This article appeared in: Lee, Jiyeong / Zlatanova, Sisi (Eds.): Proceedings of the 3rd International Workshop on 3D Geo-Information, Seoul, Korea. Lecture Notes in Geoinformation & Cartography, Springer Verlag, 2009

Making interoperability persistent: A 3D geo database based on CityGML

Alexandra Stadler, Claus Nagel, Gerhard König, Thomas H. Kolbe

Technische Universität Berlin, Institute for Geodesy and Geoinformation Science,
Straße des 17. Juni 135, 10623 Berlin, Germany
{stadler | nagel | kolbe}@igg.tu-berlin.de, gerhard.koenig@tu-berlin.de

Abstract. Virtual 3D city models are becoming increasingly complex with respect to their spatial and thematic structures. CityGML is an OGC standard to represent and exchange city models in an interoperable way. As CityGML datasets may become very large and may contain deeply structured objects, the efficient storage and input/output of CityGML data requires both carefully optimized database schemas and data access tools. In this paper a 3D geo database for CityGML is presented. It is shown how the CityGML application schema is mapped to a relational schema in an optimized way. Then, a concept for the parallelized handling of (City)GML files using multithreading and the implementation of an import and export tool is explained in detail. Finally, the results from a first performance evaluation are given.

Keywords: CityGML, 3D city model, 3D spatial database, database import and export, JAXB, GML processing

1 Introduction

Like many cities in Germany, Berlin is currently establishing a virtual 3D city model. More and more applications require additional height information and object structuring in vertical direction – just think of urban and landscape modelling, architectural design, tourism, 3D cadastre, environmental simulations, radio network planning, disaster management, or navigation. In order to assess the comprehensive 3D geoinformation for a city like Berlin, an appropriate data management component has to be built. Here, data may be collected, compared, adapted, updated, and exchanged. The data is used for urban studies, planning variants, calculation of intervisibility, impacts of vegetation alterations, and semantic data explorations. A necessary precondition is the existence of a standardised data model, ensuring consistent and interoperable data structuring.

CityGML [9] is an international standard for the representation and exchange of 3D city and landscape models issued by the Open Geospatial Consortium (OGC). The common information model behind CityGML defines classes and relations for the most relevant topographic objects in cities and regional models with respect to

their geometrical, topological, semantic and appearance properties. By covering thematic information and structures, CityGML complements 3D graphics formats like KML and X3D/VRML. CityGML is implemented as an application schema for the Geography Markup Language (GML) 3.1.1 [3] of the OGC and the ISO TC211.

Based on CityGML, a 3D geo database for the official Berlin 3D city model was established. The main development objective was to achieve both the efficient storage and fast processing of CityGML. For this reason, the CityGML data model was mapped to a compact relational database schema. Moreover, an import/export tool was realised to facilitate the high-performance processing of CityGML and GML structures. Both aspects are considered integral parts of the 3D geo data management in Berlin. In this paper, we therefore address the modelling and database design of the Berlin 3D geo database as well as the software engineering aspects of the import/export tool.

2 A 3D geo database for Berlin

The 3D geo database has been realised as an Oracle 10G R2 Spatial relational database schema. In the first project phase, the Institute of Geodesy and Geoinformation, University of Bonn, built a first version that was confined to a subset of CityGML [1]. In the second phase, we now have redesigned the existing database schema to fully comply with CityGML 1.0.0. For this upgrade, additional support of interior building structures, the new appearance model and the full set of CityGML's thematic modules is provided.

In detail, the database implements the following key features of CityGML:

- **Complex thematic modelling**
  The description of thematic features includes attributes, relations, and nested aggregation hierarchies (part-whole-relations) between features. Since on the spatial level geometry objects are assigned to features, both a semantic and a geometrical aggregation hierarchy can be employed. The rich semantic information can be used for thematic queries, analyses, or simulations.
- **Five different Levels of Detail (LODs)**
  Following the idea of multi-representation, every geo object (including DTMs and aerial photographs) can be stored in five different LODs. With increasing LOD, objects not only obtain a more precise and finer geometry, but do also gain in thematic refinement.
- **Appearance data**
  In addition to semantic information and geometry, features can have appearances, i.e., information about the observable properties of a feature's surface. Appearances can represent textures and materials, but are not restricted to visual properties. In fact, appearances can transport any surface based theme, such as infrared radiation, noise pollution, etc.
- **Complex digital terrain models (DTMs)**
  DTMs may be represented in four different ways in the 3D geo database: by

regular grids, triangulated irregular networks (TINs), 3D mass points and 3D break lines. For each LOD a complex relief can be aggregated from any number of DTM components of different types. For example, 3D mass points and break lines can be used together to form complex terrain models.

- **Representation of generic and prototypical 3D objects**
  Prototypical objects are used for memory-efficient management of objects that frequently occur in the city model at different locations, e.g., pieces of street furniture like lanterns, road signs or benches. Each instance of a prototypical object may refer to a particular prototype object for each LOD.
- **Free, also recursive aggregation of geo objects**
  Geo objects can be aggregated to groups according to user-defined criteria, e.g., to model a building complex consisting of individual building objects. The groups themselves represent geo objects which can be assigned names and additional classifying attributes. Groups again may contain other groups as members, resulting in aggregation hierarchies of arbitrary depth.
- **Flexible 3D geometries**
  The geometry of 3D objects can be represented through the combination of surfaces and solids as well as any, also recursive, aggregation of these elements.

The previous version of the database added two aspects to the underlying information model which exceed the capabilities of CityGML [1]. These aspects have been retained in the upgraded database:

- **Management of large aerial photographs**
  The database can handle aerial photographs of arbitrary size. Using Oracle 10G R2 Spatial GeoRaster functionality, tiled, homogeneous photographs can be aggregated to one single image.
- **Version and history management**
  The version and history management employs Oracle's Workspace Manager. It is largely transparent to applications that work with the database. For administration of planning areas and embodied planning alternatives, the tool "PlanningManager" was implemented and added to the 3D geo database. Furthermore, procedures saved within the database (Stored Procedures) are provided, which allow for comfortable management of planning alternatives or versions [10].

The following sections explain the different steps of the database development. Important design decisions are pointed out. The three main steps are marked as (a), (b), and (c) in fig. 1:

(a) **Simplification of CityGML's data model** (section 3)
  In order to achieve a more compact database schema and improve query performance, the complex data model was simplified at some critical points.

(b) **Derivation of the relational database schema** (section 4)
  The simplified object-oriented data model has been mapped to relational tables. The number of tables was optimized in order to minimize the number of joins for typical queries.

(c) **Creation of an import and export tool** (section 5)
The database administration tool allows processing of very large CityGML instance documents (>> 4 GB). Multiprocessor systems or multi-core CPUs are leveraged through a multithreaded architecture.

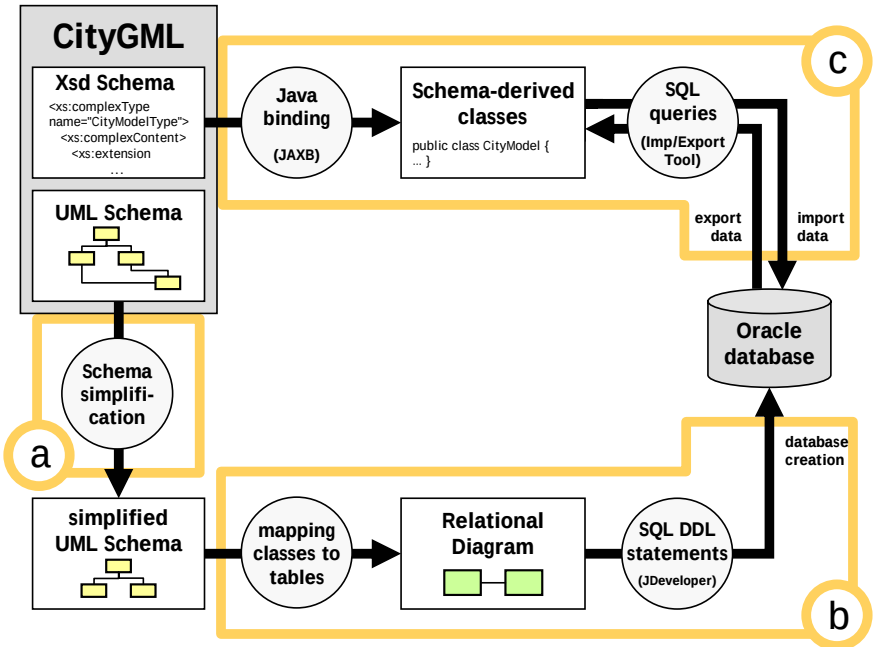

Fig. 1. Tasks in the development of Berlin's city geo database:
(a) Simplification of CityGML's data model,
(b) Derivation of the relational database schema,
(c) Creation of an import and export tool for CityGML instance documents.

3 Simplification of CityGML's data model

In order to cope with arbitrary CityGML datasets, the 3D geo database has to cover all modelling aspects of CityGML 1.0.0 [9]. However, CityGML uses concepts which are seen as quite complex by some users and software implementors. Complexity is found on different levels:

- CityGML comprises **different thematic models** to represent the topographic objects in cities and regional models. These models are different in structure, e.g., regarding relations and aggregation hierarchies between features, and spatial properties of features. Consequently, there is no simple 'mechanic' way to create a standardized relational model adhering to each thematic component.

- On the **level of single thematic model components** we have to face different relations between objects, allowing for modelling in various levels of complexity. First, the thematic model of CityGML is based on hierarchical decomposition of geo objects. E.g., a building may be decomposed into building parts, rooms, walls, windows, and doors, etc. Second, some cyclic relations exist, resulting in nested object structures of arbitrary depth. E.g., building parts may be recursively decomposed into nested building parts. Another prominent example is the "generalizesTo" relation for CityObjects which maintains links between corresponding objects in different LOD. Third, multifaceted aggregations are included between thematic classes and their spatial properties. For one object, several geometric representations may be provided simultaneously.
- On the **geometry level**, hierarchies comparable to those on the thematic side can be found. CityGML supports a subset of GML's geometry model comprising mainly polygonal geometries: Surfaces, CompositeSurfaces, MultiSurfaces, Solids etc., all of which may be assigned appearance information. This subset facilitates various ways of composing spatial entities, including nesting of arbitrary depth.

The ability to represent structured urban information is one of CityGML's key features. However, when it comes to data collection at a large scale (e.g., for a big city), some simplifications should be considered to enhance performance of database access. For Berlin, a simplified data model has been created and then used as basis for the derivation of the actual database structure. In the following, modifications are listed and discussed in detail:

- **Simplified treatment of recursions**
  Recursive database queries are very time consuming, especially if the recursion depth is unknown. In order to guarantee high performance, each database object stores both its direct parent element and its root element. The root element is essential for typical high level queries. E.g., all parts of a building can be obtained by simply accessing those elements storing the building's ID as root ID. This operation can be executed on thematic as well as geometry side. The explicit storage of parent elements allows for reconstruction of the whole recursion tree without information loss.
- **Alternative design of GML geometry classes**
  In CityGML, spatial properties of features are represented by objects of GML3's geometry model, consisting of primitives, which may be combined to form complexes, composite geometries, and aggregates. The usage for CityGML is restricted to a subset of the GML3 geometry package, dealing with the representation of polygonal geometries. This may be Points, LineStrings, Polygons, Solids and all valid geometry collections such as CompositeSurfaces or MultiSolids. For topology and appearance information, CityGML requires identification of geometry parts, even if contained in geometry collections. Spatial databases usually provide data types for the aforementioned geometry types. In fact, only those data types enable spatial queries within these databases. Unfortunately, database implementations of geometry collections such as MultiSurface do not allow for naming and referencing of

the internal geometry parts. This hinders their use for CityGML.
The solution is a simplified geometry model for 2D and 3D geometries as shown in fig. 2.

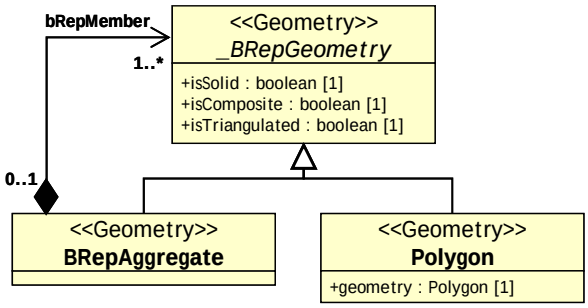

Fig. 2. Simplified modelling of polygon-based GML geometry classes.

The abstract root class is called *_BRepGeometry* and acts as basis for all surface-based geometry objects within the city database. It specialises to two concrete classes Polygon and BRepAggregate which hold explicit coordinates or information on aggregation hierarchy respectively. All surfaces are split into individual polygons such that each is identifiable. Polygon uses a native spatial data type (Oracle SDO_GEOMETRY of type Polygon) to enable spatial queries. Polygons are then aggregated using BRepAggregate, again with each being identifiable. Various flags denote the type of aggregation: isTriangulated denotes a TriangulatedSurface, isSolid distinguishes between surface and solid, and isComposite defines whether this is an aggregate (e.g., MultiSolid) or a composition (e.g., CompositeSolid). Geometric aggregates are arbitrary aggregations of geometry elements which are not assumed to fulfill specific topological constraints. In contrast, composite geometries represent a set of topologically connected primitive geometric objects of same dimension, whose interiors are disjoint. The geometric composite itself must be isomorphic to a primitive geometric object [11]. The recursive relation bRepMember enables nesting of geometry collections.

- **Project specific classes and class attributes**
  The city database of Berlin must also store orthophotos, metadata, and version controls. Since in CityGML this information is not represented, appropriate classes and class attributes have been added.
- **Data type customisation**
  Some of the data types specified in CityGML were substituted by simpler ones to enable a more efficient representation within the database, e.g. code lists and colour vectors were replaced by strings and non-polygonal GML geometry types were mapped to their Oracle-specific data type SDO_GEOMETRY.
- **Reducing multiplicities of class attributes**
  Simple attributes with an unbounded number of occurrences (*) are represented in the database by either an array containing a predefined maximum number of elements or by a data type permitting storage of arbitrary values in

one object (e.g., several values may be represented by using the data type String with a predefined separator to detach elements from each other). Only then such attributes can be included into a single database column.

- **Cardinalities and types of relations**
  To represent relations of cardinality n:m in a database, an additional table is needed, which contains pairs of linked object IDs. With the simplification to 1:n or n:1 relations this additional table can be avoided. Therefore, all relations defined in CityGML were tested for more restrictive interpretation. One result was changing the aggregation between rooms and furniture to a composition, since furniture is always bound to a specific room.

4 Derivation of the relational database schema

GML is inherently object oriented. The GML application schema of CityGML contains specialization and aggregation hierarchies, nested objects, and complex attributes. In order to map these structures to a relational schema different rules have been proposed and discussed in the past. For example, the mapping of specialisation hierarchies onto database tables may follow one of three different mapping methods [13]. Fig. 3 illustrates these methods for the mapping of our simplified geometry representation, introduced in fig. 2 (the bRepMember relation is omitted here).

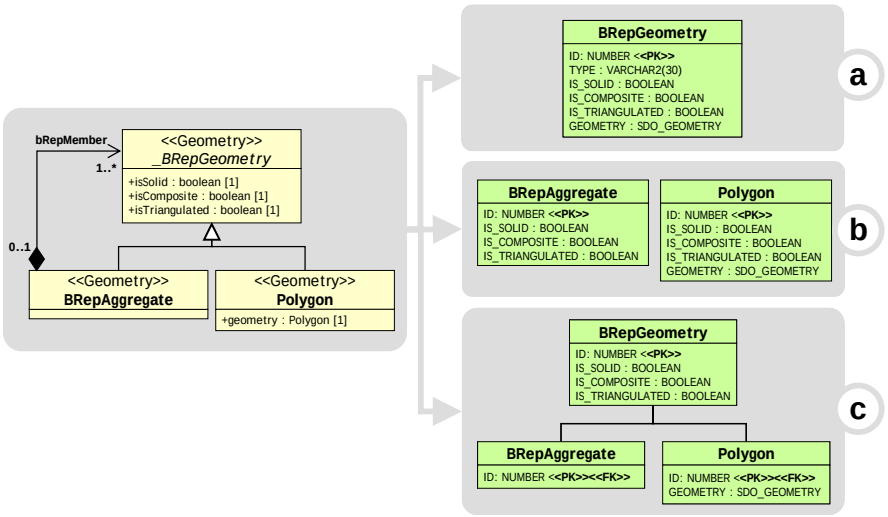

Fig. 3. Mapping a class hierarchy to database tables: (a) mapping all classes to a single table; (b) mapping non-abstract classes to their own tables; (c) mapping each class to a single table.

In order to create space-efficient representations, systems that are capable of automatically deriving relational schemas from GML application schemas typi-

cally employ variants (c) or (b). For example, the two commercial systems GO Loader [8, 12] from Snowflake Software and SupportGIS [4] from CPA Geo-Information follow this line.

For the Berlin 3D geodatabase the mapping was carried out manually allowing careful optimisations. For each CityGML class the mapping alternative was chosen individually. The criteria were the expected number of tuples within the tables, the number of joins to reconstruct the CityGML objects, and the overall complexity of the most important queries (with respect to the CityGML object structure). Emgård and Zlatanova [6] provide a more detailed discussion about the mapping of a 3D information model to a relational database. They outline and evaluate two alternative mappings that are optimised for spatial or semantic queries respectively.

Central CityGML classes were identified that require individual tables. They include CityObject, CityModel, SurfaceData, and major thematic classes such as PlantCover or AbstractBuilding. Remaining classes have been merged into these tables either due to similar structure (e.g., BuildingInstallation and IntBuildingInstallation) or due to inheritance (e.g., Road, Track, Railway, and Square into TransportationComplex). See [9] for CityGML class diagrams.

The overall root class Feature has no associated table. Instead, its attributes GMLID, GML_NAME, and NAME_CODESPACE have been moved to direct subclasses of CityObject. In addition, GMLID has been equipped with an attribute GMLID_CODESPACE. It contains the full path of the originating CityGML instance document or a user-defined value. The codespace is required since GMLIDs are to be unique in single instance documents only. Only the combination of ID and codespace ensures uniqueness within the database. For the class CityObject, the attributes GML_NAME and NAME_CODESPACE have been pushed down an additional level along the inheritance hierarchy. This is due to typical database queries, such as searching for a specific building named “Brandenburger Tor”. Storing the attributes directly in table CityObject would require unnecessary table joins to identify affected buildings.

Finally, the attribute CLASS_ID has been added to CityObject. It helps implementing queries starting in table CityObject, such as by GMLID, by spatial extent (e.g., a bounding box), or through meta information. To facilitate further database scanning, the attribute CLASS_ID provides information on the class affiliation of the identified entries in CityObject and enables direct access to relevant tables.

The implementation of geometry in the 3D geodatabase (fig. 4) follows the first approach (fig. 3 (a)). By avoiding multiple tables the number of joins is reduced, resulting in higher database performance. As explained in section 3 (Simplified treatment of recursions), the bRepMember relation (fig. 2) was reduced to two attributes PARENT_ID and ROOT_ID. Further explanations can be found in the database documentation [2].

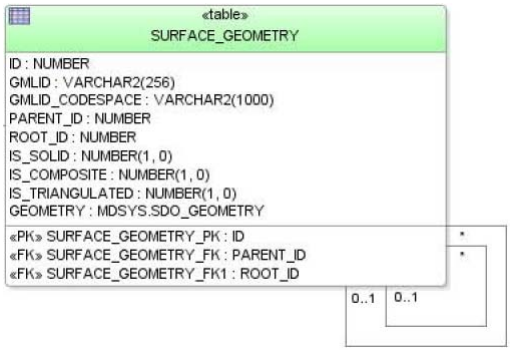

Fig. 4. Implementation of 3D BRep geometry in the 3D geo database. This table stores everything from single Polygons over MultiSurfaces, CompositeSurfaces, TriangulatedSurfaces, up to Solids, MultiSolids, and CompositeSolids. By only using Oracle's SDO_GEOMETRY type for single polygons, every polygon is a tuple with an ID value and may be referenced e.g. from appearance information.

5 Creation of an import and export tool

In this section, we introduce a software tool for the import and export of CityGML datasets for the 3D geo database. In addition to the relational geo database schema, the tool is considered an integral part of an overall solution for the efficient processing, storage, and retrieval of 3D city models.

In this context, the processing of CityGML and GML structures has to face three main challenges:

1. GML provides a powerful and expressive data model and XML is verbose. This results in voluminous encodings of spatial data. Thus, instance documents quickly grow in file size and may exceed main memory limits (e.g., one million buildings in LOD1 require about 7GB of disk space).
2. Objects may be arbitrarily nested leading to complex data structures which have to be efficiently parsed and mapped to according database tables.
3. Property elements may reference their value using XLinks pointing to remote objects within the same or an external document. These Xlinks have to be resolved in order to correctly represent objects within the geo database.

The import/export tool addresses these challenges by employing strategies for the handling of CityGML instance documents of arbitrary file size and the resolving of XLinks. Furthermore, high-performance data processing is achieved based on a multithreaded software architecture. Fig. 5 gives an overview of the import and export tool's implementation. It is composed of three parts. In the middle, the process of binding the underlying CityGML XML schema definition to a Java object model is illustrated, further explained in section 5.1. The top and bottom parts show the process of data import and export, addressed in sections 5.2 and 5.3.

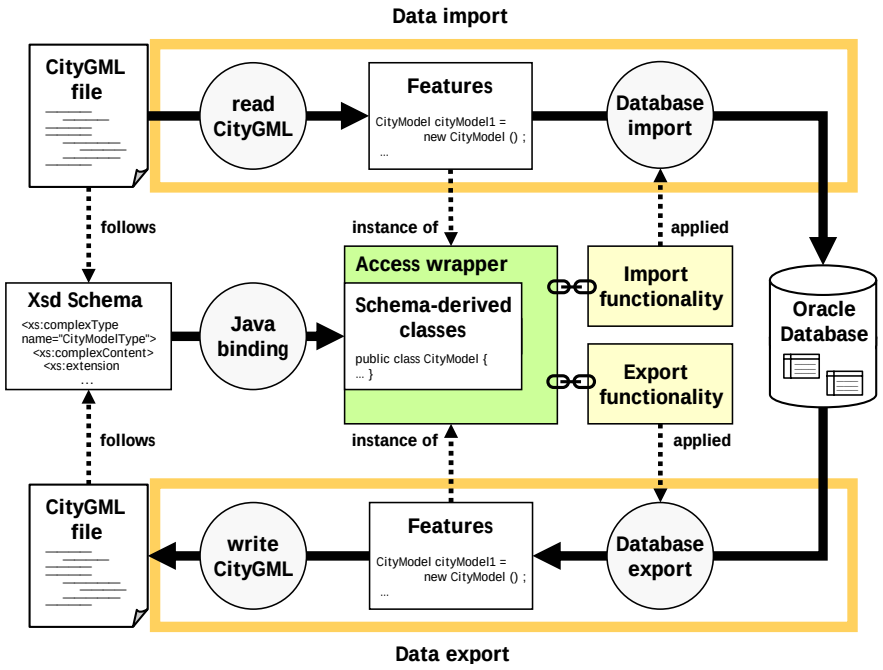

Fig. 5. Import and export tool: Overview of the overall data flow.

Special attention is directed to the management of object IDs, because only a systematic and careful usage of a worldwide unique identifier guarantees consistent updating of geo objects stored in the database. As GMLIDs are optional and unique only within one dataset, all imported objects are either assigned a new worldwide unique identifier (UUIDs) or only objects with missing GMLIDs are assigned such UUIDs. In both cases, the attribute GMLID_CODESPACE described in section 4 is added to every object (including surface-based geometries).

5.1 Software design of the import/export tool

The import/export tool is implemented as a Java application. It employs a chunk processing strategy for XML documents in order to handle instance documents of arbitrary file size. Furthermore, the software design is entirely based on multi-threading allowing for high-performance and concurrent data processing.

Support for CityGML instance documents of arbitrary file size

The processing of large XML documents is realized through a stream-based XML object binding. This approach utilizes two existing Java frameworks for reading and writing verbose XML files. On the one hand, the Java Architecture for XML Binding (JAXB) is employed to facilitate an object-oriented view of XML data. On the other hand, the Simple API for XML (SAX) is used for the stream-based

and event-driven parsing of XML documents. By combining both frameworks, the benefits of each API can be leveraged while limiting their downsides.

The JAXB framework provides a convenient and easy-to-use way to validate and process XML content using Java objects by binding the underlying XML schema definition to a Java object model. It enables storing and retrieving of data in memory in any XML format without the need to implement a specific set of XML loading and saving routines. Moreover, the generated object model captures the structure of XML better than general-purpose APIs like the XML Document Object Model (DOM) or SAX. Like most object-binding APIs, JAXB requires the entire XML stream or file to be present in main memory before data processing can begin. Consequently, memory consumption limits the maximum XML document size.

In contrast, streaming parsers such as SAX, allow for a serial access to XML documents. Each element of the document is passed to the application in sequence of its occurrence via user-defined callback methods. Accordingly, the memory footprint of this event-driven approach is mainly based on the data stored in a single XML element, and thus is always only a small fraction of the size of the entire document. However, stream-based parsing is unidirectional, i.e., previously parsed data cannot be re-read. This hinders a simple and object-oriented handling of complex XML content.

In order to provide both an object-oriented view of XML data and a solution to the data overload problem, the import/export tool implements a two-stage approach: 1) XML documents are parsed using a SAX streaming parser which splits the document into single chunks of manageable size. For each chunk, the occurring SAX events are recorded using a memory buffer. 2) The buffer contents are individually mapped to Java objects for data processing using JAXB. For writing XML documents, the inverse process is applied.

By this means, memory usage is no longer dependent on the document size, but only relates to the amount of data kept in the memory buffer. For CityGML instance documents, reasonable XML chunks are embraced by <cityObjectMember> tags which represent the top-level features within a CityGML model.

In anticipation of future changes, a thin access wrapper is added to the JAXB binding classes. This wrapper abstracts from the underlying binding classes to enable parallel support of CityGML version 0.4.0 and 1.0.0.

Concurrency of data processing

The overall architecture of the import/export tool is based on multithreading, i.e., concurrent execution of multiple interacting computational tasks. Generally, each task within the import/export process of CityGML data is carried out by separate threads. The decoupling of compute bound from I/O bound tasks and their parallel non-blocking processing usually leads to an increase of the overall application performance. In a multi-core environment, threads can even be executed simultaneously on different CPUs.

However, a common and simplistic thread-per-task approach faces disadvantages for a large number of active threads such as thread life-cycle overhead and resource thrashing. For this reason, the import/export application reuses threads

for multiple but homogenous tasks by applying a pooled threads model. Thread-creation overhead is spread over many tasks, and because a thread already exists within the pool when a task is requested, the delay introduced by thread creation is eliminated.

The thread pool model is implemented as a work queue combined with a fixed group of associated worker threads (fig. 6). The work queue is realized as a blocking queue which is a common pattern in multithreaded programming: one thread produces objects, i.e., single tasks to be performed, and places them on a queue for consumption by other threads, which remove them from the queue. The blocking queue implementation is following the producer-consumer design pattern [7] to avoid deadlocks of the associated worker threads.

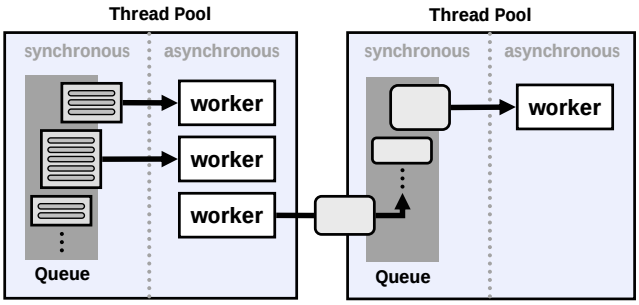

Fig. 6. Thread pools are implemented as work queues with fixed groups of associated worker threads.

Single thread pools can be easily combined to entire process chains. The worker threads within a thread pool concurrently process the tasks placed on their associated work queue, and pass their processing results to the work queue of the subsequent thread pool. This allows for the decoupling of process steps in a modular way, allowing for easy extension by further process steps. Inter-process communication between the process steps is realized by an event-dispatcher thread which is implemented as single-worker pool allowing for asynchronous and synchronous message transfer according to the publisher-subscriber design pattern [7].

The optimal number of worker threads within a thread pool mainly depends on the number of processors available, the nature of the task, e.g., I/O bound or compute bound, and the cost of thread context switching. The import/export tool supports user-defined threshold values and a management unit per thread pool for autonomous adaptation based on the overall workload. Furthermore, the queue size is a determining factor for memory consumption, and thus can be adjusted to specific system configurations.

5.2 The import process

The process of importing CityGML datasets into the database is illustrated in fig. 7. The workflow is implemented by chaining thread pools which cover single

steps of the import process. It comprises thread pools for the chunk-processing of the input XML document (parser thread pool), the conversion of single chunks to JAXB objects representing CityGML top-level features (converter thread pool), and the data processing of these features (importer and XLink thread pools). The associated work queues are also shown in fig. 7.

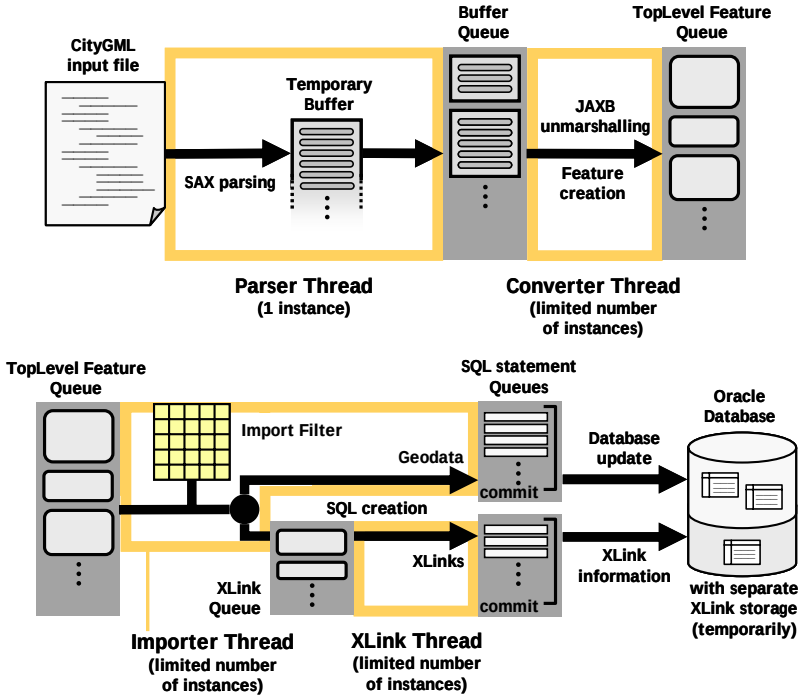

Fig. 7. Import Tool (phase 1) – Detailed workflow from input CityGML instance documents to database storage. XLink references are separately stored in temporary tables to enable XLink resolving in phase 2.

The first two steps of the process chain, i.e., chunk processing of the input data and unmarshalling of CityGML top-level features, have already been discussed in section 5.1. By analyzing the resulting JAXB objects, the subsequent importer thread shown in fig. 7 derives the final SQL statements using the import functionality, which maps the objects to corresponding tables of the relational database schema (section 4). This step can be customized by user-defined import filters. For example, the import may be restricted to a given set of feature types, e.g., only CityGML buildings, or to features located within a geographic bounding box. Filtered objects can be immediately skipped and released from memory.

In order to optimize database response times, SQL statements are precompiled and stored in corresponding Java objects by the importer thread. The precompiled objects can then be used to efficiently execute the same statement multiple times with varying data. Furthermore, SQL statements are collected and only forwarded

to the database if a certain amount of statements has been reached (bulk processing). This allows for an efficient batch processing on the database side. The optimal number of statements within a batch depends on the complexity of the imported CityGML dataset, and thus can be customized.

The multithreaded design of the import tool allows for the concurrent execution of all process steps. For example, the parsing of the XML document is never blocked by threads waiting for database response, and multiple JAXB objects can be processed simultaneously. However, the chunk-processing of the input data also introduces a further level of complexity. In CityGML instance documents, properties of features like relations to other features or geometry objects may be denoted using the XLink concept of GML3. Thus, instead of having the XML content inline within the feature element, the content is given by reference to a remote object identified by its GMLID. Since backward references within the same XML document are allowed, a one-stage approach for resolving of XLinks would require the whole document to be present in memory.

To solve this problem, the import tool employs a two-phase strategy. In the first run (fig. 7), features are written to the database neglecting references to remote objects. However, if a feature contains an XLink, an additional entry is written to a temporary table by a separate XLink thread. This entry mainly contains the referencing feature and the referenced GMLID. Furthermore, the import tool keeps track of every previously encountered GMLID and its corresponding object representation in database. In a consecutive run, only these temporary tables are revisited and queried. Since the entire XML document has been processed at this point in time, valid references can be resolved and processed accordingly. The second phase of the import process is depicted in fig. 8.

By means of this two-phase import process, the advantages of chunk-processing can be kept and a correct storage of CityGML instance documents within the database can be ensured.

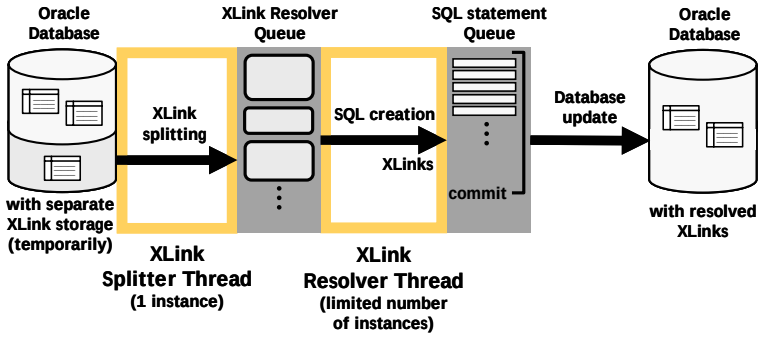

Fig. 8. Import Tool (phase 2) – After phase 1 is completed, the information stored in the temporary tables is queried (splitter thread pool) to resolve XLink references to remote objects (resolver thread pool).

5.3 The export process

The workflow for exporting CityGML datasets from the database is shown in fig. 9. As for the import, the workflow is realized as a process chain combining several thread pools, each of which covers a separate process step.

The first step of this workflow is to query the database according to user-defined criteria. These export filters are identically defined as the import filters, e.g., allowing for spatial queries of features within a given geographic bounding box. Spatial queries are performed within the database itself using the spatial indexing capabilities of Oracle 10G R2 Spatial. Furthermore, the queries are only executed on a single database table in the first run, which only holds general information for all contained features. Thus, the queries can be rapidly processed by the database and returned to the export tool.

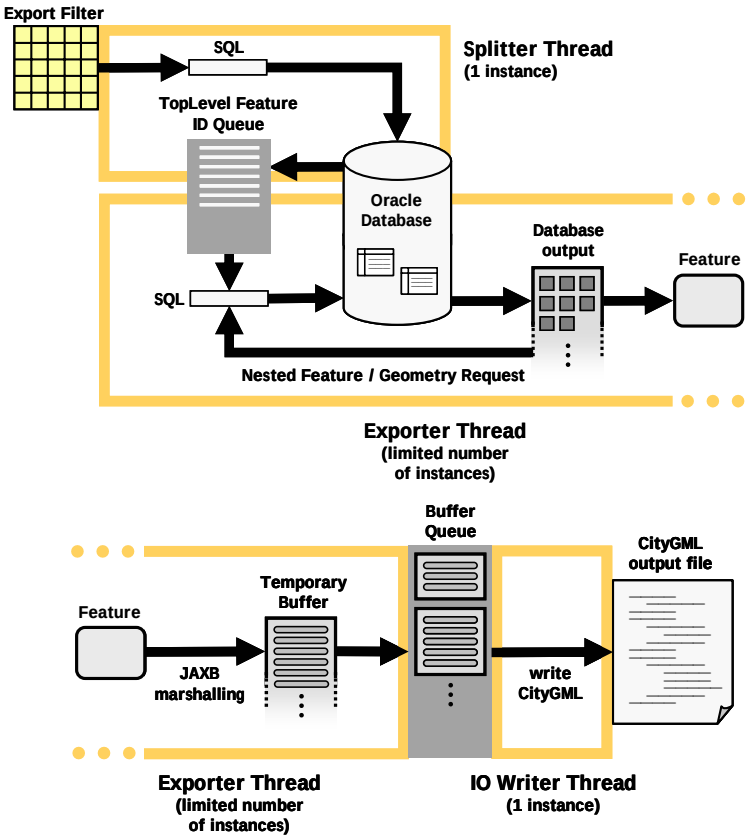

Fig. 9. Export Tool – Detailed workflow from database filtering to the export of CityGML instance documents.

As soon as the first query results are returned, the splitter thread places them on the work queue of the subsequent exporter thread. By this means, the worker

threads of the export pool already start to reconstruct the corresponding CityGML feature objects while the splitter thread still processes the initial database response.

Depending on the type of feature to be reconstructed, the exporter workers have to execute more complex database queries spanning several tables in order to retrieve the necessary feature data. The export functionality provides respective code for each feature type. At this stage, most of the data processing is done by the database server due to efficient use of SQL join statements. Furthermore, if a single worker thread is waiting for database response, it does not block the other threads within the export pool. If all data has been received, it is mapped to corresponding JAXB objects. Also the export tool keeps track of the feature's GMLID in order to avoid duplicate output of objects and to use XLinks instead. The JAXB objects are marshalled to SAX events which are recorded by a temporary memory buffer. The buffered SAX events are placed on the queue of the subsequent thread pool. Afterwards, the export thread continues to work on the next feature.

The last step of the export process, i.e., writing the entire feature data to a CityGML instance document, is carried out by an I/O writer thread. This thread takes the SAX event buffers from its work queue and translates them into XML elements. Again, these file-based processes are decoupled from all other steps of the process chain.

5.4 Performance

Currently, performance has not been measured thoroughly. However, initial tests exhibit good throughput in the following setting:

- **Database server:** 2 x Intel Xeon Dual Core@3GHz, 6GB RAM, SCSI Raid 5 Disk Array, Windows Server 2003 SP2, 100MBit LAN
- **Client running the import/export tool:** Intel Core2 CPU 6600@2.4GHz, 2GB RAM, WindowsXP SP2, 100MBit LAN

The database server was running a default Oracle 10G R2 Spatial installation without manual tunings or additional settings. All import tests have been performed with spatial indexes disabled. Additionally, indexes on table rows containing VARCHAR2 values (e.g., GMLID, GMLID_CODESPACE) were switched off. Unless stated otherwise, Oracle's table versioning was disabled. The performance measurements for the import of different CityGML instance documents are shown in tab. 1.

Tab. 1. Performance measurements for the import of CityGML datasets using the developed import tool. Row headers are as follows: [1] File size, [2] Overall import time, [3] Top-level features per second (please note that these may contain nested features which are not counted here), [4] Workers per thread pool, [5] Imported top-level features / geometries / texture images / Xlinks, [6] Versioning using Oracle Workspace Manager

Dataset	[1]	[2]	[3]	[4]	[5]	[6]
1 mio. LOD1 buildings	7.5GB	1:15h	228	4	1055951 / 11511040 / 0 / 0	off

Dataset	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**
1 mio. LOD1 buildings	7.5GB	2:25h	121	1	1055951 / 11511040 / 0 / 0	off
209 complex LOD3 objects	40.1MB	17s	12	4	209 / 73823 / 0 / 0	off
-“-	40.1MB	24s	9	1	209 / 73823 / 0 / 0	off
-“-	40.1MB	3:20h	0.02	4	209 / 73823 / 0 / 0	**on**
10927 fully textured LOD2 buildings	163MB + 57 MB textures	25min	7	4	10927 / 434714 / 43348 / 391006	off
-“-	163MB + 57 MB textures	42min	4	1	10927 / 434714 / 43348 / 391006	off
-“- (w/o textures)	163MB	6:30 min	28	4	10927 / 434714 / 0 / 391006	off

Corresponding export tests are illustrated in tab. 2.

Tab. 2. Performance measurements for the export of CityGML datasets using the developed export tool. Row headers are as follows: [1] File size, [2] Overall export time, [3] Top-level features per second, [4] Worker per thread pool, [5] Exported top-level features / geometries / texture images / XLinks

	[1]	**[2]**	**[3]**	**[4]**	**[5]**
1 mio. LOD1 buildings	7.5GB	38min	455	10	1055951 / 11511040 / 0 / 0
-“-	7.5GB	1:57h	150	1	1055951 / 11511040 / 0 / 0
209 various and complex LOD3 objects	40.1MB	7s	30	10	209 / 73823 / 0 / 0
-“-	40.1MB	17s	12	1	209 / 73823 / 0 / 0
10927 fully textured LOD2 buildings	163MB + 57 MB textures	4:20min	42	4	10927 / 434714 / 43348 / 391006
-“- (w/o textures)	163MB + 57 MB textures	2min	91	4	10927 / 434714 / 43348 / 391006

6 Conclusions and Future Work

In this paper a relational 3D geo database for the storage of CityGML data was presented. Optimizations on the object model and its mapping to a relational schema were discussed. Furthermore, a concept for multithreaded processing of

large (City)GML files was proposed. The first performance evaluations showed that the usage of concurrent worker threads leads to significant speedups.

The 3D geodatabase has been developed in the course of the project "Geo data management for the Berlin government – The official Berlin 3D city model" [5]. By relying on CityGML's comprehensive data model, members of different communities, such as city planners, architects, surveyors, etc. are provided with a database for semantically rich city models. With the project close-out at the end of this year the entire software will be released as Open Source on [2].

In the future, the database administration tool will be extended by data matching functionality, i.e. buildings in the database that represent the same real world object will be detected and linked automatically. This allows for the exchange of thematic information attached to either of the buildings as well as automated updating procedures. In case of equivalent geometries, one of the objects will be deleted to avoid database inconsistencies.

Finally, there is room for performance optimisations. Important topics not yet covered are logical table and index partitions as well as their optimal distribution on physical volumes.

References

1. 3D city database, version 1, 2006. http://www.3dcitydb.org
2. 3D city database, version 2, 2008. Accessible via http://www.citygml.org
3. Cox, S., Daisey, P., Lake, R., Portele, C., Whiteside, A, 2004. OpenGIS Geography Markup Language (GML) Implementation Specification, Version 3.1.1, OGC Doc. No. OGC 03-105r1, Open Geospatial Consortium
4. CPA Geo-Information, SupportGIS product website, 2008. http://www.supportgis.de
5. Döllner, J., Kolbe, T.H., Liecke, F., Sgouros, T., Teichmann, K., 2006. The Virtual 3D City Model of Berlin - Managing, Integrating, and Communicating Complex Urban Information, In: Proc. of the 25th Urban Data Management Symposium UDMS, Aalborg
6. Emgård, L., Zlatanova, S., 2007. Implementation alternatives for an integrated 3D Information Model, In: Advances in 3D Geoinformation Systems, Springer
7. Gamma, E., Helm, R., Johnson, R.E., 1995. Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, Amsterdam
8. Snowflake Software, GO Loader product website, 2008. http://www.snowflakesoftware.co.uk/products/goloader/index.htm
9. Gröger, G., Kolbe, T.H., Czerwinski, A., Nagel, C., 2008. OpenGIS City Geography Markup Language (CityGML) Encoding Standard, Version 1.0.0, OGC Doc. No. 08-007r1, Open Geospatial Consortium
10. Gröger, G., Kolbe, T.H., Schmittwilken, J., Stroh, V., Plümer, L., 2005. Integrating versions, history and levels-of-detail within a 3D geodatabase, In: Proc. of Int. Workshop on Next Generation City Models, Bonn, EuroSDR publications
11. Herring, J., 2001. The OpenGIS Abstract Specification, Topic 1: Feature Geometry (ISO 19107 Spatial Schema), Version 5. OGC Document Number 01-101
12. Müller, H., Curtis, E., 2005. Extending 2D interoperability frameworks to 3D, In: Proc. of Int. Workshop on Next Generation City Models, Bonn, EuroSDR publications
13. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., 1991: Object-Oriented Modelling and Design, Prentice Hall