UNIVERSITY OF OSLO
Department of Informatics

Optimizing spatial cache performance for mobile applications

Cand Scient thesis

Henrik Solgaard

2nd May 2007

Contents

Chapter 1

Introduction

Maps have for a long time been an important part of many websites. The processing power of a modern computer makes it possible to provide advanced map functionality even inside a web browser.

Maps are even more useful when they can be accessed anywhere. Most existing map systems for mobile devices are based on simple static raster maps with limited possibilities for interaction. With recent developments in mobile computing technology, more advanced map functionality can also be implemented on mobile devices.

Map data are large and usually stored in databases to provide fast access to many users simultaneously. Mobile devices have limited bandwidth and the mobile data traffic may be expensive. The combination of low bandwidth and high latency in mobile networks can impair the user experience. It is necessary to minimize the network traffic and maximize the response time.

A common way to reduce the effects of network limitations is to use a cache to store previously fetched content to make it quickly accessible the next time it is needed. Caching spatial data is special because the data are two-dimensional. In this thesis, I will explore alternative implementations and study how parameters such as cache size, tile size and placement of the clipping function affect performance of a spatial cache used for a map application on a mobile device.

Chapter 2

Problem description

In this chapter, I will give a more detailed presentation of the subject for this thesis.

The term *Geographic Information System (GIS)* is frequently applied to geographically oriented computer technology [7]. It is also commonly used to refer to tools for analyzing geographic data, but in this thesis it will only refer to the technology in general.

The terms *geographic data* and *spatial data* are often mixed. To be precise, geographic refers to the Earth's surface, while spatial refers to any space [14]. In this thesis, I will mostly use spatial, but in some cases when I mention concrete data from the Earth, I will use geographic.

The subject for this thesis is related to search in spatial databases. More theory related to GIS and spatial data will be presented in chapter 4.

I define a mobile terminal to be a portable device with a screen, support for some kind of user interaction and a wireless network connection of any kind, which is not necessarily online at all times. Laptops, PDAs and smartphones are examples of mobile terminals.

2.1 The problem

In use-cases where spatial data is needed in a mobile environment, access to the spatial database is often a bottleneck. The amount of data required can be large, and mobile networking technologies are still slow compared to wired networks.

Caching is well known from web applications, where static images and text objects are stored by the web browser and sometimes also by a proxy server between the client machine and the web server.

However, for spatial data, these simple cache mechanisms do not work since the number of objects (points, lines etc.) prevent caching of single objects and

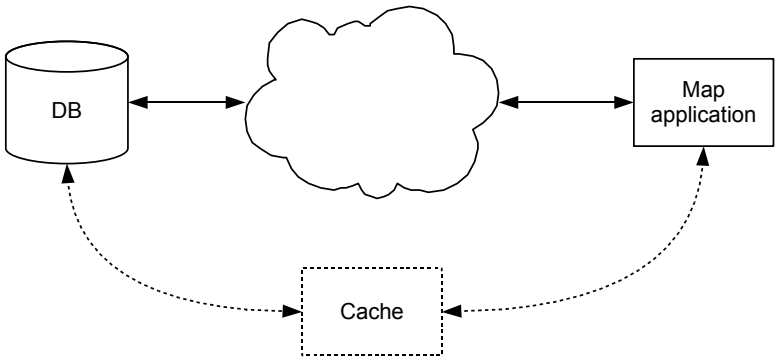

Figure 2.1: System overview

an aggregate structure such as a tile is needed. Caching is further complicated because the data domain is two-dimensional.

Thus, there is a need for a specialized spatial cache somewhere between the client application and the server as illustrated in figure 2.1.

2.2 The goal

To explore the problem area, I will develop a cache for use between a map client and a map database and explore how different parameters affect performance. I will study the effect of caching on the overall response time for the map client. In order to visually inspect the result of caching, and verify the result of the spatial search, I will also implement a map client, instrumented to find exact application performance. The map client will display maps with multiple layers and used for evaluation of how a cache improves responsiveness.

I want to find a reasonable size for the cache with a high probability that frequently accessed data are cached. The goal is not to create a perfect cache, but to observe the general effect of caching spatial data and have some background for discussion of how a cache can be useful in different situations.

Chapter 3

Related work

In this chapter, I will present some existing work related to caching in databases and caching of spatial data in particular. I will not cover the general concept of caching, used in processors, memory systems, disk systems and for web clients and proxies, but focus on caching spatial data.

3.1 Database caching

Standard database systems such as Oracle Database and the open source alternative PostgreSQL perform internal caching to speed up repeated search operations. They can also take advantage of the underlying operating system's disk caching. However, the database is usually centrally located and central caching does only speed up the query until a certain level. Local caching is needed in order to mitigate transmission delays and leverage local terminal capabilities (memory, cpu).

3.2 Web based map services

Existing map services are mostly raster based with pre-generation of raster tiles from vector or raster data on the server side. Examples include popular services such as Google Maps and Microsoft Local Live. Open Geospatial Consortium's Web Map Service (WMS) Implementation Specification [19] defines a standard format for requsts and responses to/from web based map services.

3.3 Caching spatial data

The client side of web map services implent caching of varying degree. Some clients fetch new data every time the view changes, while others keep previously

fetched data in memory to speed up repeated access to the same areas.

The WMS Tiling Client Recommendation (WMS-C) [1] from the Open Source Geospatial Foundation (OSGeo) is a Non-binding recommendation paper for the authors of WMS clients. It is based on the WMS 1.1.1 specification, but provides extensions and constraints to make the clients use the WMS servers in a way that better supports caching. The key is to divide the map into regular tiles. Standardization of parameters, parameter order, bounding box, etc. is used to increase the probability of repeating identical requests. This can improve the effect of caching both in the browser, in the web server and in proxy servers.

A WMS-C compliant client will improve cacheability with any WMS server, but for full effect, the extensions must also be implemented on the server side. TileCache [2] from MetaCarta Labs is WMS-C compliant server-side cache. It can be used with GeoServer [3] and MapServer [4], two major open source map servers.

3.4 Advanced map client

Simple raster based solutions are not always sufficient. When more dynamic maps are required, vector maps may be necessary. Brinkhoff [1] suggests a web based solution with a Java applet running in a web browser. However, browsers on some mobile terminals have limited functionality. On these platforms, a dedicated application may be the only alternative for displaying advanced maps.

3.5 Map rendering library

In order to experiment with caching of spatial systems, I have access to a comprehensive map rendering library called SmartMap developed by the company Faster Imaging. SmartMap can be used to render maps from raster and vector data. For this thesis, only the vector functionality will be used.

Capabilities of the SmartMap library include:

- Rendering lines with different colors, line widths, line styles and with or without anti-aliasing.
- Rendering poygons with different colors and fill styles.
- Rendering text positioned by points or lines.

[1] http://wiki.osgeo.org/index.php/WMS_Tiling_Client_Recommendation
[2] http://www.tilecache.org/
[3] http://www.refractions.net/geoserver/
[4] http://mapserver.gis.umn.edu/

- Zooming and panning within a dataset.
- Transforming vector data in memory between different projections.

The MapBrowser application developed for this thesis uses the SmartMap library to render maps from vector data. Zooming and panning is not needed because the application fetches a new set of vector data every time the position or size of the view is changed. Coordinate transformation is not used.

Chapter 4

Theory

In this chapter, I will detail key aspects of spatial data such as features, geometry types, storage formats, database interfaces and how spatial data can be retrieved.

4.1 Basic GIS terminology

Many types of data models are used in GIS systems [14]. This thesis will focus on a vector model because of its storage efficiency and its ability to render high quality maps at arbitrary scales.

A geographic entity encoded using the vector data model is called a feature [14]. There are two main categories of feature representations: simple features and topologic features. Both types are illustrated in figure 4.1.

4.1.1 Simple features

A simple feature is represented by a geometric object. I will refer to the geometric part of a feature as a geometry. In a simple feature data model, every geometry belongs to only one feature.

Advantages

- Easy to create and store.
- Easy and fast to read and render.

Disadvantages

- Lack of connectivity relationships.

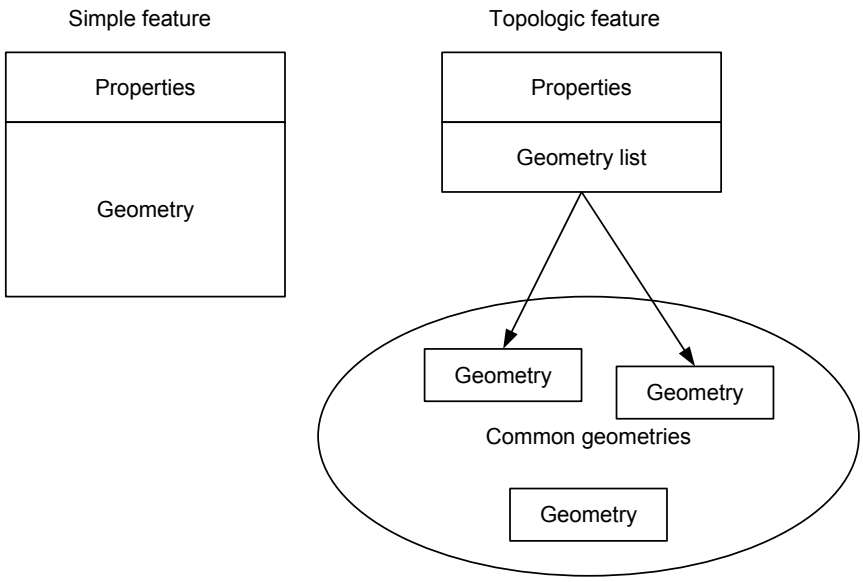

Figure 4.1: Simple and topologic features

- Limited possibilities for spatial analysis.
- When features are adjacent, their edges are duplicated.

4.1.2 Topologic features

In a topologic data model, there is a common set of geometries. Each feature contains a list of references to geometries which together form the geometric representation of the feature.

Advantages

- No duplication of geometries.
- Enables various kinds of spatial analysis.

Disadvantages

- Complicated to read.
- Even more complicated to create.

4.2 Geometry types

Three categories of geometry types are usually used in maps, Pnoint, Curve and Surface. ISO 19107 [10] developed by the Open Geospatial Consortium provides the following abstract definitions. A geometric primitve is defined as "a geometric object representing a single, connected, homogenous element of space".

Point 0-dimensional geometric primitive representing a position.

Curve 1-dimensional geometric primitive, representing the continuous image of a line.

Surface 2-dimensional geometric primitive, locally representing a continuous image of a region of a plane.

In the following, I will refer to these categories of types as point types, line types and polygon types respectively.

Some concrete implementations of these types will be presented in section 4.4.

4.3 Coordinate systems

Because the earth is round and maps on screen or paper are flat, coordinates on the surface of the earth must be transformed to a Cartesian coordinate system [14]. This mapping is called a projection. All projections necessarily distorts the real world, so each type of projection must make tradeoffs on what kind of properties it preserves and what it distorts.

In addition to Cartesian coordinates, map coordinates sometimes include a third and/or a fourth dimension. In this case, the third dimension is elevation above a reference height (usually sea level) and the fourth dimension (or third if there is no elevation) is a measured value for the point.

4.3.1 Universal Transverse Mercator

The Universal Transverse Mercator (UTM) system is a projection created by wrapping a cylinder around the poles and projecting each point on the Earth's surface onto the cylinder [14]. There are 60 zones in the system, and each zone corresponds to a half cylinder wrapped along a particular line of longitude, each zone being 6 degrees wide. Norway is covered by zones 32-36. For datasets that cover the whole country, zone 33 is usually used. The dataset used in this thesis was delivered in UTM zone 33, but was converted to UTM 32 because almost all features are within this zone.

4.4 Vector map formats

Many formats exist. Common simple feature formats include ESRI Shapefile and OpenGIS Consortium's Well Known Binary (WKB), Well Known Text (WKT) and Geography Markup Language (GML). Topologic formats are less common because they are more complicated to create and use and because the topology information in many cases is not needed. An example of a topologic format is the Vector Product Format (VPF) developed by National Geospatial-Intelligence Agency (NGA).

Some of these formats are described in more detail below.

4.4.1 ESRI Shapefile

Shapefile is a format defined by Environmental Systems Research Institute (ESRI) [3]. It is frequently used for distribution of map data, and most GIS tools can import and/or export Shapefiles. I will use Shape or Shapefile depending on the context to refer to the Shapefile standard and a file in the Shapefile format. A Shapefile (.shp) consists of a number of records, each containing a geometry. It is always accompanied by an index file (.shx) which points to the start of each record in the shapefile, and a dBASE database file (.dbf) which contains non-spatial data for the geometries in the Shapefile. The index file is not necessary when reading a Shapefile sequentially, but it speeds up random access.

A Shapefile can also be accompanied by a file with projection information (.prj) and two spatial index files, (.sbn) and (.sbx) in a proprietary format which can only be read by ESRI's own GIS products.

The Shapefile specification contains 14 geometry types. The two-dimensional types are listed in table 4.1. There are additional types for three-dimensional points and points with an extra measured value which can have a special "no data" value. Each Shapefile contains geometries of only one type with one exception: There is a special Null type with no geometric data. This can be present in Shapefiles of all types and is primarily used as a temporary placeholder.

4.4.2 Open Geospatial Consortium formats

The Open Geospatial Consortium has published a number of GIS-related specifications. The Abstract Specifications, consisting of a number of topics, provides a conceptual foundation for the other specification. ISO standard 19107 [10], which was mentioned above, is identical to one of the topics in the Abstract Specification.

The Simple Feature Access specification [17] defines the Well Known Binary (WKB) format. It is a binary geometry format which includes all geometry types

Type	Description
Point	Consists of a pair of double-precision coordinates in the order X,Y.
MultiPoint	Represents a set of points.
PolyLine	An ordered set of vertices that consists of one or more parts. A part is a connected sequence of two or more points. Parts may or may not be connected to one another. Parts may or may not intersect one another.
Polygon	Consists of one or more rings. A ring is a connected sequence of four or more points that form a closed, non-self-intersecting loop. A polygon may contain multiple outer rings. The order of vertices or orientation for a ring indicates which side of the ring is the interior of the polygon. The neighborhood to the right of an observer walking along the ring in vertex order is the neighborhood inside the polygon. Vertices of rings defining holes in polygons are in a counterclockwise direction. Vertices for a single, ringed polygon are, therefore, always in clockwise order. The rings of a polygon are referred to as its parts.

Table 4.1: Geometry types in the Shapefile format

Type	Description
Point	A 0-dimensional geometric object. It represents a single location in coordinate space. A Point has an x-coordinate value, a y-coordinate value. If called for by the associated Spatial Reference System, it may also have coordinate values for z and m.
LineString	A 1-dimensional geometric object usually stored as a sequence of Points with linear interpolation between the Points. Each consecutive pair of Points defines a Line segment.
Polygon	A 2-dimensional geometric object defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the Polygon. Each boundary is a closed (starts and ends in the same point) and simple (does not intersect itself) LineString.
MultiPoint	A collection of Points. The Points are not connected or ordered in any semantically important way.
MultiLineString	A collection of LineStrings.
MultiPolygon	A collection of Polygons. The interiors of any two Polygons in a MultiPolygon may not intersect. A MultiSurface whose elements are Polygons.
GeometryCollection	A collection of geometric objects of any of the types listed in this table. The geometry objects in a GeometryCollection doesn't have to be of the same type.

Table 4.2: Geometry types in the WKB format

commonly used in maps and some types that are only used as part of the definitions of the other types. All instantiable types are listed in table 4.2.

4.4.3 Compatibility

In this thesis, I will use "Shape PolyLine" and "Shape Polygon" when talking about Shape geometries and "WKB LineString" and "WKB Polygon" when talking about WKB geometries to avoid confusion.

The definitions above shows that different implementations of the same abstract types can have slightly different semantics. This can cause problems when when converting from one format to another. The map data used in this thesis were delivered as Shapefiles and were converted to WKB with the PostGIS

database import tools.

A point consists of a single coordinate pair in both Shape and WKB, so this conversion is trivial. A Shape PolyLine can contain multiple disjoint sequences of line segments. A WKB LineString can only contain one continuous sequence of line segments. This can be handled by storing all Shape PolyLines as WKB MultiLineStrings. A Shape Polygon can similarly contain multiple outer rings which is not allowed in a WKB Polygon. This can be handled by storing all Shape Polygons as WKB MultiPolygons.

Another difference between different geometry formats is how points are ordered in the rings of a polygon. In a Shape Polygon, the points in an outer ring must be listed in clockwise order. Points in an inner ring must be listed in counterclockwise order. In the WKB specification, the order is unspecified. This means that rings can be converted directly from Shape to WKB without worrying about the order of the points.

4.5 Spatial data retrieval

4.5.1 Indexing

Spatial data requires a different kind of indexing than normal relational data because it is two-dimensional. Many types of spatial indexing techniques have been developed. Research has shown that even a basic spatial index is enough to cause very significant improvements in spatial data access [14]. There are three main categories of spatial indexes: grid indexes, quadtrees and R-trees. The most common type of spatial index in spatial databases is the R-tree [8].

A spatial search selects from a table all features that are in a specified spatial relationship with a query geometry. This can be done in two steps. They are called primary and secondary filtering [20] or filter step and refinement step [21].

Primary filtering

Primary filtering uses only the spatial index to select candidates. This makes it inaccurate and can cause it to return too many hits, but never too few.

Secondary filtering

Secondary filtering is typically used in combination with primary filtering, but can also be used alone, but with much lower performance because the spatial index is not used. Each geometry is compared to the query geometry with a function checking the spatial relationship between the two geometries.

4.5.2 Clipping

Many operations can be applied to the result of a spatial query. One of them is the Intersection function which returns a geometry that represents the point set intersection with another geometry [17]. If a rectangle has been used for spatial selection, the same rectangle can be used with the intersection function to remove all parts of all geometries that are outside the rectange. I will refer to this as clipping. This will reduce the size of the returned data without affecting the data inside the requested rectangle.

4.6 Caching

A cache is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch or compute relative to reading the cache [1]. When a requested data item is found in the cache, it is referred to as a cache hit. If it is not found, the cache has to fetch it from the original location. This is referred to as a cache miss. The cache hit ratio is the total number of cache hits divided by the total number of cache requests.

4.6.1 Cache replacement algorithms

When there is not enough space to insert more data in a cache, some old data must be deleted. This is called cache replacement. Cache replacement algorithms usually try to maximize the cache hit ratio by attempting to keep the data items which are most likely to be referenced in the future in the cache [23]. A simple and common strategy is to delete the least recently used (LRU) items until there is enough space.

More sophisticated cache replacement algorithms may also try to minimize the cost of cache misses. They will try to keep data items which took a long time to fetch.

Special replacement algorithms for spatial data have been developed. Brinkhoff [2] demonstrates that a spatial page-replacement algorithm outperforms LRU for some distributions, but not for all investigated query sets.

4.6.2 Cache consistency algorithms

Cache consistency algorithms enforce appropriate guarantees about the staleness of cached data items [23]. The consistency requirements can be very different for different types of caches. Cache consistency algorithms for database systems

[1] http://en.wikipedia.org/wiki/Cache

usually enforce strong consistency (i.e. no stale data returned to clients) while web servers can have more relaxed consistency requirements.

4.6.3 Pre-fetching

A common way to speed up the response time of a cache, is to try to predict which data items will be requested in the future and fetch them before they are needed. Kang [12] proposes a tile-based combined pre-fetching and cache replacement algorithm. It uses the global access pattern of all users to find probable future tile transitions. Tiles likely to be accessed in the future are pre-fetched. Tiles unlikely to be accessed in the future can be replaced when the cache is full because transition probabilities are already calculated.

Chapter 5

Architecture

In this chapter, I give an overview of the architecture consisting of a map database, a map application and a cache.

This is a client/server architecture where the map application is the client and the map database is the server. The problem I wanted to solve was that bandwidth on a mobile terminal is limited and sometimes expensive. I wanted to introduce a cache between the client and the server to reduce the amount of network traffic between them.

The map database is an ordinary spatially enabled database system with no custom functionality, accessed through a standard database API. The client and the cache was developed specifically for this thesis.

5.1 System overview

One of the choices I had to make was where to insert the cache. It could be located anywhere between the client and the server. The possible locations can be divided into three groups. The cache can be part of the central system along with the database, it can be placed at the edge of the network the client is connected to, or it can be included in the mobile terminal. The following sections will describe each of these alternatives in more detail. One of them was selected for implementation. Based on the testing and analysis of this implementation, the two others are discussed in chapter 9.

5.1.1 Centralized cache

This architecture is illustrated in figure 5.1. The cache is located on the same server as the database. The client will connect via the internet either directly to

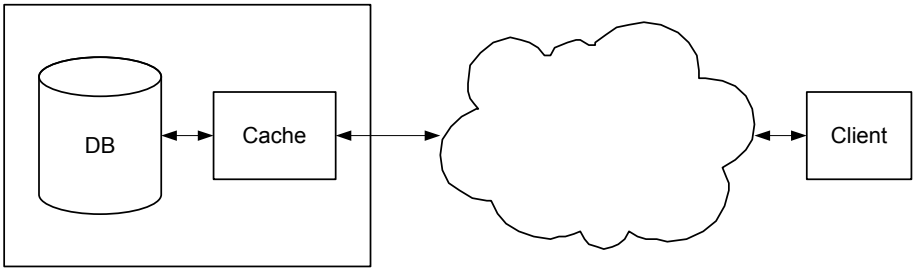

Figure 5.1: Centralized cache

the cache or to a frontend service. With this architecture, it is even possible to systematically cache all data before making the cache available to clients.

Advantages

- Spatial searches can be computationally intensive. A cache will reduce the CPU load.
- Doesn't use memory resources in the clients.
- Powerful hardware can be used for the cache.
- Storage is less expensive on a server.

Disadvantages

- Databases already have built-in caching, so the added effect may be small.
- The cache may compete with the operating system's caching of disk pages if the memory is limited.
- Response time will be higher than with a local cache.

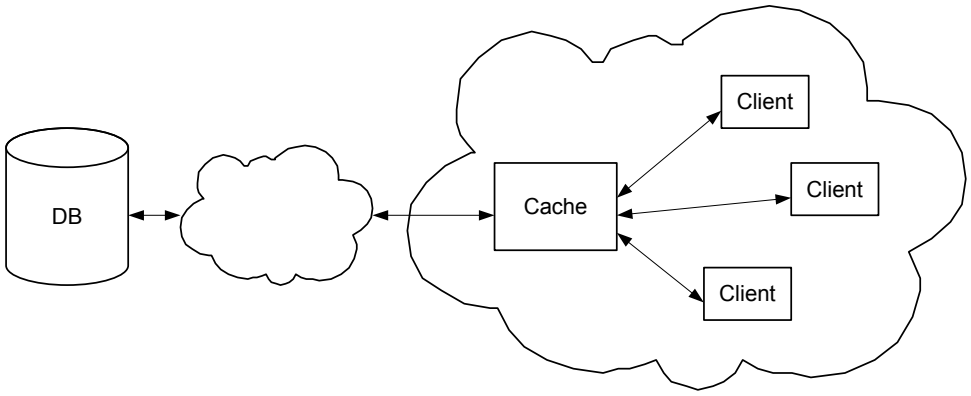

Figure 5.2: Distributed cache at the edge of the network

5.1.2 Distributed cache at the edge of the network

This architecture is illustrated in figure 5.2. It is typical for applications where a number of potential users are located at the same place such as in a train or on a ship with a wireless network, and we want to leverage spatial search with minimal access to the central database due to limited or expensive bandwith (e.g. satellite connection). I will call this a subnet cache because it serves users in a single subnet. If the client has too little memory to allow caching, this is a good solution.

Advantages

- Doesn't use memory resources in the clients.
- Powerful hardware can be used for the cache.
- Cached data can be available when the connection to the remote server is down.
- The users in a subnet may be interested in the same content which may be different from the users in other subnets.
- Response time will be lower than with a centralized cache due to lower network latency.

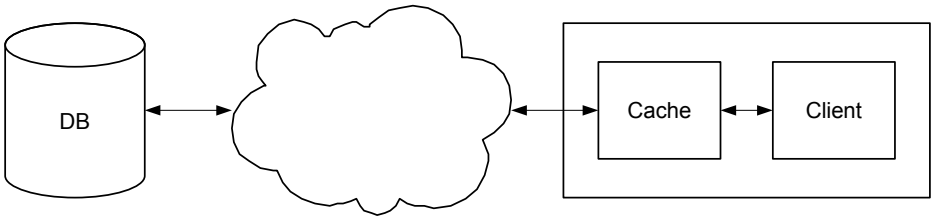

Figure 5.3: Distributed cache in the mobile terminal

Disadvantages

- Response time will be higher than with local cache.
- Requires the client to communicate with the local network. Not all terminals may have this capability.

5.1.3 Distributed cache in the mobile terminal

This architecture is illustrated in figure 5.3. The cache is installed locally in the mobile terminal and uses only the terminal's memory. As mobile terminals are becoming more and more powerful, this may be an attractive solution.

The network connection to the database is the bottleneck in this architecture. It is important to minimize the size of the data sent over the network.

Advantages

- Very quick response when requested data are in the cache.
- Cached data can be available offline.

Disadvantages

- Cache capacity is limited by device resources.
- More expensive memory in the mobile terminal.

5.2 Constraining the problem

I have decided to concentrate on the distributed cache in the mobile terminal since this is the most interesting of the architectures due to the popularity of mobile terminals such as laptops, SmartPhones and PDA devices with significant local resources. In order to focus on the cache performance, I have decided to use a standard laptop as the target mobile teminal used in the performance tests.

5.2.1 System architecture

Because the cache is local, it was developed as a static library and linked into the clients. The clients control the initialization of the cache and specifies which database to connect to, and which database table to fetch data from.

5.2.2 Single table

A cache can only contain data from one table. The map application will create multiple caches in order to display multiple map layers.

5.2.3 Single projection

The test dataset is in UTM 32. The cache does not support projection transformation on the fly. If the system should cover a larger geographic area like Europe, the UTM projection could not be used because of too much distortion.

5.2.4 Geometry types

I only test with lines and polygons because the rendering library has no point primitives. Text can be positioned by a point layer, but because the rendering library is intended for a slightly different use, text is not rendered in a readable way.

5.2.5 Direct database access

In a real scenario, the client application would most likely not communicate directly with the database because of security issues and other technical considerations. A web service or some other frontend would handle the requests from the application and perform the appropriate database query. For simplicity, this extra layer has been omitted in this architecture.

5.3 Database

The database is an essential part of the architecture. The choice of a database system was limited by the requirement of spatial support. The database system had to support geometric data types, spatial indexing and search, and clipping. The most common commercial option is Oracle Database. Except for clipping, all required spatial functionality is part of the standard database system. An optional extension called Oracle Spatial includes clipping support.

The cost of a commercial database system like Oracle Database forced me to find an alternative. There are two major open source database systems available: MySQL [1] and PostgreSQL [2]. MySQL has support for geometric types and spatial indexing and search, but not clipping. PostgreSQL in itself has no spatial support, but the third-party PostGIS [3] extension adds all the functionality I needed. Because it is also free to use, this was the obvious choice.

5.3.1 Interface

I used the Open Database Connectivity (ODBC) API to communicate with the database. This is a standard API for database access developed by Microsoft and supported by most database systems. I made my own abstraction layer which in theory should make it easy to add support for other spatial database interfaces such as Oracle Spatial[4] or ArcSDE[5].

5.3.2 Indexing

PostGIS has two spatial index implementations. Both are based on R-trees. The index used in this thesis, is an R-tree implementation on top of a GiST [13] index.

PostGIS has no way way to do only primary filtering. It has the *"&&"* operator which uses the spatial index for a primary filtering, but also eliminates geometries whose bounding box is disjoint from the query's bounding box. For secondary filtering, it supports all the spatial relationship functions specified by Open Geospatial Consortium's specification of an SQL interface for simple features [18].

[1] http://www.mysql.com/
[2] http://www.postgresql.org/
[3] http://postgis.refractions.net/
[4] http://www.oracle.com/technology/products/
[5] http://www.esri.com/software/arcgis/arcsde/

Chapter 6

Implementation

In this chapter, I will describe the design and implementation of the cache, the client application and the test framework. Figure 6.1 illustrates how the different parts of the system are related to each other. The grey parts were already available. The white parts were impemented as part of the work with this thesis.

6.1 Development tools

The cache, the MapBrowser application and the test framework was developed in C++ on Microsoft Windows XP Service Pack 2 with Microsoft Visual Studio 6.0 and Microsoft Platform SDK for Windows Server 2003 Service Pack 1. The test executables were built in Release mode with default optimizations.

6.2 Data format

I needed a data format for internal representation of geometries with attributes. I identified some essential requirements, evaluated some existing formats and eventually decided to create a new format specifically for this cache implementation as explained below.

6.2.1 Requirements

I wanted the data format to meet all or most of the following requirements:

- It must be general enough to support different database backends. Specifically, it must allow line geometries with multiple disjoint lines and polygon objects with multiple outer rings, not only because this is allowed in Shape

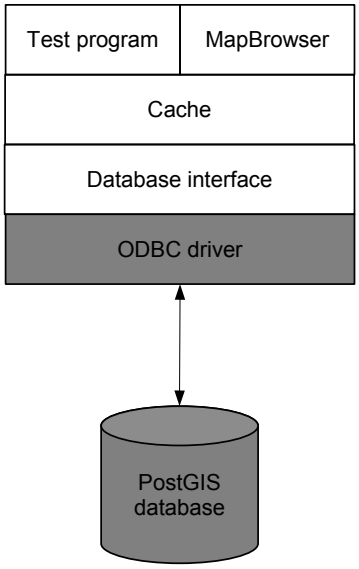

Figure 6.1: System overview

which is a common format for distribution of map data, but also because clipping can split geometries into multiple disjoint parts.

- Orientation of polygon rings must not be significant. This is because ring orientation is insignificant in WKB which is a common standard for spatial database interfaces.
- It must support text attributes which can be used as labels in a map.
- It must support at least one non-spatial attribute per geometry in addition to the text attribute. This can be used to control the style of rendered items (e.g. color or line width).
- It must be possible to move and copy records fast without having to see inside them. This means that the record size must be stored in a fixed location inside the record.
- Alignment of data items in memory is important for performance and required on some hardware architectures [15]. All data items must be aligned.

6.2.2 Existing formats

Before I decided a define a new format, I evaluated some existing formats to see if they could meet the requirements above. Textual formats like Well Known Text (WKT) [17] and Geography Markup Language (GML) [16] were not considered because textual representation is less efficient than binary in terms of space and processing time.

The following formats were evaluated.

Shapefile

The Shapefile format was described in chapter 4.

This format fails to meet the alignment requirement, and the orientation of rings is significant. It has a complete separation of spatial and non-spatial data. This makes it less useful as a format for geometries with associated non-spatial attributes.

Well Known Binary

Well Known Binary (WKB) was described in chapter 4.

This format also fails to meet the alignment requirement. The total record size for some geometry types is not easily accessible. A more serious shortcoming is

that it has no support for embedded non-spatial attributes. This makes it unsuitable for use in the cache.

6.2.3 New format

Because I was unable to find a format that met my requirements, I defined a new format which I call GeoCache Area (GCA). Appendix A contains a detailed description and illustrations.

The format contains the following structures:

Feature record Representation of a single map feature. It contains the coordinates of a geometric object, an optional text string and another optional non-spatial attribute.

Area record A sequence of feature records. It starts with the number of geometries and ends with an end-of-record marker.

Cache line This is an area record with more data and functionality. It contains the following information:

- Time of creation.
- Time of last update.
- Time spent fetching the data from the database.
- Position in the 2D tile index (described later).
- Request number.
- Pointer to previous and next cache line in the sorted cache line list (described later).

This format meets all the requirements above at least partially as explained below.

Multi-types

Like in WKB, there are special types for geometries with multiple disjoint lines or multiple disjoint outer rings. I will refer to them as multi-types.

Ring orientation

The orientaton of rings is by definition not significant.

Text attribute

Each geometry can contain a text string which can be used as a label in the map.

Other non-spatial attribute

Each geometry can contain a non-spatial attribute in addition to the text attribute. The available types are listed in table A.4 .

Fast handling of records

Geometry records have their size stored in a fixed location. This makes it possible to move and copy records very quickly.

Alignment

The format has been designed to ensure alignment for all data items up to 4 bytes.

- 4-byte alignment of the first feature record in an area record is ensured because it is only preceded by a 4-byte field. Alignment of the following feature records is achieved by inserting extra bytes of padding after each feature record.
- Inside a feature record, all fields before the non-spatial attribute and the label are either 4 or 8 bytes, thereby preserving 4-byte alignment.
- The non-spatial attribute follows next. It can have any size from 1 to 8 bytes, which means that nothing is known about the alignment of fields after it.
- The last field is the label. Because it consists of 1-byte characters and a 1-byte length, it has no alignment requirements.

6.3 Cache design

The internal data structures are illustrated in figure 6.2. The following sections describe their functions and relationships.

6.3.1 Tiles

It is infeasible to cache individual features, so I had to form larger units of features. A simple method is to logically partition the space of data into rectangles of equal size. Each rectangle is called a tile [12]. Data are fetched from the database

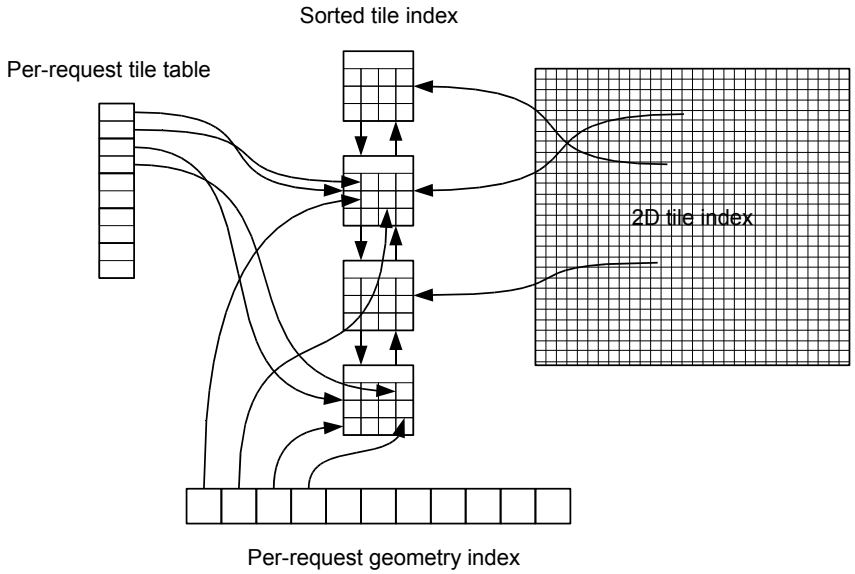

Figure 6.2: Cache internals

in units of one tile. This is efficient because spatial indexes are also based on partitioning the space into rectangles.

6.3.2 2D tile index

The 2D tile index is a two-dimensional array of cache line pointers. There is one pointer for each possible tile. This array is used to quickly determine whether a specific tile is in the cache. If a tile is present in the cache, the corresponding pointer points to the cache line object representing this tile.

6.3.3 Sorted tile index

All cache line objects are connected in a doubly linked list. This list is ordered with the most preferred candidate for removal first and the least preferred last. The ordering is determined by the cache replacement algorithm being used. This is explained later.

6.3.4 Per-request tile table

This table is specific for each request. Each element in the table contains a pointer to a cache line which contains a tile covered by the request. It also contains a pointer to a geometry inside the tile.

6.3.5 Per-request geometry index

This index is specific for each request. Each element in the index points to a geometry which will be included in the response.

6.4 Cache algorithm

Pseudo code for the cache algorithm is presented in figure 6.3.

6.4.1 Clipping

Clipping can be done in three places:

- By the database.
- By the cache.
- By the application using the cache.

An advantage of doing it in the database is that the amount of data traffic between the cache and the database is reduced. This is important if the bandwidth is low or the data traffic cost is high.

6.4.2 Merging tiles

When a request intersects multiple tiles, the geometries from the tiles must be merged into a common area record.

Duplicate elimination

If clipping is enabled, all geometries will be unique and there is no need for eliminating duplicates. However, if clipping is disabled, geometries will be included in multiple tiles if they cross the tile borders, and duplicates must be eliminated in the merging process.

The geometries in a tile are sorted by geometry id. This means that the tiles have a common ordering of geometries. Duplicate elimination is done by a variant

of phase two of the Two-Phase, Multiway Merge-Sort algorithm [6]. It uses the per-request tile table (see figure 6.2) to store a pointer into each tile. The pointers are initialized to point to the first geometry in each tile. In each iteration, the target of each pointer is checked, and the geometry with the lowest geometry id is remembered. If any other geometries are found with the same id, the pointers for these tiles are advanced to the next geometry in the tile. At the end of each iteration, a pointer to the geometry with the lowest id is added to the per-request geometry index (see figure 6.2), and the size of the geometry is added to the total geometry size.

When all pointers have reached the end of its tile, a buffer with space for all the geometries is allocated, and all geometries in the per-request geometry index are copied into the buffer.

Filtering

Geometries that don't intersect the request rectangle are excluded from the response. For points, this is simply a matter of checking if the point is inside the request rectangle. For lines, the bounding box of the line is calculated and tested for intersection with the request rectangle. For polygons, it is enough to calculate the bounding box of the outer ring(s) because this is guaranteed to contain all the inner rings. For multi-types, a common bounding box is calculated for all the parts.

A potential optimization has been discovered but not tested. It is possible check in advance which tiles are fully contained in the request rectangle. All geometries from these tiles are guaranteed to intersect the request rectangle and don't need to be tested.

6.4.3 Bypassing the cache

If a request covers a large area and thus intersects many tiles, the cache overhead can be so high that it is more efficient to query the database directly, depending on how many of the tiles that are already cached. More importantly, the total data size of the tiles intersected by a request rectangle must not exceed the maximum data size for the cache.

This could be solved by setting a maximum size for requests handled by the cache, and bypass the cache if a request is larger. However, in a real system, a cache will usually be used only for a limited range of scales, and another cache with less detailed data and a larger tile size will be used for larger requests.

In this cache implementation, the cache will never be bypassed. The request rectangles in the tests are assumed to be within an acceptable range.

6.4.4 Cache replacement

When the cache is full and another tile is about to be inserted into the cache, one or more exisiting tiles must be dropped. This is called cache replacement. The selection of candidates for replacement is an important topic, because the choices made here can have a great impact on the cache's performance.

Replacement is triggered by the function which is used to allocate space in the cache for a new cache line. If the available space is less than the requested size, existing tiles are dropped until there is at least 10% of the total data size available. This value is chosen somewhat arbitrarily. More experimentation is necessary to find an optimal value.

Priority

The cache lines are sorted according to a replacement priority. When the cache is full, cache lines are removed from the start of the list until there is enough free space. The replacement priority can be based on a number of available values:

- Time of last access.
- Data size.
- Time spent fetching the tile from the database.
- Number of accesses. (Not available, but easy to implement.)
- A combination of the above, either in a specified order or the sum of the attributes, each multiplied by a weight.

The sorting order is implemented by an overridable comparison method in the cache class. A subclass of the cache class can override this method to get a different ordering. The cache lines are sorted in a doubly linked list based on this comparison function.

Currently, cache lines are sorted by time of last access. Cache line insertion is optimized for this sorting order. With another order, cache line insertion may be less efficient.

Locking cache objects

When the cache is full, some old cache lines must be dropped. But it is important that the cache doesn't drop a cache line which is needed by a request in progress as this will cause unnecessary delays and possibly even deadlocks. Because this is a single-user cache where only one request will be processed at a time, the solution is simple. At the start of each request, the cache tags all cache lines that

are cached and that will be used by this request with a number which is unique for each request. All cache lines created for this request are tagged with the same number. The cache will not drop cache lines whose tag is identical to the current request number. The tag is simply a number which is set to 0 when the cache is initialized and incremented for each request. When it wraps around, it is highly unlikely that any cache lines with low numbers still exists.

In a multi-user cache, this method will not work because multiple requests with different request numbers can be in progress simultaneously.

6.4.5 Cache consistency

Cache consistency is usually not an issue in map systems because the data are static.

Expiration

When a cache line is created, the time of creation is stored in the cache line structure. This makes it possible to force cache lines to be discarded at a certain age. In this cache implementation, no expiration mechanism has been implemented.

6.5 Tile size

Selecting a tile size involves making a tradeoff between data redundancy in the database communication and overhead in tile retrieval and administration. A small tile size reduces the amount of data redundancy because the tile borders will better match the request rectangle, but it also increases the number of database queries and the tile administration overhead. If clipping is disabled, a small tile size will also increase the amount of duplication of lines and polygons across tiles. A large tile size on the other hand, increases the amount of redundancy, but reduces the number of database queries and the tile administration overhead. If clipping is disabled, it reduces the amount of duplication of lines and polygons across tiles.

6.6 Database connection

6.6.1 Queries

The contents of a layer is defined by an SQL query. This query is required when initializing the data source which will be used when initializing the layer's cache. If clipping is enabled, the query must contain the appropriate clipping statement. When the cache fetches a tile from the database, it appends the tile's bounding box

```
Client requests an area from cache.
If caching is enabled:
        Translate the area coordinates to tile index values.
        Create an array to hold tile pointers, one for each intersecting tile.
        For each intersecting tile:
                If already in cache:
                        Update the tile's last-used value.
                        Tag the tile with request number.
                        Move to sorted position in sorted tile index.
        For each intersecting tile:
                If not in cache:
                        Delete some existing tiles if the cache is full.
                        Create a new tile.
                        Tag the tile with request number.
                        Insert the tile into the 2D tile index.
                        Fetch data into the tile.
                        Update the tile's last-used value.
                        Insert the tile into the sorted tile index.
                Insert the tile into the array of intersecting tiles.
        Create an index of unique geometries from the intersecting tiles,
        excluding geometries that don't intersect the request rectangle.
        Copy all selected geometries to the output buffer.
Else:
        Fetch the whole area directly from the database.
        Convert the data to output format.
```

Figure 6.3: Pseudo code for the cache algorithm

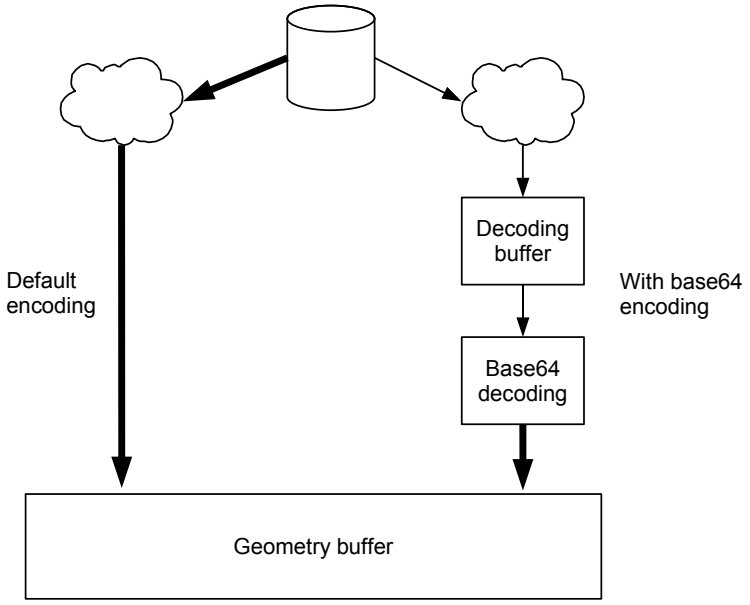

Figure 6.4: Database response with and without Base64 encoding

to the query, using the *"&&"* operator. As explained in chapter 5, this operator takes advantage of the spatial index. An "ORDER BY" clause sorting the result by geometry id is also appended to the query because the duplicate elimination algorithm requires that geometries are sorted.

6.6.2 Compression

Base64 encoding [11] of the data was implemented for reasons explained later in chapter 8. It was implemented by creating a new buffer and reading data into this instead of the geometry buffer when compression is turned on. Then the encoded data are decoded into the geometry buffer as illustrated in figure 6.4. This required a minimum of code changes. The compression can be enabled and disabled with a parameter to the initialization function.

The term compression needs some clarification. Base64 is not a compression algorithm, but a textual encoding of binary data which takes more space than the binary representation. But if the data is not encoded explicitly, the database protocol will automatically encode the data in an even less efficient way. The exact figures will be presented in chapter 8.

Name	Type	Description
szDSN	string	Database connect string.
szQuery	string	SQL query.
fClip	boolean	Clipping on/off. If this is true, the database clips the geometries along the area boundaries.
labelColumn	unsigned int	Number of the label column in the query if a text column is included.
attrColumn	unsigned int	Number of the non-spatial attribute column in the query if there is any.
fCompress	boolean	Compression on/off. If this is true, the geometries will be encoded in Base64 to avoid hex-encoding, which takes more space.

Table 6.1: GEO_DATASOURCE_POSTGIS::FInit

6.7 Cache interface

6.7.1 API

The cache API consists of two classes. GEO_DATASOURCE_POSTGIS represents a layer from a PostGIS database, and GEO_CACHE represents the cache. The most important methods of these classes are described below. In the following tables, all parameters to these methods are listed with name, type and description. The type GEO_GEOMTYPE is an *enum* type containing all supported geometry types.

GEO_DATASOURCE_POSTGIS

This class inherits from the abstract class GEO_DATASOURCE. It acts as a wrapper around a database connection and contains all the information necesssary to create a query for the layer it represensts. The following methods are important.

FInit Initialize the data source. This is specific for PostGIS. The parameters are listed in table 6.1.

GetArea Get feature data in GCA format for a specified area. The parameters are listed in table 6.2 This method is used by the cache to get data for a tile.

Name	Type	Description
x1	double	Lower left x-coordinate
y1	double	Lower left y-coordinate
x2	double	Upper right x-coordinate
y2	double	Upper right y-coordinate
type	GEO_GEOMTYPE	Geometry type (This is necessary in case clipping has created multiple geometries of different type. Then the data source needs to know which geometries to return. Why isn't this an initialization parameter?)
pLine	pointer	Pointer to an area record which will be filled with data.
pCache	pointer	Pointer to the cache which uses this data source.

Table 6.2: GEO_DATASOURCE_POSTGIS::GetArea

Cache GEO_CACHE

This class contains the cache implementation. The following methods are important.

FInit Initialize the cache. The parameters are listed in table 6.3

GetArea Get feature data in GCA format for a specified area. The parameters are listed in table 6.4

6.8 Viewer application

In order to test the usability of the cache, I developed a map application called MapBrowser. It had a number of purposes. First of all, I needed a tool to test and debug the cache while I developed it. Second, I needed a visual verification of the response from the cache. Third, I needed to get an initial feel for the response times and the effects of caching. Finally, I wanted to demonstrate the use of multiple caches, each contributing to one layer in a map.

6.8.1 Features

MapBrowser has the following features.

Name	Type	Description
pDataSource	pointer	pointer to a data source object.
x1	double	Lower left corner of world.
y1	double	Lower left corner of world.
x2	double	Upper right corner of world.
y2	double	Upper right corner of world.
geomType	GEO_GEOMTYPE	Type of geometry in this cache.
numTilesX	unsigned integer	Number of tiles we divide the world into horizontally. This is only used if tileWidth and tileHeight are 0.
numTilesY	unsigned integer	Number of tiles we divide the world into vertically. This is only used if tileWidth and tileHeight are 0.
maxSize	integer	Max space to use for cache data. 0: no limits, <0: no caching.
tileWidth	double	Optional horizontal tile size.
tileHeight	double	Optional vertical tile size.

Table 6.3: GEO_CACHE::FInit

Name	Type	Description
x1	double	Lower left corner of area.
y1	double	Lower left corner of area.
x2	double	Upper right corner of area.
y2	double	Upper right corner of area.
pDataSize	pointer	Pointer to variable where data size will be returned.
ppData	pointer	Pointer to pointer to data.
pInfo	pointer	The cache can return info in this struct.

Table 6.4: GEO_CACHE::GetArea

Zoom in/out

There are two toolbar buttons for zooming, one for zooming in and one for zooming out. When one of them is depressed, a click in the map will recenter on that point and zoom in or out. The SmartMap API makes it possible to zoom and pan within the already fetched data, but because the purpose of the application is to test the cache, I always fetch new data from the cache when the view coordinates are changed.

Pan

There is a toolbar button for panning. When it is depressed, a click in the map will move the map view in the direction of the click. The map is divided into nine invisible zones in a three by three grid. A click in the center zone has no effect. A click in the left middle or right middle zone will move the map view horizontally. A click in the upper middle or lower middle zone will move the map view vertically. A click in any of the corner zones will move the map diagonally. The step size is one half of the map view size.

Resize

The window can be resized. When this happens, the existing map data will be redrawn, but the application will not fetch more data from the cache to fill any new areas that may come into view. There is a refresh button that can be used to fetch new data and refresh the display after resizing.

Multiple layers

The first implementation of the MapBrowser application supported only one layer of either line, polygon or text data, and the display was only black and white. This was enough to test the performance, but it wasn't a very useful application. It was extended to support multiple layers. Color and style can be set individually for each layer. See figure 6.5. An information window (see figure 6.6) was added because the existing interface was unable to display timing and other info for more than one layer.

Results

The status bar in the application window displays the following information for the last map update:

- Lower left and upper right coordinates of the current view.

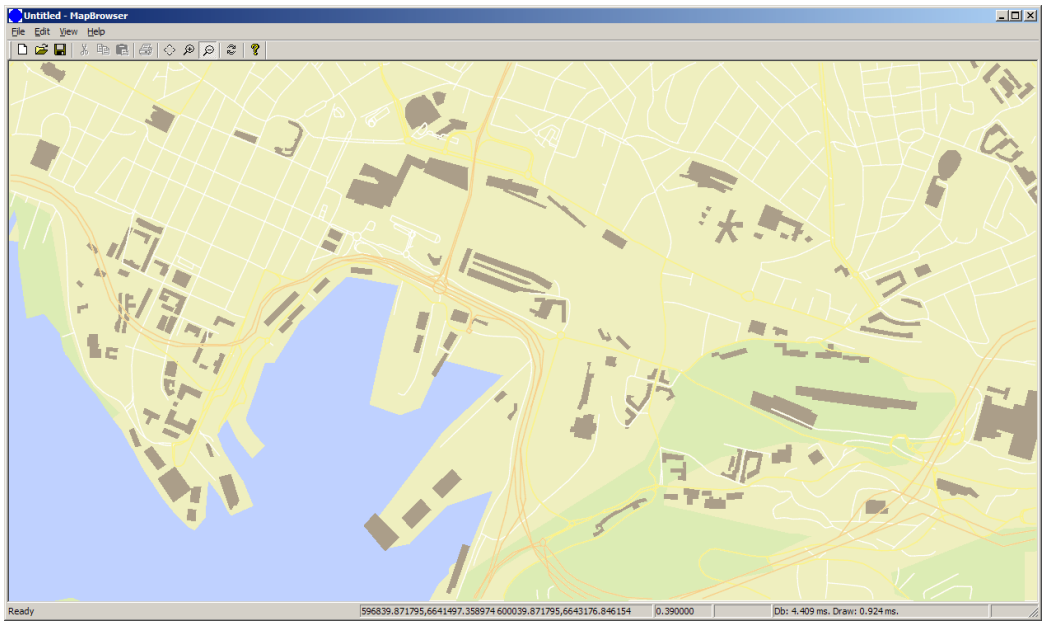

Figure 6.5: MapBrowser application window

Cache info

Name	x1	y1	x2	y2	DB time	Data size	Duplication	Redundancy	Use cache	% used
ta_lc	75	41	2	1	0.07	6856	0.00%	44.75%	yes	0.00%
ta_wa	75	41	2	1	0.03	3840	0.00%	15.19%	yes	0.00%
buildings_oslo	75	41	2	1	1.16	26512	0.00%	89.53%	yes	0.00%
roads78_m	75	41	2	1	2.66	79764	0.00%	87.48%	yes	0.00%
roads3456_m_2	75	41	2	1	0.42	18288	0.00%	76.88%	yes	0.00%
roads012_m_5	75	41	2	1	0.07	6196	0.00%	54.83%	yes	0.00%

Figure 6.6: MapBrowser information window

- Current scale.
- Total time spent fetching data from the caches in milliseconds.
- Time spent rendering the map in milliseconds.

The information window shows the following information per layer:

- The indexes of the lower left tile and the number of tiles, in each direction, covered by the query. These values are 0 if the cache is disabled.
- Time spent fetching data from the layer's cache in milliseconds.
- Data size in bytes.
- The amount of duplication across tiles relative to the size of the data after duplicate elimination. This value is 0 if clipping is enabled.
- The amount of data outside the request rectangle but inside the intersecting tiles relative to the total amount of data in the intersecting tiles.
- Whether the cache was used.
- How much of the memory reserved for the cache is in use.

6.8.2 Implementation details

MapBrowser was created with Microsoft Foundation Classes (MFC). It is based on the Document/View framework which is Microsoft's implementation of the Model-View-Controller pattern [5]. The Document object handles the initialization and use of the caches. The View object renders the data from the cache using the SmartMap library.

Due to the general-purpose nature of the SmartMap library, it has a rather complex interface. I created an object-oriented wrapper that hides the details and provides a much simpler rendering interface for use by the MapBrowser application.

Scale limitation

The MapBrowser application can display multiple layers. Each layer can have a lower and upper scale limit. When the map scale is outside this range, the layer will not be visible. This makes it possible to ignore very dense layers when the query rectangle is large, thus avoiding very long response times.

6.9 Performance tests

The performance tests are implemented as a single executable. It can run a number of different tests described in chapter 7. Which test to run must be specified at runtime. It reads test cases in the form of coordinates of request rectangles from a file and runs the specified test once for each test case. The results are written to *stdout*, and more detailed information is written to *stderr*. It is possible to specify an output file for the results and a log file for the details.

6.9.1 Parameters

The test program has the following configurable parameters:

- File with test cases
- Which test to run
- ODBC data source name.
- Database table name.
- Type of geometry.
- Enable/disable clipping.
- Output file name.
- Log file name.

In addition the different tests have the following parameters:

Clipping test

- Output directory for rendered maps

Cache size test

- Maximum cache size
- Minimum cache size
- Cache size step
- Tile size

Tile size test

- Minimum tile size
- Maximum tile size
- Tile size step
- Cache size

Chapter 7

Method

In this chapter, I will present the methods and test data used in the analysis.

7.1 Test data

7.1.1 Dataset

Most of the geographic data used in this thesis are from the MultiNet product sold by Tele Atlas [1]. MultiNet contains a wide range of vector data and covers 64 countries as of this writing. The data used here covers the southern part of Norway. It covers 12 of the 19 Norwegian counties and contains 8 of the 10 largest cities in Norway by population. Figure 7.1 illustrates the coverage of the MultiNet data. The data covers the dark gray area in the map. Note that it shows administrative boundaries, not physical boundaries.

A separate layer of buildings provided by the Norwegian Map Authority was used because buildings are not included in the MultiNet product. This layer covers only the city of Oslo.

All data were provided in UTM 33. They were converted to UTM 32 after import because most of the data are within this zone. The first row of table 7.1 contains the coordinates of the bounding box of all the layers.

The layers used in MapBrowser are listed in table 7.2. The road layers were all extracted from the same road layer in the Tele Atlas data. A road class attribute was used to separate the roads into three levels. In the original data, the road layers are divided into very small parts because they contain a lot of attributes. Every time the value of one of the attributes changes, a new line geometry must be started. This creates a lot of unnecessary overhead when using the data. A Shapefile manipulation program called ShapeTool developed by Faster Imaging

[1] http://www.teleatlas.com/

	x1	y1	x2	y2
Entire dataset, UTM 32	259000	6424000	706000	6954000
Oslo UTM 32	576833	6626705	614112	6658192

Table 7.1: Bounding boxes of the whole dataset and the area used for testing

Name	Type	Source
ta_lc	Land cover	Tele Atlas
ta_wa	Water	Tele Atlas
buildings_oslo	Large buildings	Norwegian Map Authority
roads78_m	Small roads	Tele Atlas
roads3456_m_2	Medium roads	Tele Atlas
roads012_m_5	Large roads	Tele Atlas

Table 7.2: Layers used in MapBrowser and the performance tests

was used to combine adjacent roads with the same road name into a common geometry.

For the performance tests, the largest line layer (roads78_m) and the largest polygon layer (lc) were used.

7.1.2 Test cases

The performance tests were run with a set of random test cases. A test case represents a request from a client and is defined by the coordinates of the lower left and upper right corner of the request rectangle.

It would have been better to have a set of test cases representing a real usage pattern. For a cache to be efficient, it depends on some kind of predictability in the requests. In a typical map system, some urban areas will be visited frequently, while other rural areas will never or very seldom be visited. With a random set of test cases, this predictability is absent. However, a realistic test of test cases was not available and is hard to create.

If I had used random test cases for the entire area covered by data, cache hits would have been very rare. Instead, I restricted the test area to a rectangular area of about 1174 square kilometers around the city of Oslo. The coordinates of the area are listed in the second row of table 7.1

I decided to use only square request rectangles, based on the assumption that when a user centers the map around a point of interest, he would be equally interested in context is all directions.

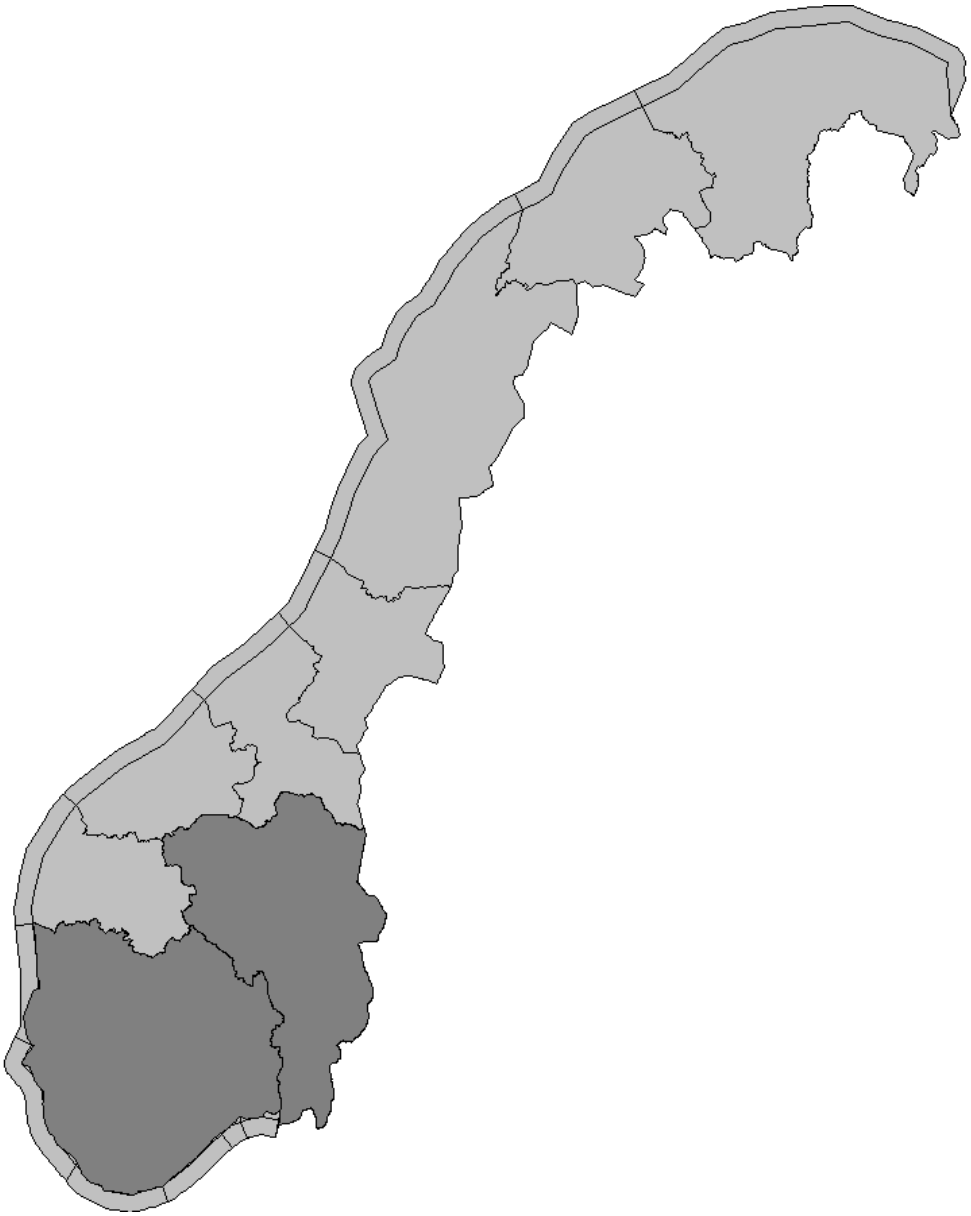

Figure 7.1: Coverage of the MultiNet data

The size of the test cases are between 300x300 meters and 3000x3000 meters. I chose these limtits by using MapBrowser to find a reasonable minimum and maximum size of a map at the innermost zoom level. See figure 7.3 and figure 7.4 for examples of the lower and upper size limit. Because of large zoom steps in MapBrowser, it was not possible to get maps of exactly 300x300 and 3000x3000 meters. The former is close to 300x300, but the latter is about 2300x2300.

Generating random test cases

The test cases were generated with a perl script. It generates a number of squares of random size between a specified minimum and maximum size at a random position within the specified bounding box. The squares will always be fully inside the bounding box. The coordinates of the lower left and upper right corners of the query rectangles are written to a file which will be read by the performance tests.

I also developed a small program that can create a shape file containing all query rectangles from a file as polylines. This makes it easy to get an impression of the amount of overlap between the requests and verify the randomness of the test cases. Figure 7.2 shows the 1000 test cases used in the performance test.

Ignoring requests with empty results

It can be argued that requests with empty results should be ignored in the performance tests because areas with no data will rarely be viewed by a real user of a real map application. However, because each layer has its own cache, an empty response for one layer doesn't mean that the same area will get an empty response for all layers. For this reason, I decided to not ignore requests with empty results. The clipping test is an exception. Because the rendering library will not render an empty dataset, I had to ignore requests with empty results in the clipping tests.

7.2 Timing

All time measurements in MapBrowser and the performance tests are in clock time, because the actual elapsed time is what a user of a map application sees. I used QueryPerformanceCounter and related functions, which is part of a high-resolution timing API built into Windows. Anti-virus software was disabled to prevent it from interfering with the test results. All tests were run at least twice and the results compared to verify that the tests were not disturbed by any temporary performance drops on any of the machines involved or on the network, and the confidence interval was sufficiently small.

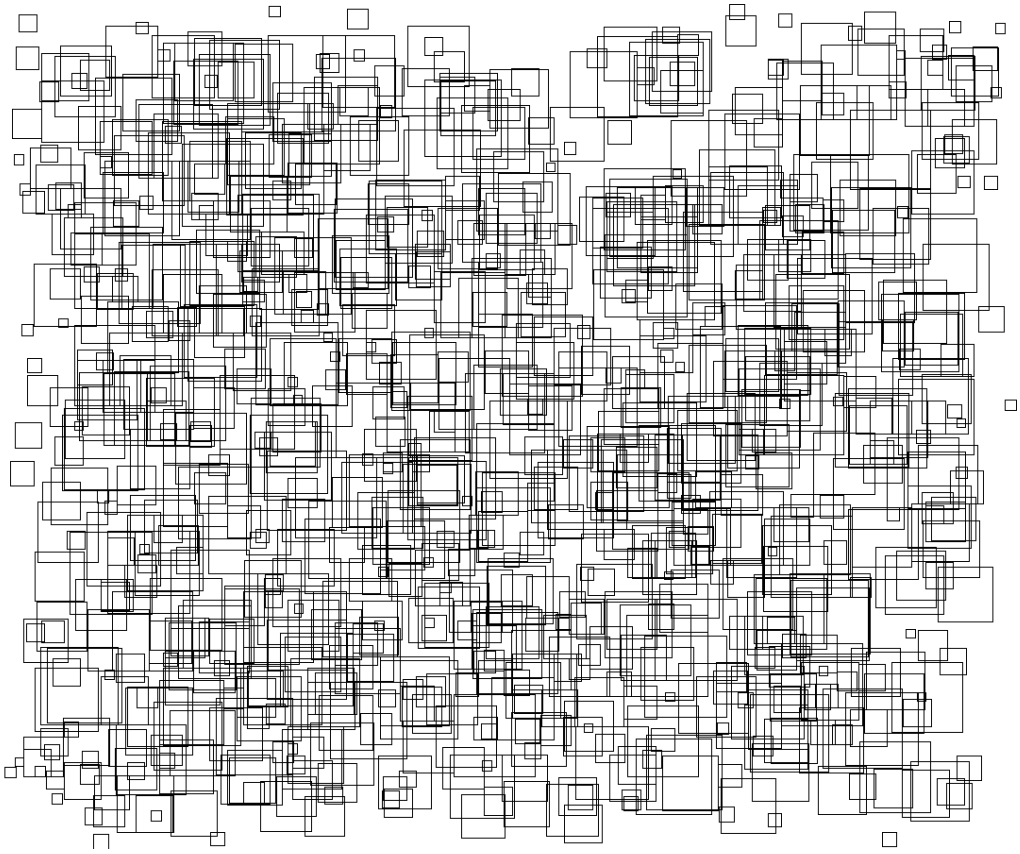

Figure 7.2: Random test data with 1000 cases

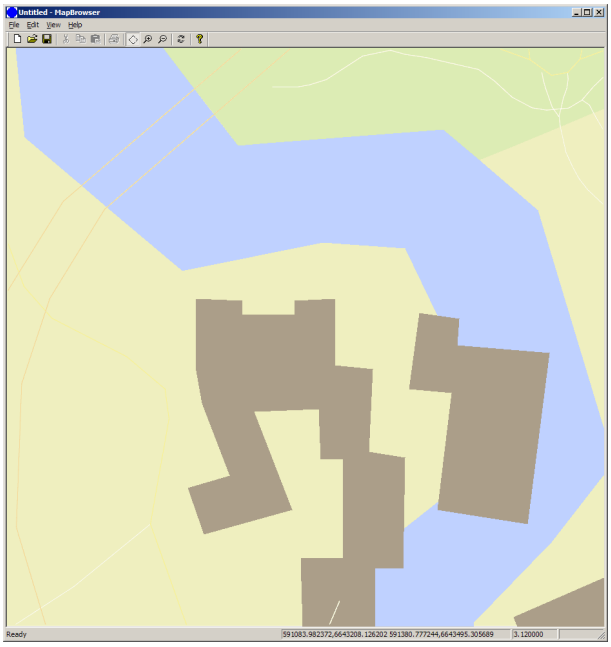

Figure 7.3: Example map of size 300 x 300

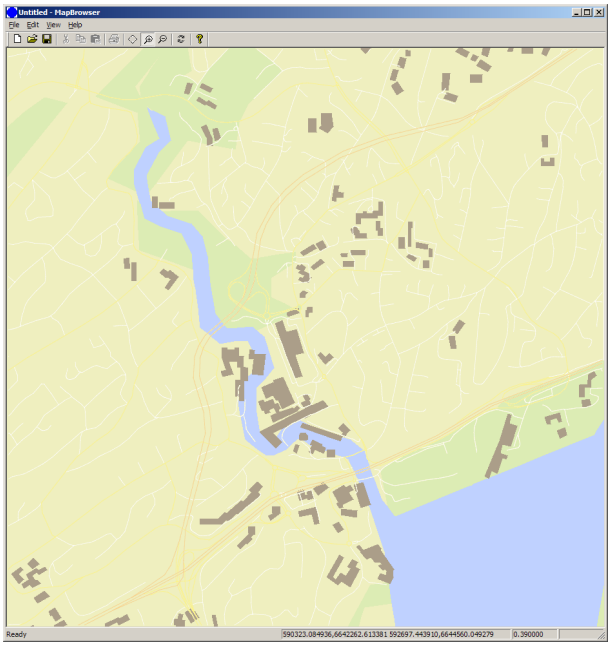

Figure 7.4: Example map of size 2300 x 2300

Name	Operating system	Processor	Memory
Machine A	Windows XP Professional	Intel Pentium M 1.8GHz	1.5 GB
Machine B	Linux (Fedora Core 4)	Intel P4/Xeon 2.80GHz	2 GB

Table 7.3: Specifications of test machines

The resolution of the timing API is approimately 0.28 microseconds (μs). The function call overhead is about 1.7 (μs). In MapBrowser and the performance tests, I record the time in milliseconds (ms) with two decimals which is well within the accuracy of the timing API.

All performance tests measure average response time over a number of iterations. The timer is "started" immediately before calling GEO_CACHE::GetArea() and "stopped" immediately after. The times are accumulated, and at the end, the total time is divided by the number of iterations to get the average response time. A manual check was performed to ensure that the confidence interval was small enough for the measurements to be valid for repeated measurements using the same data set.

7.3 Infrastructure

7.3.1 Hardware

I used two computers to test the cache. Initial testing was done on a laptop (machine A) with the clients and database on the same computer. When testing with a remote database, I used a server (machine B) for the database. The specifications of the two machines are listed in table 7.3.

7.3.2 Network

The bandwidth was about 5600 kilobits per second (kbps) from the server to the client and about 610 kbps from the client to the server.

7.3.3 Software

I used different software versions during development, but all tests were finally run on PostgreSQL 8.1.5 with PostGIS 1.1.3. PostGIS was built with the GEOS library [2] which provides geometric operations and the PROJ4 library [3] which pro-

[2] http://geos.refractions.net/

[3] http://proj.maptools.org/

vides coordinate transformations.

7.3.4 Traffic analysis

In the early phases of development, the database traffic was monitored to detect possible bottlenecks. NetLimiter 2 Lite [4] displays bandwidth usage per application in realtime. This was used to monitor the bandwidth used by MapBrowser.

After a problem was detected, Ethereal 0.10.6 [5] was used for examining the data packets.

MapBrowser displays response size and response time for each request. This can be used to calculate the effective transfer rate.

7.4 Test plan

The following tests are executed with the client on Machine A and the server on Machine B.

7.4.1 A: Clipping on/off

The first test uses fixed cache size and tile size and runs all test cases with and without clipping in the database. Both average cache response time and average rendering time is recorded. This is interesting as a preliminary analysis. The results from this test will decide if clipping will be used in the rest of the tests.

Clipping tests ignore test cases with empty response because SmartMap skips rendering if there is no data. The rendered maps are saved to file for visual inspection. Maps rendered with and without database clipping should be visually identical. They are not byte for byte identical, so automatic comparison does not work.

7.4.2 B: Vary cache size

This test uses a fixed tile size. It runs all test cases with different cache sizes and records the average response time for each cache size. The purpose of this test is to study the effect of varying the cache size.

[4] http://www.netlimiter.com/
[5] http://www.ethereal.com/

7.4.3 C: Vary tile size

This test uses a fixed cache size which may be different for each layer. It runs all test cases with different tile sizes and records the average response time for each tile size. The purpose of this test is to find an optimal tile size for each layer.

For each layer, the cache size will be set to the lowest cache size from test C which gives better performance than without a cache.

7.4.4 D: Cache size revisited

A new tile size is selected for each layer based on the results of test C. Test B is repeated with these tile sizes to see if performance can be improved by careful selection of tile size.

Chapter 8

Analysis

In this chapter, I will present the test reults.

8.1 A: Clipping

The clipping in the SmartMap library is very fast. I wanted to see if it was better to do a query without clipping and let the SmartMap library do the clipping than to let the database do the clipping. If clipping is not done by the database, the transmitted data are obviously larger and there may be duplicates when the request covers multiple tiles. This means that duplicates must be eliminated by the cache when merging the tiles.

The tile size used in this test was 1000x1000 meters, and the cache size was unlimited.

The results are presented in figure 8.1 and figure 8.2. Figure 8.1 shows the response time from the database, and figure 8.2 shows the rendering time in the client.

For each pair of columns, the first column is with clipping in the rendering library and the second is with clipping in the database. We see that in all cases, clipping in the database gives the best results. We also see that data size is very significant, because in the first part of figure 8.1, the database is doing less processing in the case without database clipping, but the response time for this case is much higher than in the case with database clipping since more data must be sent to the client.

Because this test shows that clipping is the database is clearly preferable, this was done in the remaining tests.

8.2 B: Varying cache size

I expected to see that the performance increased with increasing cache sizes. At some point the cache will be large enough to contain everything, and the performance will stop increasing.

For this test, a tile size of 1000x1000 meters was used.

The results are presented in figure 8.3 and figure 8.4. We see that larger cache gives better performance up to a point where the cache contains all the data. For small cache sizes, the performance is worse with the cache than without.

A too small cache slows down the system because of a lot of cache replacements and few cache hits. Then it is better without a cache.

8.3 C: Varying tile size

I expected to see that the performance is low with small tiles because of much overhead, and that large tiles are slow because each query will return a lot of data. Somewhere between the smallest and largest tile size, I expected to find an optimal tile size.

For the lc layer, a cache size of 0.6 megabytes was used, and for the road78_m layer, the cache size was 4.8 megabytes.

If tiles are not too small, we expect that using a cache is better than not using a cache. In addition, we expect that the tile size will be significant. A large number of queries caused by smaller tile sizes will slow down the cache.

We see from figure 8.5 and figure 8.6 that for the line layer, one particular tile size is best but the polygon layer, all tile sizes above a certain limit give a good result. The geometry type is not the only difference between the tables. The polygons are very large compared to the lines. Without testing more tables, it is hard to tell the reason for the different behavior.

8.4 D: Revisiting cache size

Using an optimal tile size found in test C, I expected to see a higher performance than in test B.

For the lc layer, a tile size of 2000x2000 meters was used, and for the road78_m layer, the tile size was 1500x1500 meters.

The results are presented in figure 8.7 and figure 8.8. We see that for both layers, the performace increases with a careful choice of tile size. We also see that the cache can now be smaller and still perform better than not using a cache.

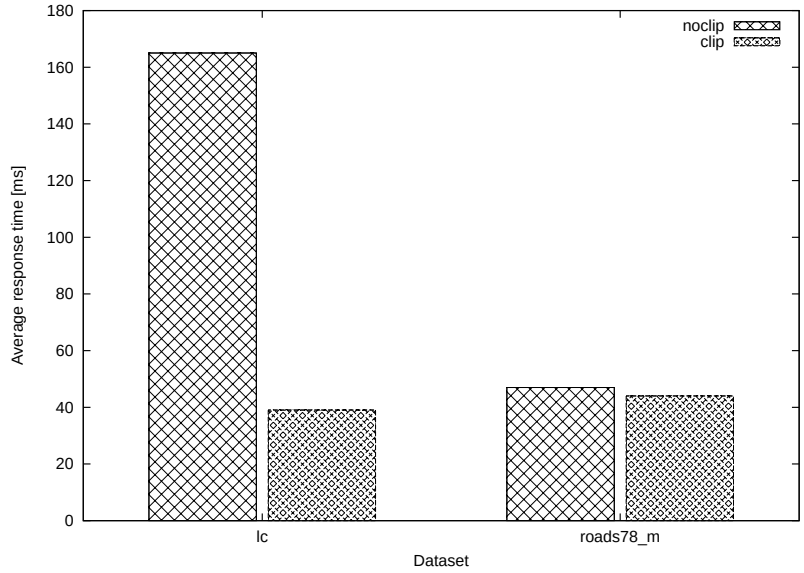

Figure 8.1: Database time with and without clipping

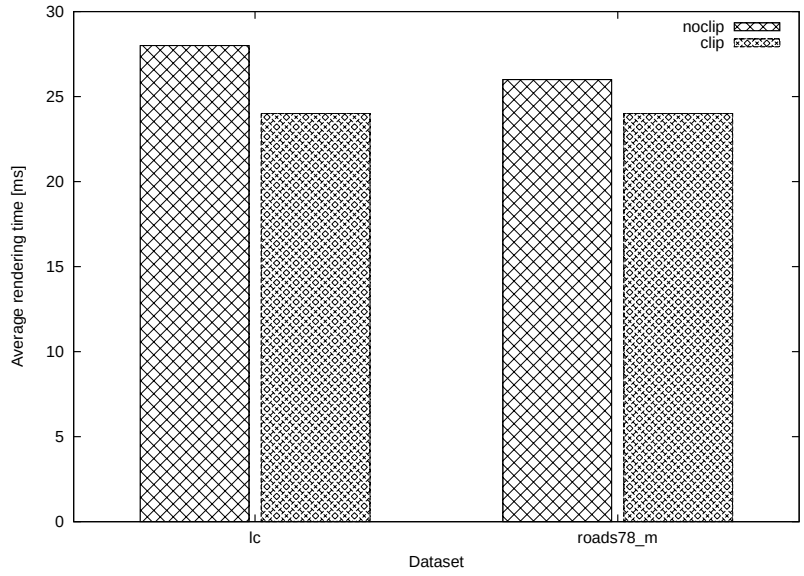

Figure 8.2: Rendering time with and without clipping

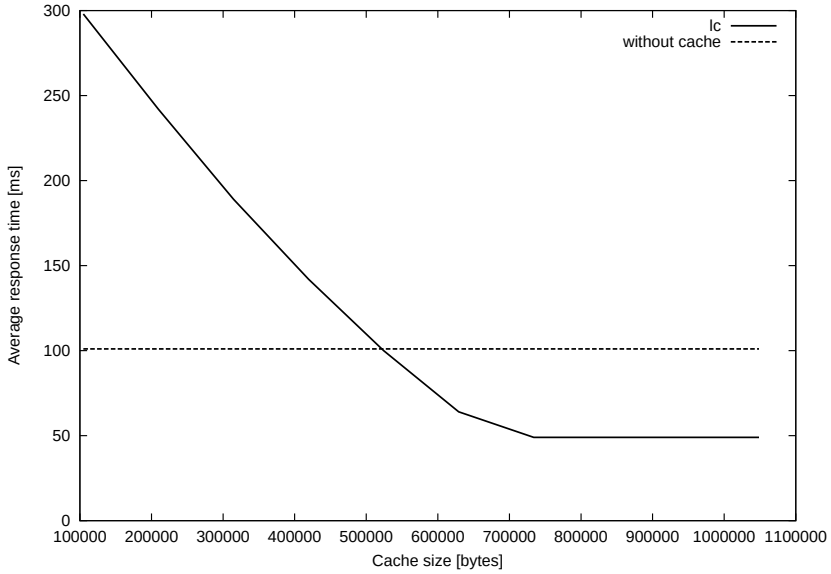

Figure 8.3: Cache size with dataset lc

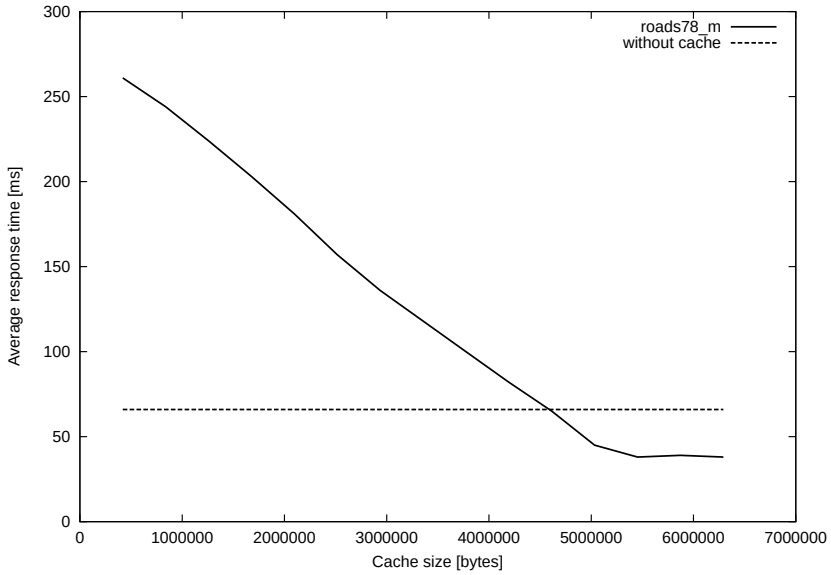

Figure 8.4: Cache size with dataset roads78_m

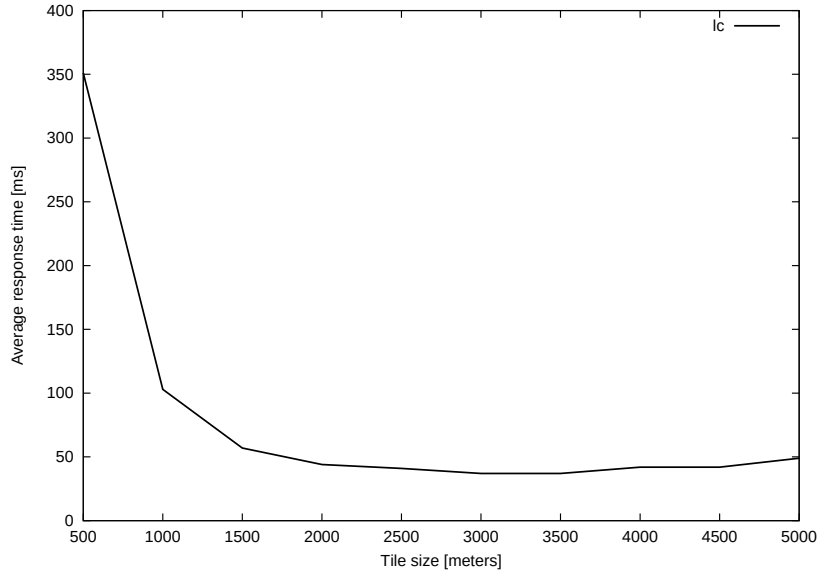

Figure 8.5: Tile size with dataset lc

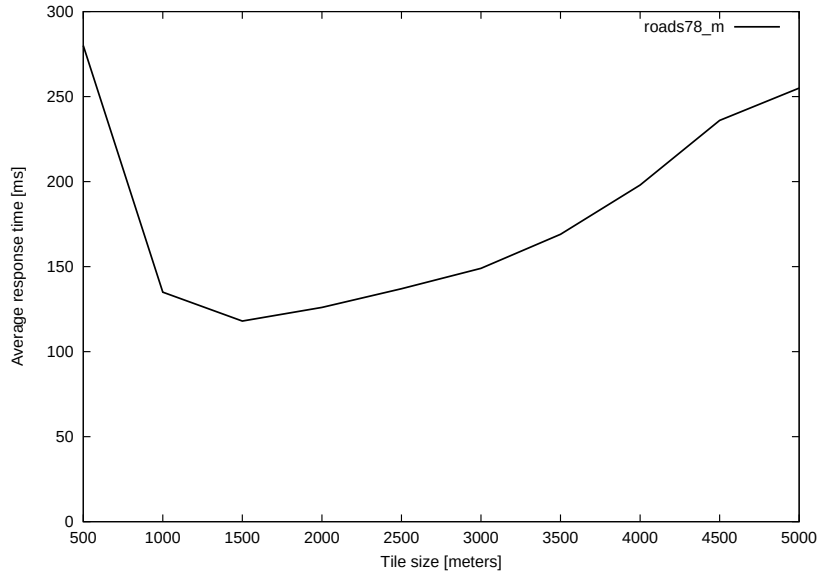

Figure 8.6: Tile size with dataset roads78_m

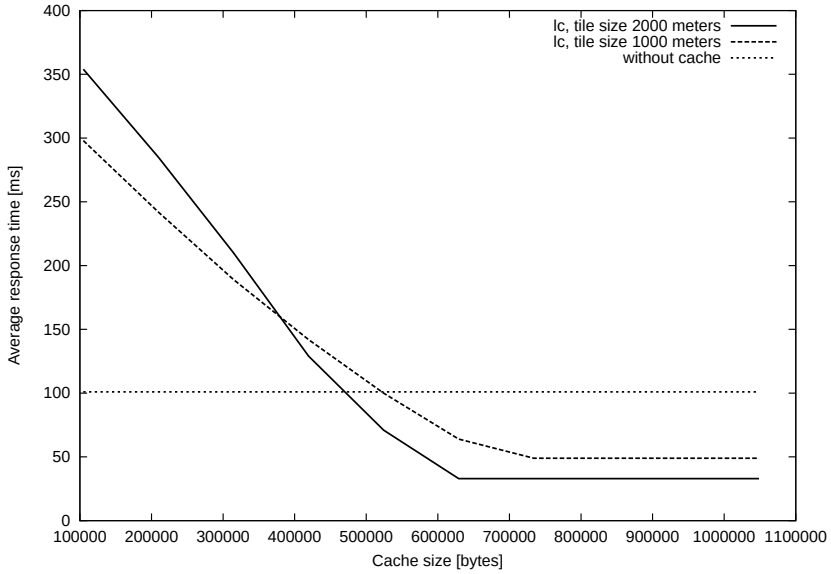

Figure 8.7: Cache size with dataset lc and adjusted tile size

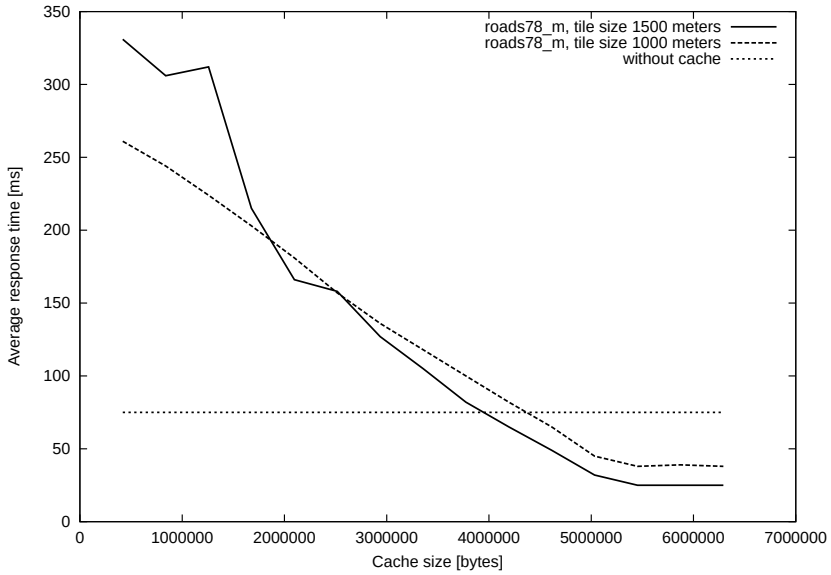

Figure 8.8: Cache size with dataset roads78_m and adjusted tile size

Expression	Encoding	Size
the_geom	Hexadecimal	200%
AsBinary(the_geom)	Octal	400%
encode(AsBinary(the_geom), 'base64')	Base64	133%
encode(AsBinary(the_geom), 'hex')	Hexadecimal	200%

Table 8.1: Database response encodings

8.5 Traffic analysis

In an early phase of the development, the effective transfer rate calculated from data size and response time displayed in MapBrowser's information window was surprisingly low compared to the actual bandwidth usage reported by NetLimiter. I found that the geometries were encoded as octal ASCII numbers in a way that increased the size by 300%.

Table 8.1 lists the possible encodings of binary data in the response from a PostgreSQL database and how they affect the data size.

When I enabled Base64 encoding in the query and implemented a Base64 decoder in the cache, the size of the transmitted data was reduced, and the response time reported by MapBrowser improved by more than 50%.

I have later discovered that the encoding is done by the ODBC interface. By using PostgreSQL libraries directly, it is possible to get binary data. This means that if the network is the bottleneck, further improvement can be achieved by rewriting the database interface layer to use the PostgreSQL libraries.

Chapter 9

Discussion

In this chapter, I will discuss the results from the analysis.

9.1 Cache replacement

LRU is inefficient as a selection criterion when the tests cases are random because it is based on the assumption that recently requested tiles are more likely to be requested again in the near future. A replacement algorithm considering the cost of cache misses would probably work better with random test cases because the cost is independent of the usage pattern. All cache lines in the cache already contain the time spent fetching them, so extending the replacement algorithm to include this factor is possible. But this doesn't change the fact that random test cases cannot tell the whole truth about the efficiency of a cache.

9.2 Cache consistency

The map data used in this thesis are very static. The MultiNet datasets are updated a few times a year. The lifetime of the cache is short because it is an in-memory cache which is destroyed when the application terminates. Because if this, it is highly unlikely that map updates on the server side will cause problems in the client. Therefore, cache consistency has been ignored in this cache implementation.

9.3 Data compression

As explained in chapter 6, a reduction in data size on the network was achieved by encoding the data with the Base64 encoding. It was also mentioned that the

size could have been reduced further by using the PostgreSQL libraries directly, making it possible to transmit raw binary data.

Using a lossless compression algorithm such as Lempel-Ziv-Welch (LZW) [22] may further reduce the size, but it is uncertain if the effect will be significant because a dictionary-based algorithm like LZW depends on repetition in the data.

Another issue with compression is the added processing required to decompress the data in the client. The question is whether the increase in power consumption because of decompression is higher than the reduction in power consumption because of reduced network traffic.

One way to reduce the network traffic without a compression algorithm is to use single-precision instead of double-precision floating point numbers for the coordinates. This would reduce the size of coordinates by 50%. The precision would still be much higher than necessary.

9.4 Reducing network traffic

The bandwidth is a bottleneck for the mobile map application, and the communication between the cache and the database could be optimized. This would require a two part system as illustrated in figure 9.1. There are several possible models:

1. The server side component delivers compressed vector data.

2. Put the cache at the server side. The cache would always deliver one prepared "package" of vectors. No duplicates will be sent over the network. But I would get the same if I did a normal query. Not sure if this would be useful in any way. Not useful by itself. This is a very different way to use a cache.

3. Split the cache into two parts, one client side and one server side. Caching will be done at both sides.

The internals of the cache subsystem can be changed without affecting the interfaces to the database and the client.

9.5 Alternative architectures

9.5.1 Server side caching

With a well chosen tile size, the bandwidth is the bottleneck. This means that a server side cache would not improve performance as long as the database can fill the bandwith. However, if the number of clients is large, the server's processing

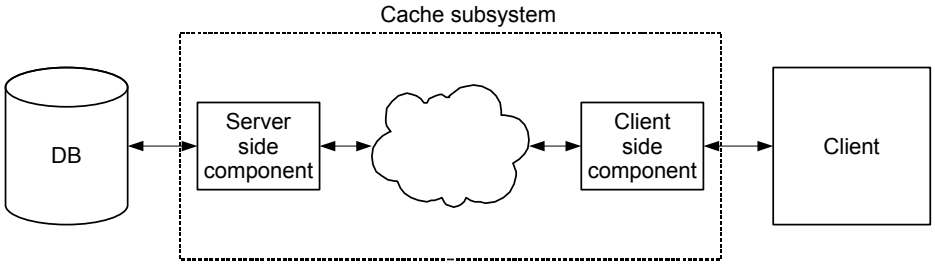

Figure 9.1: Alternative two-part cache design

power may become a limiting factor because spatial searches are computationally intensive. In this case, a server side cache could improve performance.

9.5.2 Subnet caching

Compared to a server side cache, a subnet cache may have a significantly faster connection to the clients than a server side cache. At the same time, this kind of cache can be much larger than a client side cache because it can use powerful hardware. This makes a cache very attractive in this kind of architecture.

9.6 Problems

9.6.1 Object id conflicts

Initially, I used an object id of 0 to mark the end of a sequence of geometries. However, it turned out that the PostGIS data loader creates geometries with object ids starting at 0. I used the highest possible number instead *(0xffffffff)*. This will work as long as this number is not used as an object id. This will most likely never happen.

9.6.2 Clipping

Clipping introduced a new problem. Sometimes when clipping a polygon, the return type is GeometryCollection because the result contains both lines and polygons. This was discovered with MapBrowser. The solution was to handle the response as a special case if the type was GeometryCollection and extract only the polygons.

9.7 Possible improvements

9.7.1 Pre-fetching

Pre-fetching was mentioned in chapter 4 as a way to increase the probability of cache hits if it is possible to predict which tiles will be used in the near future. A disadvantage with prefetching in a mobile setting is that for some mobile/wireless communication services the price depends on the amount of transferred data. In this case, the user will be charged for some data he never uses.

An attractive compromise is to implement on-demand prefetching. When the user knows that his mobile terminal is connected to a network with free traffic, he can request pre-fetching. Then the cache will fill itself with data that the user is likely to need in the future according to some pre-fetching algorithm.

Subnet caches open up another possibility of prefetching. If multiple subnet caches are connected to each other and are able to identify the connected users, they can learn from each other what the users are likely to need and start prefetching as soon as the users enter the subnet [9].

9.7.2 Parallelizing queries

If the bandwidth hadn't been the limitation, parallelizing the requests for the tiles covered by a request could have improved performance because a database can be better utilized when it processes multiple requests in parallel. Parallelizing may even an advantage when the bandwidth is limited, because without it, the network will not be used while a query is being processed on the server.

9.7.3 Alternatives to the 2D tile index

As explained in chapter 6, a two-dimensional array is used as a tile index. This makes it extremely quick to look up a tile. However, this tile index uses four bytes per possible tile, and with a large area it can become quite large. The size of the tile index used in this thesis is about 413 kilobytes. This is not much for a modern laptop computer, but for a PDA or SmartPhone, this amount of memory may be

too large. For these platforms, another data structure must be used which requires less memory at the expense of slightly slower lookups.

Chapter 10

Conclusion and future work

In this chapter, I will draw conclusions based on the work in the previous chapters. I will also point to some areas where further work is necessary.

10.1 Conclusions

10.1.1 Test data

Testing a cache with random tests has limited value. Real test cases representing a typical usage pattern are necessary to get an exact figure of the expected performance improvement of a cache. There will always be clustering of popular hits, and this will have a positive effect by introducing the cache.

10.1.2 Encoding

The experiments have revealed that a database often uses far from optimal encoding of data results. Using a standard ODBC driver, 300% overhead was measured. Thus, binary access directly to the database is a preferred method. If the data can be represented by single precision instead of double precision floating point numbers, up to 50% can be saved.

10.1.3 Clipping

Clipping features on the server side is clearly important for tables containing large geometries when the bandwidth is limited.

10.1.4 Cache size

The performance of the cache depends on the cache's maximum data size. Larger cache size results in better performance up to the point where the cache is large enough to contan all data. This is exactly as expected.

10.1.5 Tile size

Tile sizes also have great impact on the performance of a query. The experiments revealed that for some tables, the choice of tile size is particularly important because the performance decreases significantly both above and below the optimal tile size. For other tables, a range of tile sizes around the optimal tile size gives almost as good performance. This probably depends on the size of the features in the table.

The significance of tile sizes is important to keep in mind when designing map systems with multiple zoom levels with different tile sizes.

The tile size can also affect power consumption. According to Forman [4], sending consumes much more power than receiving. This means that even if the total amount of transmitted data is the same, minimizing the number of queries is preferable.

10.1.6 Alternative architectures

A centralized cache will not have an effect unless the number of clients is large, because with a limited number of clients, the bandwidth of the clients will be the limiting factor.

A subnet cache seems to be a more attractive alternative because it may provide faster connections to the clients and because it brings powerful hardware closer to the clients. It has also the benefit over a centralized cache that the usage pattern of the users on a subnet may be more predictable than the one for a large number of users using a centralized server because users of a subnet are located close to each other and may be particularly interested in maps of their surroundings.

10.2 Further work

I have implemented only one of three alternative cache architectures. I have tried to make some conclusions about the alternatives based on the experiments with this implementation, but facts can only be established by implementing these alternatives. Both require a more advanced cache design because they must support

multiple simultaneous users.

The cache replacement algorithm used in this implementation is based only on LRU. Many other cache replacment algorithms are described in the literature. Some of them should be tried and compared with the LRU algorithm to see how much improvement a more advanced algorithm can make.

Appendix A

Data format

This appendix contains a description of the data format used in the cache implementation. The cache converts data from the spatial database into this format for internal storage. The output from the cache is also in this format.

Each tile is represented by an area record. An area record consists of a number of feature records. The format of area records and feature records is illustrated in figure A.1. The fields of the feature records are described in table A.1. Table A.2 shows the format of the record header. Table A.3 lists the supported geometry types, and table A.4 lists the supported attribute types. The format of the geometry records is described in table A.5 The orientation of rings is not significant. A point is a pair of x,y coordinates. The coordinates are double-precision floating point numbers. The byte order is platform dependent.

Field	**Description**	**Size in bytes**	**Comment**
AOF	Attribute offset	4	
ATR	Attribute	Type dependent	
EOB	End-of-buffer	4	
HDR	Header	4	
LBL	Label	Number of characters	No terminator
LOF	Label offset	4	Rel. to start of feature record
LSZ	Label size	1	
NOR	Number of records	4	

Table A.1: Record fields

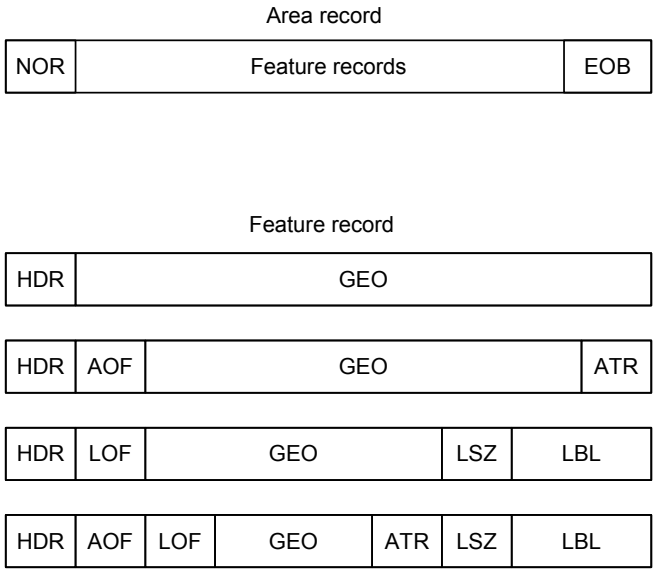

Figure A.1: Record format

Bits	**Field**	**Contents**
0-2	GTY	Geometry type
3	BLB	Has label, 0 - no, 1 - yes
4	BAT	Has non-spatial attribute, 0 - no, 1 - yes
5-7	ATY	Type of non-spatial attribute
8-31	RSZ	Record size in bytes incl. header

Table A.2: Header format

1	POINT
2	LINE
3	POLYGON
4	MULTIPOINT
5	MULTILINE
6	MULTIPOLYGON

Table A.3: Geometry types

Value in ATY	**Attribute type**
0	8-bit signed integer
1	8-bit unsigned integer
2	16-bit signed integer
3	16-bit unsigned integer
4	32-bit signed integer
5	32-bit unsigned integer
6	Float (32 bits)
7	Double (64 bits)

Table A.4: Non-spatial attribute types

Point

Offset	**Contents**	**Size**
0	point	16

Line

Offset	**Contents**	**Size**
0	numpoints	4
4	points	$numpoints * 16$

Polygon

Offset	**Contents**	**Size**
0	numrings	4
4	numpoints	$numrings * 4$
Variable	points	$(\sum_{i=0}^{numrings-1} numpoints[i]) * 16$

MultiPoint

Offset	**Contents**	**Size**
0	numpoints	4
4	points	$numpoints * 16$

MultiLine

Offset	**Contents**	**Size**
0	numparts	4
4	numpoints	$numparts * 4$
Variable	points	$(\sum_{i=0}^{numparts-1} numpoints[i]) * 16$

MultiPolygon

Offset	**Contents**	**Size**
0	coordinate offset	4
4	numparts	4
8	numrings	$numparts * 4$
Variable	numpoints	$(\sum_{i=0}^{numparts-1} numrings[i]) * 4$
Variable	points	$(\sum_{i=0}^{numparts-1} (\sum_{k=0}^{numrings[i]} numpoints[k])) * 16$

Table A.5: Geometry format

Bibliography

[1] Thomas Brinkhoff. The impacts of map-oriented internet applications on internet clients, map servers and spatial database systems. In *Proceedings of the 9th International Symposium on Spatial Data Handling*, Beijing, China, 2000.

[2] Thomas Brinkhoff. A robust and self-tuning page-replacement strategy for spatial database systems. In *Extending Database Technology*, pages 533–552, 2002.

[3] Environmental Systems Research Institute, Inc. *ESRI Shapefile Technical Description*, July 1998.

[4] G. H. Forman and J. Zahorjan. The challenges of mobile computing. Technical Report TR-93-11-03, 1993.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[6] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson Education International, 2002.

[7] Michael F. Goodchild, David J. Maguire, and David W. Rhind. *Geographical Information Systems: Principles and applications*. Harlow, UK: Longman, 1991.

[8] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[9] Tomasz Imielinski and B. R. Badrinath. Data management for mobile computing. *SIGMOD Record*, 22(1):34–39, 1993.

[10] International Organization for Standardization. *ISO 19107. ISO/TC 211 Geographical information - spatial schema*, May 2003.

[11] Simon Josefsson. The base16, base32, and base64 data encodings. RFC 4648 (Proposed Standard), October 2006.

[12] Yong-Kyoon Kang, Ki-Chang Kim, and Yoo-Sung Kim. Probability-based tile pre-fetching and cache replacement algorithms for web geographical information systems. In *ADBIS '01: Proceedings of the 5th East European Conference on Advances in Databases and Information Systems*, pages 127–140, London, UK, 2001. Springer-Verlag.

[13] Marcel Kornacker. High-performance extensible indexing. In *The VLDB Journal*, pages 699–708, 1999.

[14] Paul A. Longley, Michael F. Goodchild, David J. Maguire, and David W. Rhind. *Geographic Information Systems and Science*. John Wiley & Sons, August 2001.

[15] David Loshin. *Efficient Memory Programming*. McGraw-Hill, 1998.

[16] Open Geospatial Consortium, Inc. *OpenGIS Geography Markup Language (GML) Implementation Specification*, February 2004.

[17] Open Geospatial Consortium, Inc. *OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture*, October 2006.

[18] Open Geospatial Consortium, Inc. *OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option*, October 2006.

[19] Open GIS Consortium, Inc. *Web Map Service Implementation Specification, Version 1.1.1*, January 2002.

[20] Oracle Corporation. *Oracle Spatial User's Guide and Reference, 10g Release 1*, December 2003.

[21] J. A. Orenstein. Redundancy in spatial databases. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 295–305, New York, NY, USA, 1989. ACM Press.

[22] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1998.

[23] Junho Shim, Peter Scheuermann, and Radek Vingralek. Proxy cache design: Algorithms, implementation, and performance. *Knowledge and Data Engineering*, 11(4):549–562, 1999.