Geospatial Object Server (GOS) that lies at the heart of the OS Master Map project deployed Ordnance Survey in the UK. The aim of OS Master Map is to underpin all commercial and government activity that involves spatial data into the next millennium. Here we describe what the GOS does, why it does these things and how it is implemented to scale to billions of objects. We also look at project risks and business benefits.

GOS Overview

The Geospatial Object Server (GOS) holds the largest, seamless spatial dataset in the world, covering the whole of the UK at 1:10000 scale. Its primary task is to supply this data in various XML formats in response to queries. These queries are specified in two ways: *Area Queries*, which request a set of objects within a certain spatial extent; and *TOID Queries*, which specify a list of objects using their object identifiers. There are two supply modes: *Feature Serving*, where the result set is returned directly to the client; and *Bulk Supply*, where query results are written to local disk and the client is informed, for example by email, when the data is ready to collect.

In support of this main task the GOS must be updated. As surveys are undertaken by the Ordnance Survey and other parties, new information is incorporated into the GOS database as rapidly as possible, and these updates must be transactional. In this way the data in the GOS represents the most recent, accurate and consistent surveyed data of the UK.

The GOS is highly scalable and supports ever larger numbers of users. It provides a transactionally consistent view of Ordnance Survey data available all day, every day of the year (24x7). The database at the centre of this system is the high-speed object database, ObjectStore™ C++ from Progress Software

GOS as 'Black-Box'

The GOS system was designed as a fully encapsulated component within a wider computing context and as such, client processes of whatever type can view the GOS as a 'black box'. The boundary around this black box is XML. Queries are presented as XML and results are returned as XML. The GOS update process has been designed to follow a 'check-in, check-out' protocol. Check-out requests, which lock areas while they are re-surveyed, are presented as XML, and the new survey data that results is submitted (checked-in) to the GOS for validation and update as XML.

Queries

One of the aims of OS Master Map was to assign every surveyed map feature (house, garden, fence etc.) a unique identification number called a Topographic Object Identifier (TOID). The presence of unique identifiers with a well defined life-cycle improves the reliability of survey data. As previously mentioned the GOS can be queried in two ways, by area or by TOID, but either of these can be combined with a set of 'themes' and/or a 'change-only date'.

Themes group objects into sets that have something in common of interest to the user. For example, water, buildings, railways and roads are examples of themes. A theme can be visualised as a transparency covering the map area with objects in that theme drawn on the transparency. The analogy breaks down because any object can be in more than one theme, (the same object can appear on several transparencies). By specifying a set of themes with the query a user can filter the result set at source.

The change-only date allows users to request a result set only containing those objects that have changed since a particular date. The update process of the GOS tracks all changes to map features including movements (e.g. as a result of improved survey accuracy) and deletions. This is useful for customers that hold large geo-spatial data sets that are updated from the Ordnance Survey’s main data set within the GOS.

Object Update

The GOS was designed to support two types of update. One from object based editors using the aforementioned check-in, check-out model. The other from *tiled data*, because the initial data source comprised 230,000 tiles of CITF data held by Ordnance Survey. This mode requires that part-features are ‘stitched’ across tile boundaries and is described further below.

Architecture

The GOS was designed as a truly distributed system with multiple processes running on several computers, each of which hosts a number of separate ObjectStore databases. The process architecture is divided into two broad activities; those concerned with processing queries and serving out the results and those concerned with managing data loading and updating.

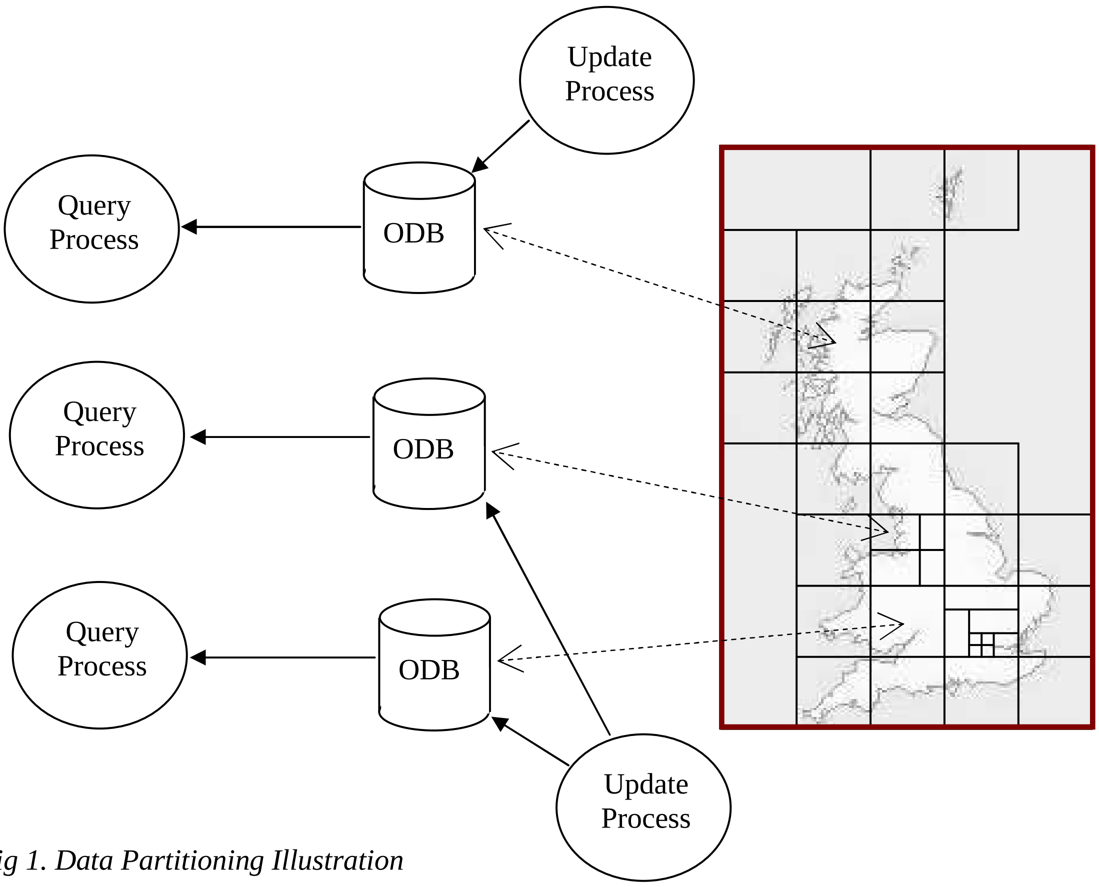

Fig 1. Data Partitioning Illustration

Each ObjectStore database holds the data for a particular geographical area, and is only read by a particular, dedicated query process, and updated by a particular update process. Using multiple processes gives two advantages: the total address space available to map persistent database objects into memory is increased, and the entire dataset can be partitioned across several machines, which means the individual query processes can execute in parallel. Exploiting true parallelism significantly reduces query times.

To date, there are 700+ databases in total spread over 12 x Computers each with 4 CPUs and 4 Gig of RAM. The total dataset comprises in excess of 2 billion C++ objects (approximately 1 billion features) currently occupying around 300Gbytes, held on RAID disk arrays with both mirroring and stripping (RAID 1+0). This was expected to grow at a rate of between 2 and 5% a year.

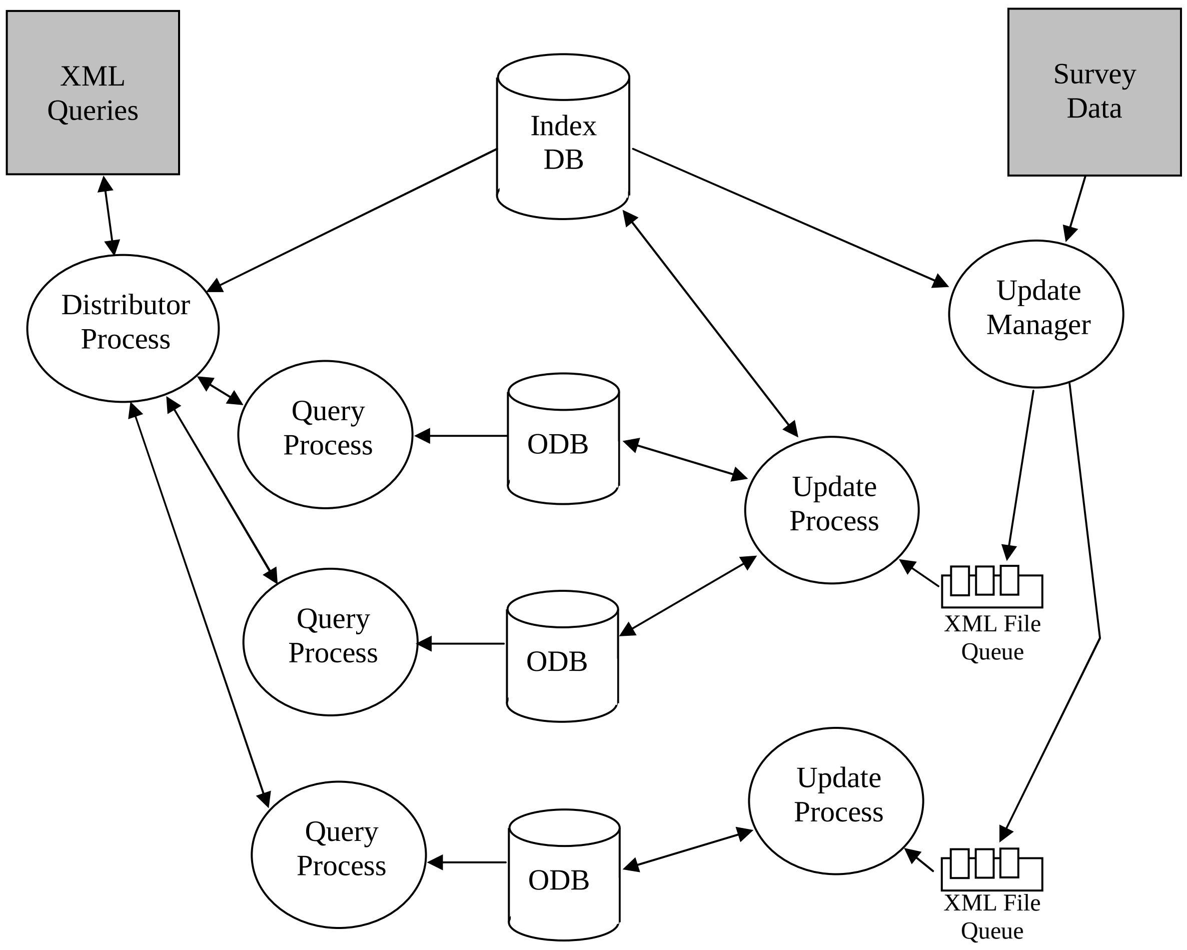

Fig 2. Process Architecture

Distributor Process

This is the process that receives queries in the form of XML from outside the GOS. The responsibility of the Distributor process is to identify which query processes are needed to execute the query, forward the query thereto, and combine the results when they are returned. To do this the distributor uses the index database to map query areas and TOID lists to the correct query processors. The distributor process has been designed not to accumulate or maintain state, and does not need to examine the result data returned, for example to remove duplicates, so that the system can scale.

Query Process

This is a process to which the Distributor delegates most of the work of executing the query. Each query process is responsible for executing queries against a small number of databases each covering a unique area of the country. In this way queries into the GOS are distributed among several processes and can run in parallel. These query processes are distributed across many CPU's and so genuine parallelism results.

The query processors open all the databases they use in a special high-throughput ObjectStore database open mode called MVCC (Multi-Version Concurrency Control mode). This allows query processors a transactionally consistent, read-only view of multiple databases, which neither impedes, nor is impeded by, concurrent updater processes.

Updater Process

There are multiple updater processes that can run in parallel if required, each being responsible for a different area of the country and each having their own data queue. Data arrives in a CITF data file and, depending on the area the file covers, is placed in the correct queue. The updater process parses the file, creates the persistent objects within the GOS databases and stitches across file boundaries to adjoining geometric objects where these are already loaded.

To ensure data integrity each tile is loaded within its own ObjectStore transaction. In the event of exceptions or data errors the database is rolled-back and the file is placed in a 'Defect' area ready for human intervention. Each process works across multiple databases, one of which is the top-level index database. Upon successful update a completed file is moved into a 'Done' area.

The top-level index has been specially designed so that updaters only need to write to the top-level index every million TOIDs or so eliminating contention and deadlocks on common top-level objects. Deadlocks and blocks can still occur between updaters working in adjacent areas. This is minimised by co-ordinating the actions of the updaters so that where possible they work avoiding common boundaries. Where deadlock does occur, one of the processes is sent an exception that is caught, the ObjectStore transaction is rolled-back and the transaction is automatically re-tried.

Inter-process Communication

All processes described above communicate using a bespoke sockets library developed for the project. This is light-weight, robust, very easy to use and executes as fast as the current fastest TCP/IP technology. The design is based around ideas used by the Java library; sockets of different types are wrapped in an iostream interface, so they can be flexibly combined.

When large amounts of data need to be moved around the system, the data is passed through a socket class which automatically compresses into, and uncompresses out of, the socket buffers using the zlib algorithm. This achieves compression ratios of 20:1 on XML.

Seamless Topological Object Model

Conceptually, the geometry used in maps is like a patch-work quilt; it consists of a set of areas that totally cover the map with no gaps and no overlapping polygons. This is because every surveyed area is classified in some way (no gaps), and any area on a map can only be coloured using a single colour (no overlapping regions). However, there is a distinction around the ideas of '*topology*' versus '*shape*'. The distinction between the two is: Topological data contains adjacency and containment information, whereas shape data does not. This is illustrated in the diagram below.

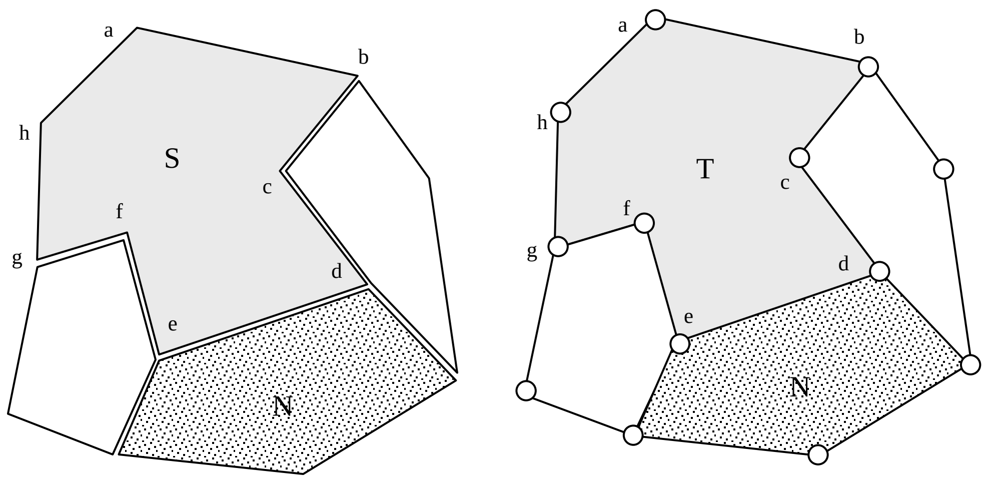

Fig 3. Shape vs Topology

On the left of fig 3 above we see a polygon S with edges labelled *a* – *h*. With shape data, the area S owns all its lines and points. This means that the polygon S can be directly implemented as a list of points *a* - *h*, the line data can be derived directly from the points. It also means that the data defining the common edge *e-d* with the neighbouring area N is stored again in polygon N. The shape model does not explicitly model the adjacency of areas S and N; given area S it is possible to find out if area N is a neighbour by analyzing the geometrical values of their constituent points.

On the right we see the same set of polygons represented topologically. Here each area is defined as a set of lines, but the lines are not owned by the area. This means that area T is an irregular octagon defined by a set of eight lines, and it is the line objects that contain the point data that ultimately determine the location of everything. This means that the data defining the common edge *e-d* with the neighbouring area N is stored only once, in an independent line object. The topological model explicitly models the adjacency of areas S and N via the line *e-d*; given area S (and suitable back-pointers from the lines to the areas either side of it), it is possible to ascertain that N is a neighbour of S by navigation.

The GOS stores a topological model, probably the biggest *seamless topological* object model in the world, where the adjacency and containment relations are explicitly modelled; areas reference line objects, and these line objects own a set of points. Each feature is only stored once. This is important for two reasons:

1. Topological error checking of data, which is a complex algorithm, is easier if the model is explicitly topological and each feature is unique.

2. With features stored once it is possible to arrange for query results to be ***generated*** without duplicates, even when different parts of the same query are being executed on remote CPUs.

The first of these impacts the quality of the data held in the database, and both significantly improve the runtime performance of the system overall.

The GOS data set is partitioned between many individual ObjectStore databases so that they can be distributed easily over many computers, and still this topological model is seamless. Superficially this appears to be a contradiction, but by associating two areas with each database, this can be resolved.

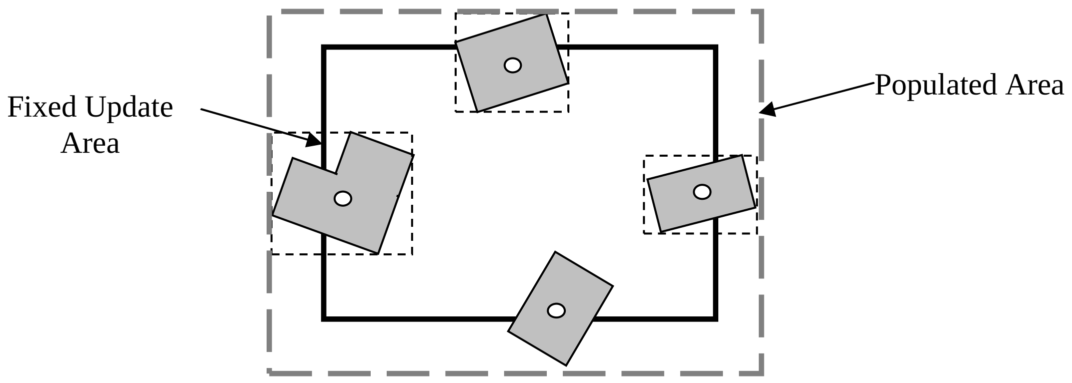

Fig 4. Nominal and Populated Database Areas

Above we see the 'nominal' fixed area of the database, which is used during update to determine which objects are held in which ObjectStore database. Each feature has a bounding rectangle. The centre point of this bounding rectangle can only fall into a single nominal area, and this determines which database an object is allocated within.

As shown, features can overlap the fixed boundaries of databases. The other area associated with any individual database is its populated area, which bounds all the feature geometry contained within, and is dynamic; as new feature allocations occur, and the population changes, so this area will change. The populated area is used to distribute queries between the various query processors.

Cross-Database References

Since the GOS stores a topological model spread amongst many databases and each feature is stored only once, there have to be cross-database references. This is not a problem for ObjectStore because it can store C++ pointers between objects located in different databases with no measurable performance overhead. However, there was a requirement to minimise the number of cross database pointers so that the entire data set can be split into individual databases and managed, not as one single seamless entity, but in smaller chunks when necessary. For example, administration tasks such as taking a database offline, moving it to another machine or restoring part of the data set from backup.

As described above there are broadly three types of objects within the GOS: areas, lines and points. Point and line features are fully contained within a single database, even where a line feature crosses a database edge. Areas are represented by a single object which is allocated according to the centre point of its bounding box, but it must *refer* to its line geometry. Consequently, the most common cross database reference within the GOS is between an area and its lines, and vice-versa. The GOS team designed a container that used TOIDs when the target object was in a remote database and direct C++ pointers to targets within the local database. The solution was based on a template class that uses two internal C++ arrays that are allocated dynamically into the database. It has two integer members, one holding the length of the array and one holding the number of cross database references.

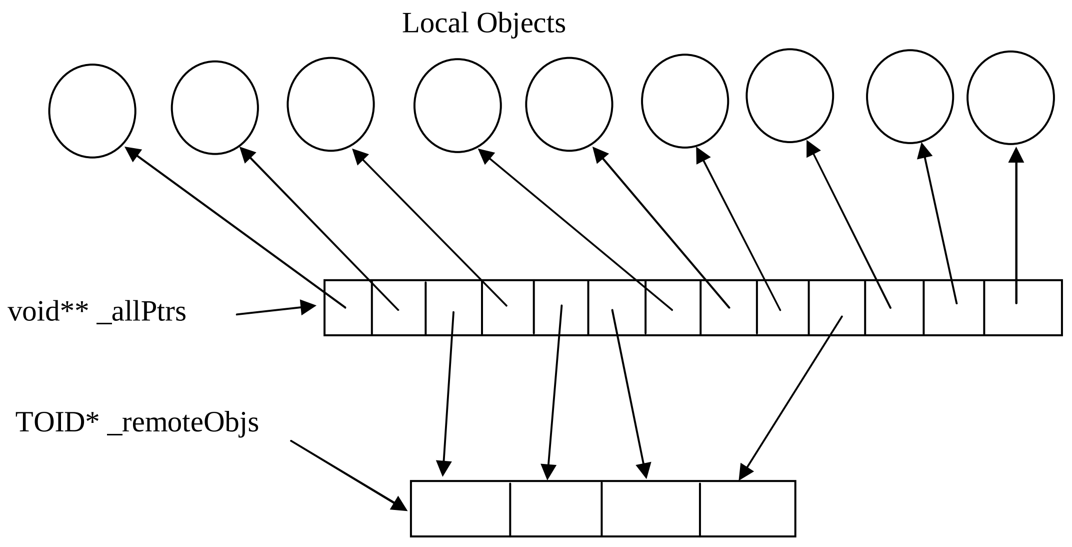

Fig 5. Cross Database Reference Implementation

The array operator is implemented to return a pointer of type T (the type of the local and remote feature objects). Assume we have a call for the ith element of the array, the array operator simply tests the value of the pointer _allPtrs[i] and checks to see if it falls within range of the remote objects array, using pointer arithmetic. If it doesn't then it must point to a local pointer and can be safely cast to a type T* and returned.

Otherwise the pointer is cast to a TOID*, the value of which is used to find the correct object in the remote database. A T* to this object is then returned.

Inserting into the collection follows a similar process. Allocate space for a copy of _allPtrs plus one extra space. Test whether the new element is within the local database or refers to a remote one. If it’s remote then copy the remote object array and add the new TOID at the end. Now copy the existing void* pointers but re-target all those that point into the old TOID array into the correct position of the new TOID array. Finally we clean up the temporaries and set the size and object count members.

This design has some interesting features:

1. Uses mostly direct pointers within the local database because most references between any area and its lines are within the local database, so statistically it is fast as possible.

2. Internal management of the relationships uses pointer arithmetic, memcpy calls and casting, all very fast operations in C++, so the runtime execution is highly efficient.

3. The design has the absolute minimum of overhead in terms of database size. When compared to using a simple C++ array of pointers to local objects for areas that do not reference any lines in remote databases there is a fixed overhead per area of one extra integer and one extra pointer.

4. As the array is updated it maintains its locality of reference characteristics well. This is because ObjectStore can store real C++ arrays and raw C++ pointers as actual arrays and pointers. If linked list structures were used here instead of C++ arrays then as objects were inserted, although the new pointers can be allocated within the same cluster and no copying of existing data values is required, there is no guarantee that all the pointers will be on the same page.

Given the efficiency of this design, and the huge numbers of area objects within the data set, and because this complexity could be encapsulated properly, and generalized appropriately with C++ templates, the extra implementation and testing effort was judged to be worth it.

A variant of this design was used to efficiently implement the back-reference from lines to the areas either side of them.

GOS Indexes

Access speed is greatly affected by index efficiency. Using an object database, such as ObjectStore, and C++ enabled the implementation of highly efficient, bespoke indexing classes, almost impossible using a relational database. There are two basic queries in the GOS each supported by dedicated index structures. One supports spatially based queries and the other supports TOID based queries.

Spatial Index

The spatial index is a refinement of the well established quad-index commonly used in GIS applications. Here we briefly describe the structure and behaviour of a quad index and highlight some of its inherent problems. Then we describe how the Spatial Index designed at Ordnance Survey solves these problems.

Assume there is a rectangular map area containing various map elements as C++ objects. A quad-index divides this map area into four equal sized cells and holds the objects that are contained within the cell in a separate collection. Queries are presented to the index in the form of rectangular areas that partially cover the map. The index works by reducing the number of objects that need to be examined to fulfil a query.

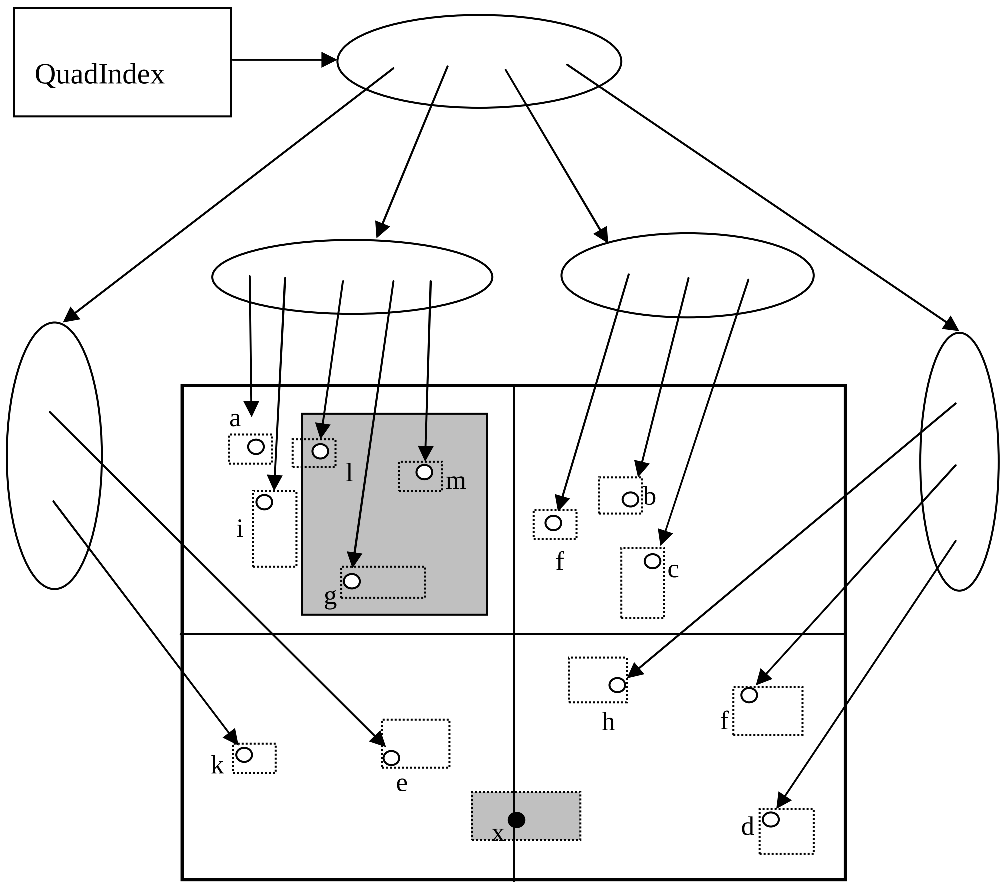

Fig 6. Quad Index Diagram

The diagram in fig 6 above shows a quad index with the map area divided into 4 cells and containing objects (a – m), each of which has an associated bounding box. Each cell is represented by a collection of pointers to the objects contained within that cell. The grey rectangle represents a query. To fulfil the query only 5 objects need to be examined (a, g, i, l, m).

First, an overlap test is done between all the cells and the query. In this case only one cell overlaps (the top left). Now the overlap test is re-applied to all the bounding boxes of the objects within that cell. This produces the query result set.

Problems with Quad Indexes

There are a few problems with quad indexes that make them unsuitable at Ordnance Survey.

Objects Overlapping Multiple Cells

Objects whose bounding boxes overlap more than one cell, as typified by object x in fig 6 above, cause problems. As the database is updated, the configuration of objects changes and objects need to be included in multiple cells. There are several strategies for accommodating these awkward objects:

1. Have copies of the object into both cells.
2. Have pointers to the object in both cells.
3. Have the object arbitrarily located in one or other of the cells.
4. Have a special top level cell containing all the awkward objects.

All have draw backs. 1 and 2 require the removal of duplicates from the query result set since queries that overlap both cells result in the objects been returned twice. Option 3 produces incorrect results with queries that touch the edge of a cell. Objects that have bounding boxes that actually overlap the query area have been randomly located inside the adjacent cell whose 'bounding box' does NOT overlap the query. Consequently, the cell is eliminated early and overlap tests against the individual object's bounding box are never run. Option 4 inhibits the smooth scaling of the index when there are large numbers of small objects that overlap cell boundaries. This is exacerbated by the process of cell fission described below.

Cell Fission Compromises Physical Object Clustering

As objects are added to a quad index, there can come a point when a cell gets so full that it takes almost as long to search that one cell as it does to search the entire map area. The quad index has a density threshold above which a cell splits. At this point, the cell is sub-divided into four smaller cells and the single 'node' of the original cell is replaced by four smaller nodes each containing its share of the pointers. These cells are 'structurally self-similar' to the cell above. Although this process is theoretically sound it presents practical performance problems.

Ideally, using the index touches as few pages as possible in the database, so we want to mirror the structure of the index in the physical structure of the database. If we assign each cell of the index to a cluster in the database this works until cell fission occurs. At this point, the single cell (one cluster) is replaced by four new cells (four new clusters) and, because an object's position in the database is fixed at construction time, it would be necessary to copy all the existing objects in the original cell into one of the new clusters based on their bounding box. Finally the original cluster is deleted.

In itself this significantly impacts index update speeds, but the problem is compounded by the pointer fix-ups for relationship management. Any other objects within the database that refer to the objects contained in the cell that is undergoing fission must be found and have their pointers re-targeted at the newly constructed copies. This is computationally expensive.

Notice also, that cell fission tends towards an increase in the number of objects that lie across two or more cells. This would further degrade quad index performance as described above.

Spatial Index Refinements

The spatial index designed at Ordnance Survey refines the ideas behind the quad index by using nine overlapping cells in every layer. This is shown below.

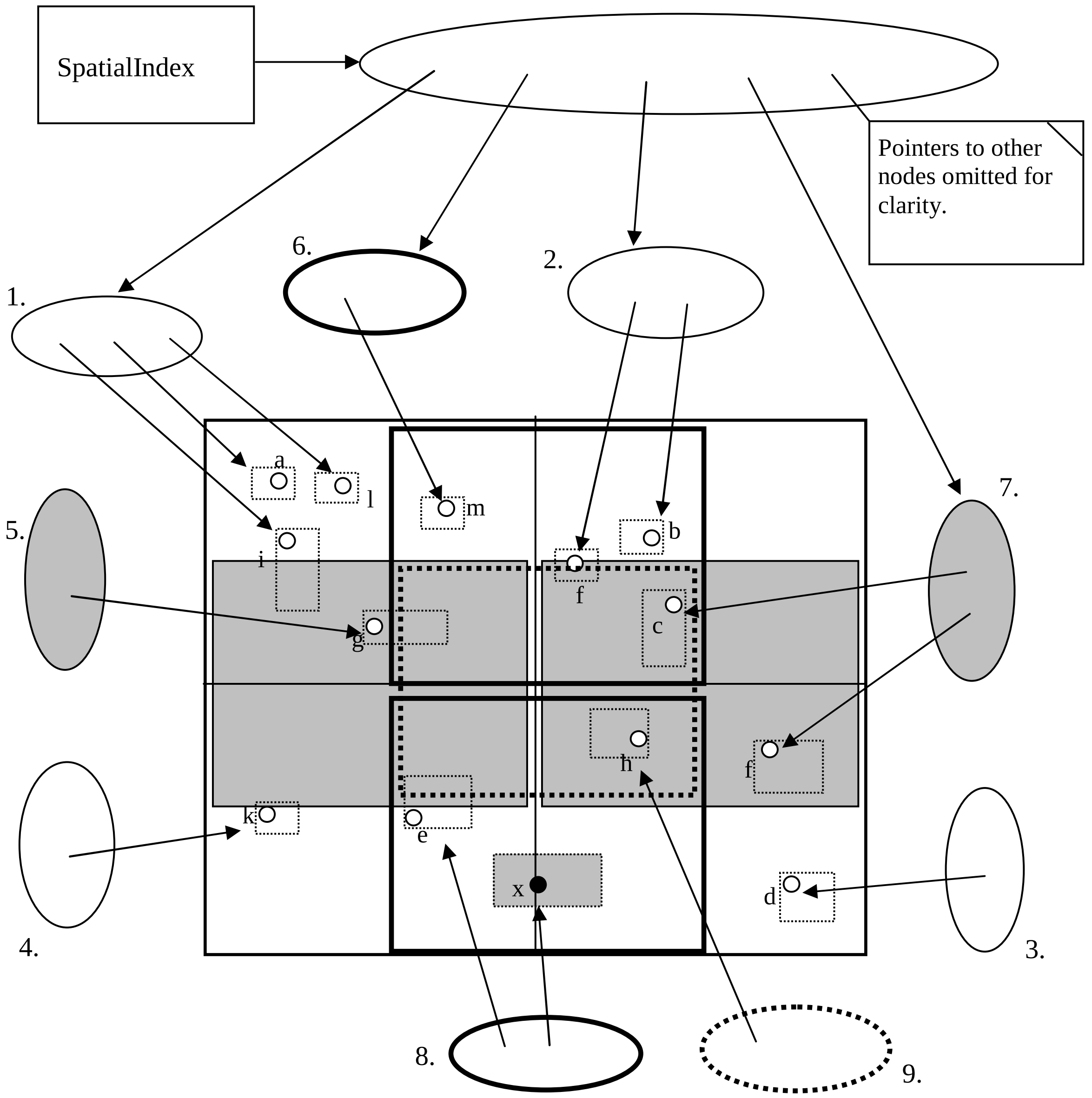

Fig 7. Spatial Index Structure

In fig 7 above four of the extra five cells (5,6,7,8) cover the joins between the original four cells (1,2,3,4). The fifth new cell (9) is placed centrally to cover the case when an object overlaps the edges of the other four newly introduced cells. In the event cell fission this same nine node structure is created in miniature within the area of the original cell.

Solutions to Quad Index Problems

The solutions to the aforementioned problems are outlined.

Overlapping Object Problem

The use of nine overlapping nodes totally eliminates the occurrence of objects which overlap nodes. Now any given object can be located in *at least one* node that fully contains it. If there are multiple nodes that are candidates for containing an object, such as when the object is smaller than the resolution of the index, one is chosen arbitrarily. For example in the diagram above, object 'c' could be equally well located in nodes 2,7 or 9. The choice does not influence index performance in any significant way. This means that there is no requirement to remove duplicates from query results or compromise index search performance with large 'catch all' nodes that do not scale smoothly.

Cell Fission and Object Clustering

In the new Spatial Index when cell fission takes place a new layer is created containing another 9 nodes with a self-similar structure to the original. However, the cell fission process takes place at a different time than in the case of a quad index. There is no density threshold associated with a cell. Each Spatial Index has a single figure associated with it **-** its resolution. This determines the smallest size for any cell. Cells created at the size of the resolution can never fission. As each object is inserted into the Spatial Index recursive cell splitting occurs so that each object is located in the smallest cell that fully contains its bounding box. This means that each object is located immediately within the smallest cell (above the resolution size of the index) that can contain it.

This combines very well with ObjectStore because once an object has been positioned in the database the index structure is created immediately to reflect that physical location. Further index updates do not require any movement of objects that already exist in the database. This eliminates the problem of object copying and pointer fix-ups during index update.

In fact if the area covered is densely populated index updates scale very well indeed, because statistically it is more likely that the nodes for the index have already been created. Where there is prior knowledge that the area is densely populated it’s also possible to pre-create all the index nodes, and all the corresponding database clusters further reducing the time for index updates.

The clustering strategy was further improved by clustering the top four cell layers of the index in a dedicated database cluster. By locating these commonly used node objects on as few pages as possible, statistically for any given query these pages are likely to be pre-fetched and mapped. The other index nodes are distributed around the database within dedicated clusters so that when they are accessed, they too result in the minimum number of page fetches.

At Ordnance Survey the developers further refined the index to support themes directly, by embedding an integer with every feature reference in the index. For each theme that the feature was ‘in’, a bit was set. Thematic queries are then efficiently satisfied using bit manipulation; in fact the spatial index is actually a *spatio-thematic* index.

TOID Indexing

The GOS contains an estimated 1 billion features each with a unique 64 bit identifier (TOID). For security, TOIDs are ‘location opaque’, which means it’s impossible to discern the location of the corresponding feature from its value. Using a naïve approach to index design, each index entry would consist of at least the key and a pointer to the object; in this case 8 bytes for the TOID and 4 bytes for each pointer. If we assume there would be another 20-30% for any internal hash-table structure this yields an index of around 15-16Gb!

To be effective the index needs to be as dense as possible so that it occupies the minimum number of memory pages. The solution in the GOS involves storing TOID ranges rather than each TOID explicitly.

TOID indexing is split into two tiers; an index database that maps TOID ranges to databases, and a local TOID dictionary, one per ObjectStore database, that maps actual TOID values to C++ pointers targeted at the feature objects.

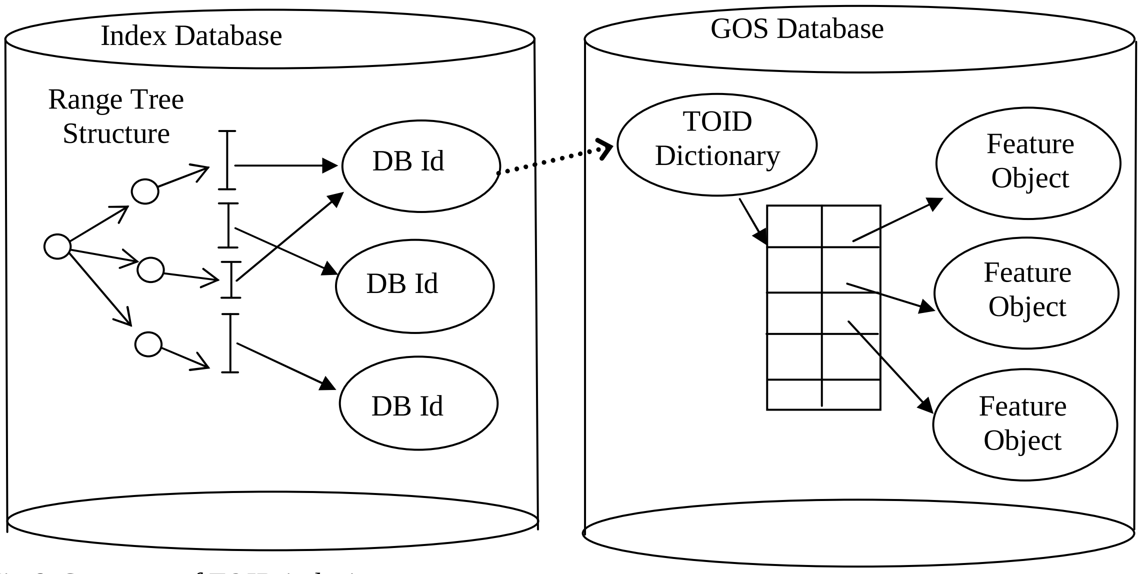

Fig 8. Structure of TOID indexing

Each database is assigned sets of sub-ranges of between 1-3 million possible TOID values, depending on the expected data density for that area. As new TOIDs are created in a particular database it uses up its assigned range. Within the database the values of TOIDs are assigned at random. When a database runs out of TOIDs it requests a new range from the index database. Each TOID range maps to a single database, and each database ‘owns’ several ranges.

To find one feature out of the billion or so within the GOS, the index database is examined to find which range its TOID falls in, and return the database that the range was assigned to. Then the persistent dictionary within the GOS database containing the feature object is accessed and a raw C++ pointer to the feature is returned. This is very fast indeed and operates at sub-second speeds.

This design has some interesting features:

1. The theoretical size calculation of the top level GOS index is now very much smaller; 1000 x databases each with an average of two ranges, each consisting of 2 integers, plus a pointer to the database identifier, totalling around 24Kb!

   In actuality, this index is implemented as a tree structure so that it can automatically accommodate arbitrarily presented TOID ranges. Even so, this index only occupies a few pages of memory, and because of ObjectStore's LRU cache eviction algorithm, it is statistically likely to be memory resident when required.

2. The rest of the index is spread between the GOS databases as feature extents keyed on TOID. Assuming an equal spread of features between databases, each dictionary contains around 2 million entries; well within their capacity.

3. Updates to GOS databases almost never require updates to the central index database. Updates only occur when a database needs to be allocated a new TOID range. This eliminates an obvious concurrency hotspot and allows update processes to run in parallel with the minimum of blocking and deadlocks.

4. The Ordnance Survey has a business requirement that other agencies external to the Ordnance Survey can also assign TOIDs to surveyed features that are globally unique *when assigned by the surveyor*. Small ranges from within the possible 2^{64} values are assigned to these organisations as needed and recorded by the GOS.

The performance of this index is best described using an analogy. It has been calculated that if each map feature were a 1mm grain of sand shaped like a cube, then the GOS would consist of a pile of sand weighing around 2.4 metric tonnes. In this analogy the GOS can sift through this pile of sand and find a particular grain in less than a second!

GOS Compression

One of the main issues for the GOS project is the sheer size of the data set. Compressing the data both in storage and during transmission between processes provides major performance enhancements. Here we examine techniques used to compress storage.

Storage Compression

There were several techniques employed here, all based on examining the data and finding the best way to remove redundancy. The first was the use of string tables and the second was the use of C++ bit-fields.

String Tables

The number of times that the word 'street' appears on a map is enormous. Each GOS database has its own string table where it stores strings in an ObjectStore dictionary keyed on an integer. Now within the database all references to a particular string, after the first occurrence, only take up 4 bytes, independent of whatever is in the string. All this is encapsulated within a String and StringTable class for ease of use. The advantages of this approach are:

1. Comparisons for string equality within the same database are very fast, because they are reduced to a single integer operation.

2. The total number of pages fetched for queries covering large areas of the database is greatly reduced. However, small queries can actually end up fetching slightly more pages because string tables tend to compromise 'locality of reference'. If the string were directly embedded within the object of interest then it would be fetched on the same page as the object. When placed in a string table, which is typically located in another cluster, then at least two pages are required for the object; one for the object and one for the string it 'contains'. Its not until the query area gets bigger, and nearly all the strings in the database area are required that the table performance improves. This is because statistically repeated string requests increase the probability that the pages of the string table containing them will have already been fetched. So for large queries the total number of pages faulted into the cache is lower - and it is exactly these queries that we need to optimise. Small queries are already very fast. So the string table works.

Bit-Field Compression

ObjectStore stores C++ objects directly to disc exactly as they are laid out by the compiler in memory (all except a little pointer swizzling). Consequently, if we can save bits in the physical layout of an object by judicious use of C++ bit-fields we can fit more objects per page. This means that, when we need to access these objects we fetch fewer pages from the server.

Compression starts by examining the storage model for classes that have many private member variables directly embedded within them. The type of the variable and the number of possible values or the range of possible values is then analysed to ascertain the minimum number of bits needed to represent this data. For example supposed we had a class Foo with the following members:

```
Colour _colour;
bool _isNull;
int _day;
int _month;
int _year;
```

If we record the number of bytes needed by the compiler to lay this object out we need 4 bytes for the enum (they are stored as ints), bool is the size of a character, 1 byte, and the other three are all 4 bytes. This 4+1+4+4+4 = 17 bytes. With word alignment this gives 18 bytes on most compilers.

Now we examine the possible values for these items and calculate the minimum number of bits necessary to differentiate them. Lets assume it’s discovered that the _colour member is an enum with 3 possible values; Red, Blue and Green. So this requires a minimum of 2 bits for storage. Boolean values only need a single bit. 31 days needs 5 bits, 12 months needs 4 bits, and the year, in this particular context 200 years is quite adequate, needing 8 bits. So after this analysis we have: 2+1+5+4+8=20 bits, with word alignment 3 bytes. This is a size reduction of around 80%.

In situations where there are millions or tens of millions of these objects then implementing the state of these objects as C++ bit-fields, or alternatively as embedded integers and access the individual values using an &-mask, can save large amounts of database space. For example:

```
class Foo
{
public:
 enum Colour {Red,Blue,Green};

 void setColour(Foo::Colour c){
   _colour = c;
 }

 Foo::Colour getColour(){
   return _colour;
 }

 void setNull(bool b){
   _isNull = b;
 }

 bool isNull(){
   return _isNull;
 }

 void setDay(int d){
   _day=d;
 }

 int getDay(){
   return _day;
 }

 // Etc. etc.

private:
 Colour _colour;
 bool _isNull;
 int _day;
 int _month;
 int _year;
};
```

```
class Foo
{
public:
 enum Colour {Red,Blue,Green};

 void setColour(Foo::Colour c){
   switch c
   {
     case Red: _colour = 1;
       break;
     case Blue: _colour = 2;
       break;
     case Green: _colour = 3;
       break;
     default:
       assert(0);
   }
 }

 Foo::Colour getColour(){
   Colour ret = Red;
   if(_colour == 2){
     ret = Blue;
   } else {
     ret = Green;
   }
   return ret;
 }

 void setNull(bool b){
   _isNull = ((b)?1:0);
 }

 bool isNull(){
   return ((_isNull==1)?true:false);
 }

 void setDay(int d){
   assert(d>0 && d<32);
   _day=d;
 }

 int getDay(){
   return _day;
 }

 // Etc. etc.

private:
 unsigned _colour : 2;
 unsigned _isNull : 1;
 unsigned _day : 5;
 unsigned _month : 4;
 unsigned _year : 8;
};
```

Fig 9. C++ Bit-field Code Example

In fig 9 is an example class Foo comparing a naïve implementation on the left, with a compressed version that uses C++ bit-fields as described. Note that the accessor functions on the class convert to the correct type as required. For example the boolean can be stored in a single bit and when isNull is called, it uses the bit-field to determine the current value of the bit and returns a bool. Note also the ubiquitous use of assert to protect against values that are outside the range of the bit-field.

The advantages of this approach are:

1. Greater database storage density. Fewer pages are fetched from the server. There is a very nearly linear relationship between performance and compression. Halving the size of the storage object typically doubles execution speed.

2. There is no measurable transient processing overhead. The cost of accessing the bit-field and returning another type was simply lost in the noise.

3. The code is very simple, albeit tedious.

The same approach can be taken with floats and doubles. Again we analyse the range of possible values and the number of decimal places required for data members of these types. Where it is discovered that the range and accuracy constraints allow the storage is converted to some smaller integer type; either bit-fields, chars, shorts or integers, and the accessor and mutator functions multiply or divide the stored value appropriately before update or return.

This was particularly effective within the GOS geometry classes, where it was discovered that it was possible to store any location within the UK to millimetre accuracy using point objects formed from two 32-bit integers, rather than requiring doubles. The saving in overall database size was calculated to be approximately 40% to 55% - there are a *huge* number of points in the GOS!

Stored as GIF images the GOS data has been measured at 4 Terabytes in size. By judicious use of strings tables, numeric type demotion and C++ bit-fields it was possible to significantly reduce the storage capacity. This not only reduces disc space but more importantly maximises the number of persistent objects brought into memory per page fetched, which increases runtime performance.

Technologies

Here we cover which technologies were used to implement the GOS and why.

XML

XML was used as the transport format because it is ideal for describing complex tree-like data structures and therefore perfect for describing the complex geo-spatial data output from the GOS.

XML is non-proprietary, standardised by the World Wide Web Consortium (W3C) and has wide acceptance across all industry sectors. These facts automatically provide a degree of 'future-proofing'. The two XML standards currently supported by the GOS are GML and SVG, both universal standards and both supported by numerous third party software vendors; a list which will only expand with time.

XML is also extensible. The idea of language extension is built into XML and this provides another type of future-proofing because the format of the XML output from the GOS can be changed to meet *any* conceivable future requirements.

XML formats are specified by a set of rules held in Document Type Definitions (DTD) or XML Schemas. These define which types of data elements appear in the data stream and their order. Any input or output from the GOS can be checked for compliance to its published DTD providing useful error checking. Also XML being an ASCII format makes the detection of data errors much simpler than is the case for binary formats. All this serves to improve the quality and reliability of the GOS.

ISO C++

C++ was the language of choice because of the very demanding scalability and performance requirements of the GOS. Java was considered and a prototype was even written, but language features, such as unpredictable garbage collection activities, the time taken to load data from disk into the VM, and the inability to really control the layout of Java objects within memory, conspired to make it unsuitable for a server of this size.

The language C++ follows ISO and ANSI standards, and has 'evolved' to solve the problems encountered on large, complex software projects. C++ supports several programming styles: object-oriented, procedural and generic, allowing programmers to develop from an abstract perspective 'close to the problem domain', such as the development of the bespoke indexing structures used by the GOS, without loosing the ability to really control the machine when necessary, such as the use of C++ bit-fields to compress the persistent storage model.

During development the team encountered many technical problems, as every computer project does, but using C++ allowed the developers to solve them all *from within the language* and in such a way that the code remained coherent and maintainable. Counter-intuitively, using the C++ language proved to be the low risk option.

ObjectStore

ObjectStore is a high-speed, highly scalable object database that can store C++ objects directly to disk as objects without unpacking them into relational tables. ObjectStore uses 'memory blasting', which is a process where pages of memory are mapped directly to and from disc. These pages of memory contain the C++ objects instantiated by the program. By mapping these memory pages in and out, the C++ objects thereupon are also saved and fetched from disk.

Relationships between C++ objects are often implemented using pointers. During memory blasting the pointers between objects are 'fixed up' or 'swizzled' so that referential integrity is maintained in the current context. Memory blasting forms the basis of the patented Virtual Memory Mapping Architecture (VMMA) and is extremely fast.

Physically within an ObjectStore database, these memory pages are stored in flexible 'storage areas' called *clusters*. As new memory pages are created these clusters grow to accommodate the new additions. The programmer can create and destroy clusters at will, and has full control over the placement of C++ objects within clusters. In this way the programmer can control the relative physical position of objects so that objects that are used together are fetched together on the minimum number of pages. A near-optimum 'locality of reference' for the GOS was established using the bounding boxes for each topographic object to determine in which cluster they should be created. This yielded a remarkable improvement in performance.

Using ObjectStore simplified the development process considerably. There was no relational schema to define, no mapping to define between tables and objects, no SQL code to write and no effort involved keeping these three things in sync. From the programmers perspective ObjectStore is just another C++ library.

Scaling and Performance

High performance is important from the business perspective. Performance opens up new markets that are not available even in principle to slower systems. For example, markets involving real-time feature serving or data streaming to mobile devices can only be considered by Ordnance Survey if the GOS can scale to meet demand.

The GOS exhibits near linear scaling characteristics. Installing and configuring the system onto 2N machines will double the performance of an installation onto N machines. This scaling characteristic is expected to continue to the point where N is large enough for the entire data set to remain resident in memory. The cost of scaling is low, because it does not require any development effort. The system is simply reconfigured, using a bespoke graphical tool developed as part of the project, and re-installed.

The GOS also scales down as well as up. It would be possible to install a full version of the GOS onto a single, albeit fairly large, PC with 300Gbytes of disk or network attached storage. In some situations a partial installation with a sub-set of the data, perhaps a single county or town, might be useful.

Current GOS statistics

Loading 230,000 tiles from scratch including all indexing and stitching across tile boundaries, (computationally a very expensive process), takes around 2 days. A full national query which covers the whole of the UK and generates 700 GBytes of GML currently takes 2-3 days, but tuning work is expected to reduce this further to around 1½ days. Note, this is achieved using the feature serving configuration of the GOS and could be even further reduced by using the bulk supply model. Finding a single object out of some 700 million features using a TOID takes less than a second.

Flexibility

The GOS was designed to support multiple output formats from a single underlying storage model. The output formats become alternative representations of the data held within ObjectStore and the GOS can easily supply different formats on a per request basis. From a reliability perspective, by deriving output from a single underlying storage model these alternative representations are more likely to be mutually consistent.

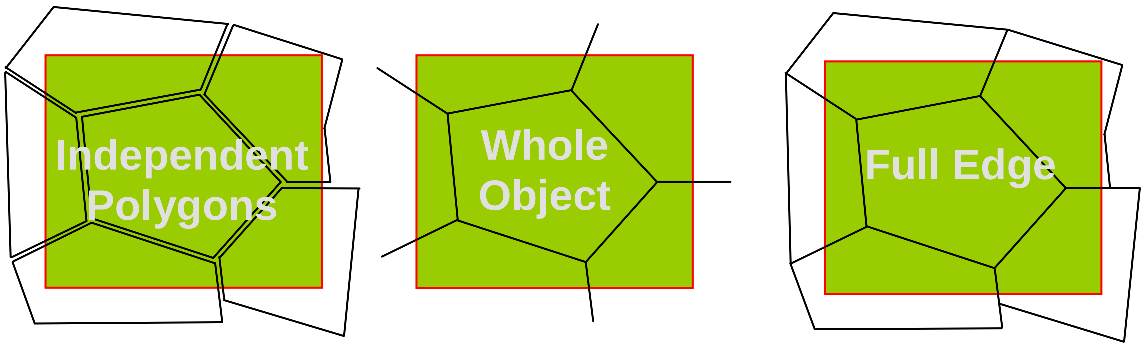

Fig 10. Example GML Output Formats

As mentioned above, the GOS stores *full topological information*; it knows that one garden is next to another and between is this particular fence. Any shape-only representations are simple derivatives of the underlying topological storage model.

There are two identified areas which will change in the system: the first is output formats, the second is the storage model. The C++ object model was designed with both these in mind; output format code uses the Visitor Pattern [1], the storage model uses the Bridge Pattern [1].

New output formats can be introduced into the system very quickly by writing a new visitor class. The GOS could produce any ‘streamable’ data format that may be required (e.g. GIF, DXF, CSV etc.) During development when it was decided to produce SVG directly from the GOS it took one programmer less than 3 days to produce working code and only another two weeks to bring this up to industrial strength. The GML specification continuously changed throughout the project and was closely tracked by a single programmer expending half a day per week.

Changes to the storage model can be introduced in-situ, while the system is online, without compromising 'locality of reference' and very quickly in parallel at full load speeds. Using the Bridge pattern allows 'changes' to existing classes to be implemented by the introduction of *new* classes. A sweeper program, that uses a simple variant of the visitor, finds all instances of an existing class and replaces each with an instance of the new class constructed from the existing object. The pointer held in the other side of the Bridge is then re-targeted at the new object.

Tool Integration

The GOS is a well defined component encapsulated behind an XML interface. Integration with other tools is flexible and controlled; flexible in that any set of objects can be queried or updated using XML, and controlled in that the National Topographic Dataset is not directly exposed to the vagaries of third party software.

ObjectStore and C++ are not visible outside the GOS. This means that clients can be developed with any technology and in any programming language. This arrangement also means that on-line and off-line clients are supported using the same mechanism. Both types of clients use the same XML interface for queries and for updates. This reduces the overall code base and means a reduced cost of ownership.

Project Risks

There are numerous areas of risk, here we look at four and how they have been mitigated in the GOS:

Changing Business Requirements. As Ordnance Survey data is utilised in new markets the requirements on the GOS are certain to change. Having a high performance system reduces the chance of failing to meet these future business requirements. Typically the demands on computer systems are always increasing, and the GOS has been designed to scale easily and cheaply. In addition, expanding into new markets or changing survey practice can result in new output formats or changes to GOS storage model. Both these are accommodated neatly and efficiently in the current system.

Data Errors. Ordnance Survey has had great problems with data errors introduced by third-party GIS editors in the past. These data errors typically require manual intervention and are very expensive to fix. Encapsulation behind an XML interface allows for central validation of updates. If data errors appear in the database it can only be as a result of failures in this central validation module. Once fixed there, the problem will be fixed for all updates irrespective of the source of the data. This will massively and systematically reduce the occurrence of data errors and help maintain the Ordnance Survey brand quality.

Technology Future-Proofing. Encapsulating the GOS behind an XML interface means that the shelf-life of the GOS interface is assured. XML is here to stay. However, the technologies used within the GOS will almost certainly be superseded at sometime in the future and can be more easily replaced, should this become necessary, because encapsulation guarantees that there are no dependencies outside of the GOS on the implementation details within the GOS. Any technology changes will be localised.

System Maintenance. The skills required to understand and maintain the GOS aside from Ordnance Survey core mapping and geo-spatial experience are: XML, C++ and ObjectStore. Both XML and C++ skills are widely available in the labour market. ObjectStore is rarer but is relatively easy to learn, particularly for team members with some prior C++ experience.

References

1. Gamma, E., et al. Design Patterns: Elements of reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.