MSc thesis in Geomatics for the Built Environment

Combination of CityJSON with PostgreSQL, MongoDB and GraphQL

Karin Staring
2020

Delft University of Technology

MSc thesis in Geomatics

Combination of CityJSON with PostgreSQL, MongoDB and GraphQL

Karin Staring

September 2020

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

Karin Staring: *Combination of CityJSON with PostgreSQL, MongoDB and GraphQL* (2020)

The work in this thesis was carried out in the:

3D geoinformation group
Department of Urbanism
Faculty of the Built Environment & Architecture
Delft University of Technology

Geonovum
National Spatial Data Infrastructure (NSDI) executive committee
The Netherlands
Amersfoort

Supervisors: Stelios Vitalis
Linda van den Brink
Co-reader: Balázs Dukai

Abstract

CityJSON is a JavaScript Object Notation (JSON)-based encoding for a subset of the CityGML data model and an alternative to the CityGML exchange-format. This new encoding reduces the data size and simplifies the usage. In addition, relational and Not only Structured Query Language (NoSQL) databases have integrated JSON. CityJSON has therefore the potential to be stored and perform efficiently in relational and NoSQL databases.

The databases are tested as part of a client-server architecture, because JSON is used in web applications and the 3D city models can not entirely be stored on mobile devices. GraphQL is used as the Application Programming Interface (API) layer between the database and the client, because it has the ability to optimize the usage of the network. This is necessary for location-based web applications on mobile devices to stay functional. This research attempts based on this use case to answer the following question: *How suitable are MongoDB and PostgreSQL for the storage and querying of CityJSON using GraphQL?*

CityJSON has first been mapped to the relational database PostgreSQL and the NoSQL database MongoDB. CityGML has also been mapped to PostgreSQL with 3DCityDB to clarify the impact of different exchange formats. The databases are after that accessed and queried through GraphQL. The queries are based on the selection process of an Augmented Reality (AR) application. The architecture is tested based on the number of queries between the databases and GraphQL, the request sizes, the response sizes and the retrieval times.

The results show that the usage of JSON maps attributes more flexibly than the mapping of 3DCityDB, which can result in less tables/ collections and therefore less joins or queries. On the other hand, querying on a JSON attribute might result in higher retrieval times, but this is not investigated. Additionally, the usage of JSON makes it possible to store fields with varying data types such as the hierarchy of arrays. A difficulty can be that software such as GraphQL is less flexible.

In general, there are no real signs yet that MongoDB and PostgreSQL are not suitable for the storage and querying of CityJSON using GraphQL. Possible signs are that the indexing mechanism with the vertices list can not be stored in MongoDB and only to some extent in PostgresQL, but this might not be a problem since the indexes can be resolved to integer or real coordinates. The vertices list easily exceeds the maximum document size of MongoDB, which is 16 megabyte (MB). PostgreSQL is on the other hand able to store a vertices list up to 1 gigabyte (GB). The attribute `presentLoDs` can not be stored in MongoDB as well, but this can probably be solved with a small adjustment.

The main recommendations of this research are that the Structured Query Language (SQL)/MongoDB Query Language (MQL) queries should be implemented as efficiently as possible without being connected to GraphQL first. This should be done to understand the performance of the databases and to understand the impact of GraphQL better. Besides this, a more general understanding of the suitability for all use cases could be provided with a framework that tests more types of queries.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface	v
AR	Augmented Reality	v
BSON	Binary JSON	1
CRUD	Create, Read, Update, Delete	12
ER	Entity-Relationship	9
EWKT	Extended Well-Known Text	40
GB	gigabyte	v
GML	Geographic Markup Language	5
GUI	Graphical User Interface	35
HTTP	HyperText Transfer Protocol	2
ID	IDentifier	5
JDBC	Java Database Connectivity	36
LoD	Level of Detail	xi
JSON	JavaScript Object Notation	v
JSONB	JavaScript Object Notation Binary	1
MB	megabyte	v
MQL	MongoDB Query Language	v
NoSQL	Not only Structured Query Language	v
NSDI	National Spatial Data Infrastructure	iv
ODM	Object-Document Mapper	9
OO	Object-Oriented	9
ORM	Object-Relational Mapper	9
REST	REpresentational State Transfer	xi
SQL	Structured Query Language	v
UML	Unified Modelling Language	9
URI	Uniform Resource Identifier	5
URL	Uniform Resource Locator	36
XLink	XML Linking Language	5
XML	Extensible Markup Language	1
3DCDM	3D Cadaster Data Model	1

1 Introduction

1.1 Motivation

3D city models are machine-readable representations of the built environment with 3D geometries of common city objects and structures [Zhu et al., 2009]. They were only used for visualization in the past, but now the use cases of them have been enriched by utilizing additional data such as attribute and semantic data [Biljecki et al., 2015]. This data is described in a data model: a conceptual model that orders data elements and relates them to each other in a standardized way. A global standardized data model is necessary to ensure consistency and to use the data interoperable. The CityGML data model is such a data model for 3D city models. It does not only represent the 3D city objects geometrically for visualizations, but it also represents the semantic and attribute data [Open Geospatial Consortium, 2012].

The CityGML data model is encoded as a Extensible Markup Language (XML)-based exchange format called CityGML to exchange data between programs in a structured way [Open Geospatial Consortium, 2012]. The term CityGML is used for both the data model and the exchange format. The CityGML data model is also encoded as a JSON-based exchange format called CityJSON. While XML schemas that represent and validate XML files have additional functionalities compared to JSON schemas, JSON is better readable for humans, it reduces the data size and it matches better than XML with the data structure of most programming languages. Beside the usage of JSON, CityJSON also simplifies the usage of the data model and it reduces the data size due to the design of the exchange format. The representation of the geometries and semantics are for instance simplified, because they are only represented in one-way, while the CityGML data model can represent them in multiple ways. The design also reduces the data size, because it prevents the storage of duplicated vertices for instance [Ledoux et al., 2019].

The exchange formats can be stored in databases. A database can be used to store, index and query the data. The database adds functionalities, namely transaction reliability, such as the ACID properties for SQL databases. It also adds methods, such as indexing, to improve the retrieval time when querying the data in the database [Frank, 1988].

There are different databases developed over the years. The relational database model became, together with SQL, the standard in the 1980's and it is still today. After the year 2000, the term NoSQL databases was introduced to express distributed databases that do not use SQL. They are used in many web applications today, because they are designed for handling large amounts of data and to guarantee an high availability [Berg et al., 2012]. These relational and NoSQL databases have their own advantages and limitations.

There is related work about 3D city models that are stored in databases with XML-based exchange formats. The software 3DCityDB stores CityGML in relational databases. It simplified therefore the CityGML data model and mapped the simplified model to a relational database schema [Kolbe, 2019]. Another data model for 3D city models called the 3D Cadaster Data Model (3DCDM) is mapped to the NoSQL database MongoDB. MongoDB stores JSON documents and therefore the data had to be converted to JSON [Višnjevac et al., 2017].

CityJSON on the other hand already uses JSON, it simplifies the usage of the data model and it reduces the data size. Besides this, both relational and NoSQL databases have integrated JSON. While the NoSQL database MongoDB uses Binary JSON (BSON) documents to insert data, the relational database PostgreSQL is able to use the data types JSON and JavaScript Object Notation Binary (JSONB). CityJSON has therefore the potential to be stored and perform efficiently in relational and NoSQL databases.

1.2 Use case

JSON is mostly used to exchange data in web applications. It is important for all web application to reduce the data size, but especially for applications on mobile devices. Mobile devices are not able to store large amounts of data, while 3D city models contain large amounts of data. Consequently, they are not suited to be stored on mobile devices without any pre-selection of objects. Pre-selection of objects is possible in databases with queries. The

3D city models must therefore be stored in databases in order to use them in combination with mobile applications [Blut et al., 2019].

These mobile applications should therefore use a client-server architecture and often an API layer in between them. A client-server architecture has the advantage that the workload can be divided between the client and the server, but on the other hand the architecture is dependent on the network bandwidth. The bandwidth is the amount of data, which can be transferred across the network, within a certain amount of time.

REST APIs and GraphQL can both serve as an API layer. The API layer is in this case an interface between the client and the server to define the interactions between them and to expose the data from the underlying data source using endpoints and HyperText Transfer Protocol (HTTP) methods. HTTP methods are used for the communication between the server and the client. REST is an architectural concept that sets structured and documented principles for these interactions. GraphQL is an alternative to REST APIs with its own architectural concept.

GraphQL usually operates on one endpoint instead of multiple. This endpoint expresses all capabilities of the API service. A REST API on the other hand uses a developer domain to show all used endpoints. Another difference is that GraphQL only uses GET and POST as HTTP methods, but REST APIs could use more of them. Although they both use HTTP GET and POST to query the data, GraphQL uses its own query language.

These different API layers influence the way in which the databases are queried. 3D city models contain large amounts of data and therefore only the necessary data must be transferred. GraphQL exposes the data through one endpoint to enable hierarchical and structured queries. The queries use a system with object types. These object types map to the data of the underlying data structure such as a database. GraphQL precisely describes which data the clients and servers need from each other and one query could obtain data from multiple underlying data structures due to GraphQL. GraphQL is therefore able to minimize the transferred data and to limit the number of requests. The usage of the network bandwidth can in this way be optimized [Gleison Brito, 2019]. Despite the fact that the network bandwidth increases (i.e. 5G), applications that are using a large amount of data still need to reduce their amount of transferred data to stay functional [Taskula, 2019]. This is especially beneficial for web-based applications that use field filtering and data from multiple underlying data structures [Gleison Brito, 2019].

3D city models can not be stored entirely on mobile devices and therefore a client-server architecture is necessary for mobile applications using 3D city models. CityJSON and GraphQL aim both to minimize the amount of transferred data. Although the architecture does not necessarily differ from other applications, the combination of mobile devices and 3D city models could be used to detect whether the user is inside or outside a building.

1.3 Objectives and research questions

The exchange formats for 3D city models have to be stored in databases, because they can not entirely be stored on mobile devices. The CityGML data model has CityGML and CityJSON as exchange formats. The differences between them are that CityGML does not use JSON, while CityJSON does. As a consequence, CityJSON encodes the CityGML data model differently to increase simplicity and efficiency. Both of them are stored in the relational database PostgreSQL in order to compare them. CityGML is stored in PostgreSQL with the software 3DCityDB and CityJSON is stored in this research.

The databases also have their own advantages and limitations. These dependent on the functionalities and structure of the database. CityJSON is therefore stored in the NoSQL database MongoDB and the relational database PostgreSQL in order to compare the suitability of the databases for the storage, accessing and querying of CityJSON.

The suitability is tested based on the use case: location-based applications for mobile devices. GraphQL is used as API layer to optimize the usage of the network. The architecture is used to tests the performance of the databases in combination with the exchange formats and GraphQL.

One of the challenges of this research is to separate the exchange formats, the databases, GraphQL and the implementations. The research questions are conducted with this in mind.

How suitable are MongoDB and PostgreSQL for the storage and querying of CityJSON using GraphQL?

The sub-research questions:

1. What are the differences between the storage of the exchange formats in PostgreSQL?
2. What are the differences between the storage of CityJSON in PostgreSQL and in MongoDB?
3. What are the differences between accessing and querying the exchange formats in PostgreSQL?
4. How do these differences influence the performance of the exchange formats in PostgreSQL?
5. What are the differences between accessing and querying CityJSON in MongoDB and PostgreSQL using GraphQL?
6. How do these differences influence the performance of CityJSON in MongoDB and PostgreSQL when querying CityJSON using GraphQL?

1.4 Scope

This research stores CityJSON in two databases: the NoSQL database MongoDB and the relational database PostgreSQL. CityGML is also stored in relational databases with 3DCityDB. 3DCityDB stores CityGML in the relational databases PostgreSQL and Oracle. However, CityGML is in this research only stored in the relational database PostgreSQL.

Not all features of CityJSON are investigated. The following features are not investigated in this research:

- `geometry-templates` object, including `geometryInstances`
- `CityObjectGroup`
- `extensions` object
- `address` object
- `MultiPoint` and `LineString` as geometries
- `appearance` object including `materials` and `textures`
- city objects other than `Building`, `BuildingPart` and `TinRelief`
- geometries that are higher than LoD 2

These features are not investigated, because the first five features only occasionally occur in the CityJSON files. Therefore, these properties and geometries are considered uncommon. The `appearance` object on the other hand is considered common, but it is excluded due to its complexity. The last two features are due to the used datasets, because they only contain `Building`, `BuildingPart` and `TinRelief`, and the geometries are only represented in LoD 1 and LoD 2.

The use case is used to test the performance of the database. The use case does not use all types of queries that can be send to the database, because no aggregate queries and 3D spatial operations are used. The scope is also limited to the retrieval time performances of the databases instead of the availability and the transaction reliability.

1.5 Report structure

The research exists of seven chapters of which the first one has been the introduction. The second chapter outlines the theoretical background and the third chapter analyses the implementation of CityGML in PostgreSQL with 3DCityDB. The fourth chapter explains the methodology and the fifth chapter the implementations of the methodology. The results are presented in chapter six and the conclusions are drawn in chapter seven based on the developed methodology, the implementations and the results.

2 Theoretical Background

2.1 3D city models

3D city models are machine-readable representations of the built environment with 3D geometries of common city objects and structures [Zhu et al., 2009]. 3D city models were mainly used for visualisation in the past, but today they are also used for tasks beyond that. They have been enriched with additional data such as semantic and attribute data [Biljecki et al., 2015].

2.1.1 The CityGML data model

A data model is a conceptual model that orders data elements and relates them to each other in a standardized way. A global standardized data model is necessary to ensure consistency and to use the data interoperable. The CityGML data model is such a data model for 3D city models. It is an open standard for 3D city models designed with an object-oriented approach. [Open Geospatial Consortium, 2012].

The CityGML data model could be decomposed in a core-module and sub-modules such as bridge, building, city-furniture, cityobjectgroup, generics, landuse, relief, transportation, tunnel, vegetation and waterbody. Each sub-module covers one of the most common city objects with their attributes, geometries and semantic properties. Although is not recommended, each object can contain a different reference system.

Attributes, that are not explicitly specified in the schema, are modelled as generic attributes. The data type of generic attributes must be specified. The data type of the attribute value may be String, Integer, Double, Uniform Resource Identifier (URI), Date, and gml:MeasureType.

The geometries of the objects are compliant with a subset of the geometry definitions in ISO 19107: `MultiPoint`, `LineString`, `MultiSurface`, `CompositeSurface`, `Solid`, `MultiSolid` and `CompositeSolid`. They are represented in the Geographic Markup Language (GML)3 exchange format for geometries, which consists of geometric primitives that may be combined to form aggregate or composite geometries. XLinks are used to store topological relationships and to share primitives, aggregates or composites. They can therefore have their own unique IDentifier (ID)s.

The geometric and semantic properties are structured in five consecutive LoDs. LoD 0 represents for instance the terrain (possibly with a texture) and LoD 4 represents the buildings interiors [Open Geospatial Consortium, 2012].

2.1.2 The CityGML exchange formats

There are two exchange formats of the CityGML data model. CityGML is both the data model and a XML-based exchange format. The exchange format is based on the CityGML 2.0 data model which uses GML version 3.1.1 [Open Geospatial Consortium, 2012]. A subset of the CityGML data model is also encoded to a JSON-based exchange format called CityJSON, which is also based on the CityGML 2.0 data model [Ledoux et al., 2019].

The subset does not include all features, which means that CityJSON does not support LoD 4, multiple reference systems in the same dataset, individual surface IDs and XML Linking Language (XLink)s to store topological relationships and to share geometries. These features are related to this thesis, but more of them are described at `https://www.cityjson.org/citygml-compatibility/`.

Another difference is that JSON is used instead of XML, because most applications use JSON instead of XML to exchange data in web applications. JSON has a smaller data size and is developer friendly. It is developer friendly, because it is human readable and it matches better than XML with the data structure of most programming languages. It is also easier to parse and generate for computers.

CityJSON uses key-value pairs instead of XML tags. Figure 2.1 and figure 2.4 show the difference between them for the property `lod2MultiSurface`.

```
<lod2MultiSurface>
                ....
</lod2MultiSurface>
```

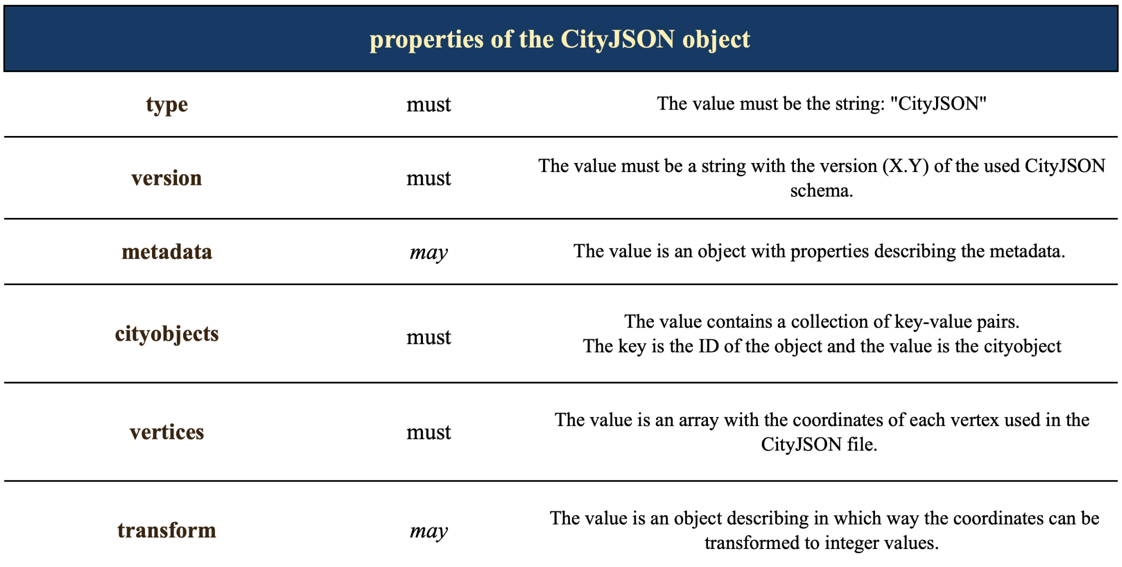
Figure 2.1: `MultiSurface` feature of LoD 2 in CityGML

```
 {
    "type": "MultiSurface",
    "lod": 2
}
```

Figure 2.2: `MultiSurface` feature of LoD 2 in CityJSON

Beside using JSON and the subset, there are also choices made to compress and simplify the exchange format [Ledoux et al., 2019]. These choices are explained in section 2.1.3.

2.1.3 CityJSON

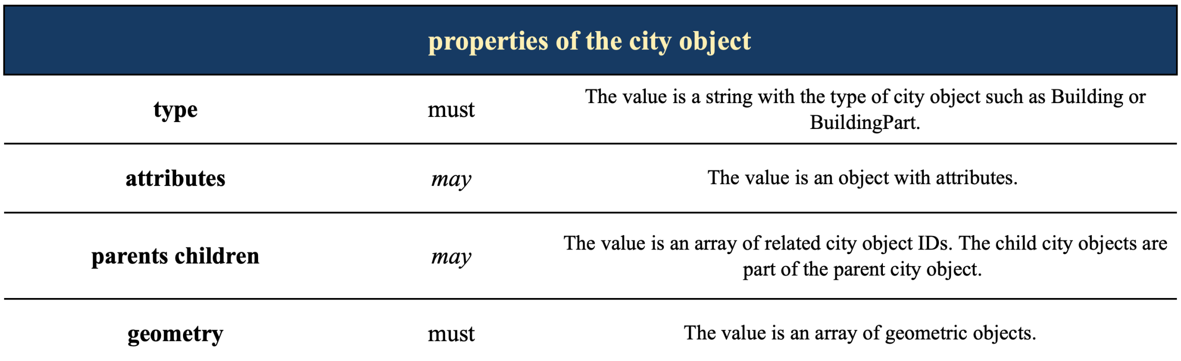


properties of the CityJSON object		
type	must	The value must be the string: "CityJSON"
version	must	The value must be a string with the version (X.Y) of the used CityJSON schema.
metadata	*may*	The value is an object with properties describing the metadata.
cityobjects	must	The value contains a collection of key-value pairs. The key is the ID of the object and the value is the cityobject
vertices	must	The value is an array with the coordinates of each vertex used in the CityJSON file.
transform	*may*	The value is an object describing in which way the coordinates can be transformed to integer values.

Figure 2.3: The properties of the CityJSON object

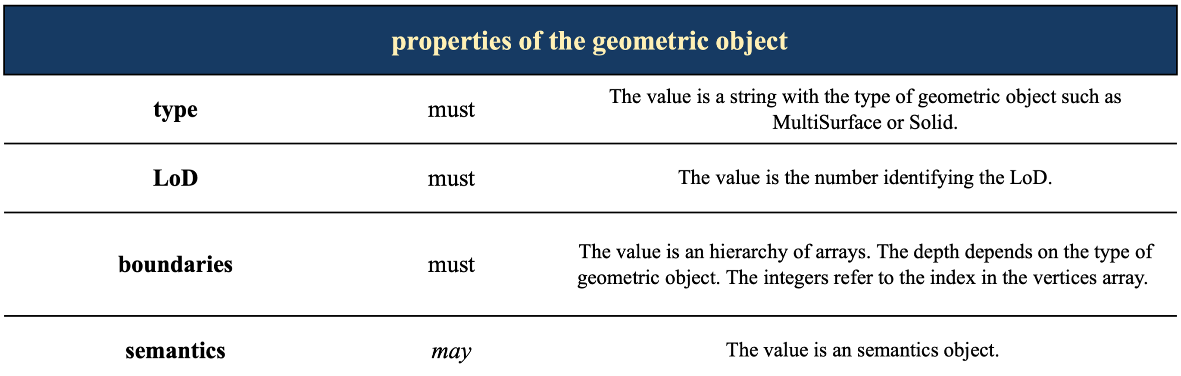
The data is stored as objects with properties and values as could be seen in figure 2.3, 2.4, 2.5, 2.6 and 2.7 [Ledoux et al., 2019]. The first level of the CityJSON file contains the entire CityJSON object. The property `metadata` usually contains a reference system that always applies to the entire CityJSON file as described in section 2.1.2. The property `cityobjects` contains a collection with key-value pairs. The key is the ID of the city object and the value is the city object. The property `attributes` handles normal attributes and the generic attributes. The property `geometry` of the city object contains an array of geometric objects. The property `semantics` of the geometric object contains a semantics object. The property `surfaces` of the semantics object contains an array of semantic surface objects. While there are 26 ways to represent a polygon using GML3 according to this blog `https://erouault.blogspot.com/2014/04/gml-madness.html`, CityJSON only allows one way to represent the geometric and semantic objects. In summary, CityJSON simplifies the CityGML data model by representing the geometric and semantic objects in one way and it considers generic attributes as normal attributes.

Another difference is related to the way in which CityJSON stores the vertex coordinates. CityGML stores the



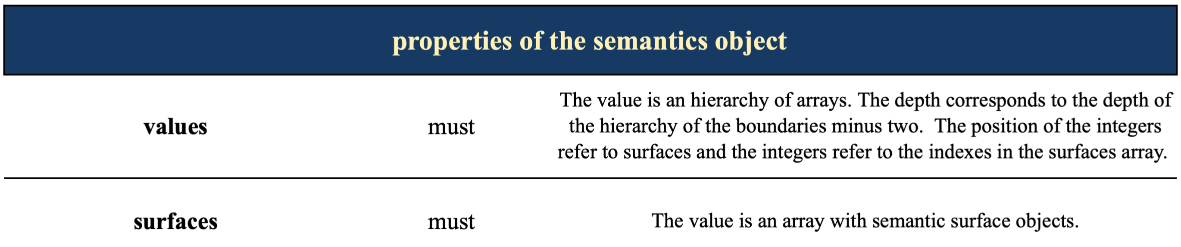
properties of the city object		
type	must	The value is a string with the type of city object such as Building or BuildingPart.
attributes	*may*	The value is an object with attributes.
parents children	*may*	The value is an array of related city object IDs. The child city objects are part of the parent city object.
geometry	must	The value is an array of geometric objects.

Figure 2.4: The properties of the city object



properties of the geometric object		
type	must	The value is a string with the type of geometric object such as MultiSurface or Solid.
LoD	must	The value is the number identifying the LoD.
boundaries	must	The value is an hierarchy of arrays. The depth depends on the type of geometric object. The integers refer to the index in the vertices array.
semantics	*may*	The value is an semantics object.

Figure 2.5: The properties of the geometric object

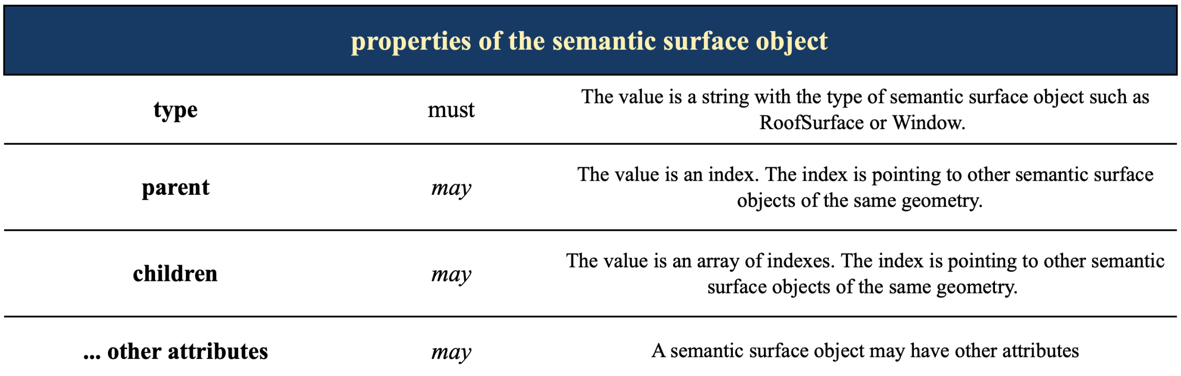


properties of the semantics object		
values	must	The value is an hierarchy of arrays. The depth corresponds to the depth of the hierarchy of the boundaries minus two. The position of the integers refer to surfaces and the integers refer to the indexes in the surfaces array.
surfaces	must	The value is an array with semantic surface objects.

Figure 2.6: The properties of the semantics object

vertex coordinates inside the geometry, but CityJSON stores them in a separate global list. The geometry contains indexes that refer to the vertex coordinates of the list. The file is in this way compressed, because vertices are not duplicated or duplicated vertices could be removed. It also stores topological relationships explicitly to make spatial operations more robust and faster. This means that geometries can share vertices, but whole geometries can not as described in section 2.1.2. An exception are the geometries of the `geometry-templates` object, but those are outside the scope of this thesis.

The property `transform` in figure 2.3 is used to represent the vertex coordinates as integers. The `transform` object transforms the integer coordinates to the real ones. The file is compressed with the transform object and the coordinates are made more robust. The file is compressed, because integers instead of float are used to represent the coordinates. The coordinates are more robust, because integers are not prone to rounding [Ledoux et al., 2019].



properties of the semantic surface object		
type	must	The value is a string with the type of semantic surface object such as RoofSurface or Window.
parent	*may*	The value is an index. The index is pointing to other semantic surface objects of the same geometry.
children	*may*	The value is an array of indexes. The index is pointing to other semantic surface objects of the same geometry.
... other attributes	*may*	A semantic surface object may have other attributes

Figure 2.7: The properties of the semantic surface object

2.2 Databases

A database is an organized collection of data. They are developed to store, index and query data. A database is managed by a database management system. The database management system is responsible for creating and managing the database. The database and the management system together are the database system, but often it is called a database as well. Databases have to offer the following functionalities [Frank, 1988]:

- A database schema is the logical description of the database, which is needed as an interface between the database and the applications.
- A database must also standardize the access to the data files with a query language. The following mechanisms can be implemented to achieve a sufficient retrieval time: index, cluster and buffer.
- Multiple users should be able to access the database simultaneously, but they should only be able to execute operations if they have the authority to do them.
- Transaction reliability has to be guaranteed to support strongly consistent data.
- The data storage and data access must be separated, because adaptions to the amount of data stored (scaling up or down) must not influence the availability of the database.

In the 1960's, only two kinds of models for databases were developed. These were the network model and the hierarchical model. The hierarchical model represents the relationships between records as a tree with a parent-child structure. The network model represents it as a graph with references.

The relational database model was designed by Ted Codd in the 1970's. Here, he disconnected the logical description of the database from the data storage. This concept improved the data management. Together with SQL, it became the standard in the 1980's. At the same time, the entity-relationship model was proposed by Peter Chen in 1976 to let designers focus on the application instead of the database structure. During the 1980's, the concept of the object-oriented database was also developed. The object-oriented database represents data files in the form of objects using object-oriented programming, which reduces the number of relations.

The term NoSQL was firstly introduced by Carlo Strozzi in 1998 for his relational database that did not use SQL. Later, the term was used by Eric Evans to express distributed databases [Berg et al., 2012]. Nowadays, there are used to handle large amounts of data and for applications that require an high availability. These requirements are both applicable to many web services.

The concepts related to data modelling are explained in section 2.2.1. Relational databases and NoSQL databases are further explained in section 2.2.2 and 2.2.3.

2.2.1 Data modelling

Data modelling is the creation of a data model. A data model is a conceptual model at the highest level that orders data elements and relates them to each other in a standardized way as described in section 2.1.1. The conceptual model is the most abstract form of the model. Entity-Relationship (ER) and Object-Oriented (OO) notations (e.g. Unified Modelling Language (UML) class diagrams) are often used to communicate the data model to others.

The ER notation is mostly used to model data in relational databases [Chen, 1976]. It models the entities, attributes and relationships. An entity is approximately the same as an object. There are dependent and independent entities. Dependent entities can not exists on their own, while independent entities can. An entity can be related to another entity. These relations are called associations. There are four types of associative relationships: one-to-one, many-to-one, one-to-many and many-to-many. Entities and relationships can both have attributes. An entity can also be a sub-class of another entity. This is called an inheritance relationship or generalization [Chen, 1976].

Beside the conceptual model, there are also logical models and physical models. The conceptual model is translated in a schema. This can be a database schema or the GraphQL schema. NoSQL databases often do not require a predefined database schema. This means that not all entities and attributes have to be modelled on forehand. It is however still useful to use a database schema, because it defines how the data is structured in the database. The data structure influences the way in which the data is queried.

GraphQL is a query language, not an Object-Relational Mapper (ORM) or an Object-Document Mapper (ODM). An ORM or ODM is needed between GraphQL and the database to translate the GraphQL query in a query that interacts with the database.

2.2.2 Relational databases

A relational database is a database that is organized according to the relational database model. The model uses tables and unordered named-tuples. Each cell in a table is a field and a table exists of columns and rows. A row contains a tuple that represents a set of related data. The tuples are not ordered, but they fit into columns with attribute names. Ordered tuples on the other hand would retrieve data based on the attribute number, i.e. give me the first attribute. This would be problematic when the second attribute is removed. Applications that use the third attribute would have to be adapted, because the third attribute would not be the third attribute anymore [Meier and Kaufmann, 2019].

Relational databases store data on one node. A node could for instance be a client or a server. Relational databases scale vertically, which means that the node must be adapted to the amount of data stored. This might be difficult, because the RAM or CPU of the node has to be increased for that. Another option is to add manual sharding as an additional feature to the relational database, but this is complex for relational databases. Sharding is the partitioning of large volumes of data across multiple nodes [Oussous et al., 2015].

The database management system uses SQL to query and manipulate the relational database. The database management system also assures the ACID properties. The ACID properties ensure transaction reliability in the following ways [Frank, 1988]:

- Atomicity means that all statements inside a transaction are committed either fully or not. When a transaction is not committed fully, the transaction is rolled back.
- Consistency guarantees that transactions never observe or cause inconsistent data.
- Isolation keeps transaction separated from each other until they are finished. Otherwise, they would interfere with each other. This means that transactions are not aware of other transactions happening at the same time.
- Durability means that once the transaction is completed, the changes in the database will be saved permanently. It might happen that the server shuts down before a transaction is stored permanently. Then this property guarantees that the changes will be kept in such a way that the server can recover.

The retrieval time may be affected by the ACID properties due to blocking and deadlocks. Blocking means that when a SQL connection with records is made, these records will be locked. Blocking in the wrong place can slow down the performance, because other connections have to wait for the record to be released. Deadlocks are contradicting locks. For instance, A waits for B to finish and B waits for A to finish. In this way, the records will never be unlocked [Fritchey, 2018].

Relational database rules

A relational database schema contains a logical descriptions of the tables, the relationships between the tables and integrity constraints, which are conditions to maintain the quality.

The relational database rules are based on the ER notation described in section 2.2.1 [Meier and Kaufmann, 2019]:

- Entities are mapped to separate tables with unique primary keys. One or more primary keys uniquely identify the row. The unique primary key(s) could be the entity ID or a sequence ID. The remaining properties of the entity are attribute columns.
- Relationships can be specified with primary and foreign keys. The primary key can be reused as foreign key in another table to create a relationship between them. This concept could be used for dependent entities, one-to-one relationships, many-to-one relationships and one-to-many relationships.
- The many-to-many relationship can be created with a separate table. The separate table contains the primary keys of the related tables as foreign keys. Properties of the relationship can be added as attributes to the created table.
- The inheritance relationship reuses the primary key of the superclass in the subclasses. This means that the superclass contains the primary keys of the subclasses and that the primary keys of the superclass are identical to the primary keys in the subclasses.

Since this thesis is about storing JSON, the two data types that are related to JSON of the relational database PostgreSQL are further explained:

- JSON stores JSON data as an exact copy (as character string) in a JSON field, which must be parsed each time the JSON field is queried. Parsing means breaking down the character string in meaningful pieces of data that the computer uses to perform tasks.
- JSONB converts the JSON data to its parsed form and stores it in a JSONB field. In this way, the JSONB field does not have to be parsed anymore.

The data type JSON is therefore more efficient for storing and querying complete JSON documents than JSONB, but JSONB queries parts of the JSON document more efficient. Additionally, JSONB supports more operators than the data type JSON [Petkovic, 2017].

2.2.3 NoSQL databases

NoSQL databases were previously associated with large amounts of data, but in reality NoSQL databases are associated with Big Data. Big Data has just as NoSQL databases no formal definition yet, but Big Data could be seen as a large amount of data (volume), with flexible data structures (variety) and real-time processing (velocity). Big Data is according to this definition not only about the volume of the data, but also about the structure and processing [Meier and Kaufmann, 2019].

There are many different NoSQL databases, but they have in common that the query language is not only SQL and they are mostly distributed databases. A distributed database is a cluster of nodes. This means that NoSQL databases scale horizontally by adding more nodes in a cluster environment when the data size increases [Oussous et al., 2015]. The data can be duplicated. This means that the data is duplicated over multiple nodes to guarantee availability and parallel computations. This is a one time event. The data is then also replicated. Replication means that the manipulations are logged and updated over all nodes. This is a continuous process.

There are NoSQL databases with strong consistency, which means that the database is always consistent. Weak consistency means that the replication process has a delay [Meier and Kaufmann, 2019].

There are many different NoSQL databases, but the main categories are key-value, wide-column, graph and document databases. A key-value database is a collection of key-value pairs and the key is the ID. It is not possible to index or query based the data on the values, which makes key-value databases only suitable for applications that use the key to access the data.

Wide-column databases are extended key-value databases. The value contains an hierarchy of key-value pairs. The hierarchies are column-families. A column-family contains columns that are usually accessed together. This makes it possible to index and query them based on the keys and the column-families. It is however more difficult to use a wide-column database for an application with an evolving database schema, because of the predefined column-families.

Document databases are extended key-value databases as well in which the value is represented as a document encoded in standard semi-structured formats such as XML, JSON, or BSON. These documents are organized into collections. Document databases support indexes and query functionalities based on attributes and values of collections. The database schema can evolve over time due to the insertion of JSON documents.

Graph databases do not focus on entities such as tables and documents, but on traversing entity relationships. These databases are based on the theory that entities are vertices and the relationships are edges. They are suitable for complex queries over highly connected entities [Davoudian et al., 2018].

Document database rules

The description of the document database rules are based on MongoDB, but they are probably applicable to other document databases as well. MongoDB contains collections, BSON documents and fields [MongoDB, 2020]. There are differences between BSON from MongoDB and JSONB from PostgreSQL according to https://www.airpair.com/postgresql/posts/sql-vs-nosql-ko-postgres-vs-mongo. They are both binary accessible formats, but they support different data types. A collection contains documents and a document contains fields. These fields are attribute-value pairs. The values can be any of the BSON data types. These documents do not have a schema. This means that the fields of the documents can differ and that the data type of a field can vary across documents as well.

There are however certain rules that apply to document databases. They can be applied to the ER notation of section 2.2.1 The document database rules are:

- Embedded documents are used to represent related data. An embedded document is a document inside another document. Data that is queried together should be mapped together.
- Particular relationships can not be represented as embedded documents, because they are many-to-many relationships or the duplication or replication of the data would lead to implications. Although references must be avoided in document databases, it is better for these relationships to use references due to the maintenance issues or the many-to-many relationship.

MongoDB uses different types of references. It uses manual references and DBRefs. The difference between them is that manual references save the `_id` field from another document as reference. This can be an `ObjectId` or a string. These references are used to reference inside a collection. DBRefs make it on the other hand possible to reference between collections. They store the `_id` field, the collection name and, optionally, the database name. In both cases, additional queries have to be done to retrieve the referenced documents or in other words to dereference the references [MongoDB, 2020].

Schema validation in MongoDB

Schema validation is possible in MongoDB using the $jsonSchema operator [MongoDB, 2020]. The $jsonSchema operator is used to check if documents match the specified JSON schema. The following features of JSON schemas are however not supported in MongoDB:

- Hyper-text and hyper-media definitions allow JSON data to be understood as hyper-text. Hyper-text is text that links text to other texts. Hyper-media is hyper-text, but then it includes graphics, videos and sounds.
- The keywords $ref, $schema, default, definitions, format, id and other unknown keywords

- The integer type can not be used. The BSON data types must be used with the bsonType keyword instead [MongoDB, 2020].

Database sizes in MongoDB

The data can be compressed on disk and uncompressed. Different compression mechanisms are used for the collections and for the indexes by default. The data in the data/db directory can be larger than the data inserted into the database. This phenomenon occurs, because MongoDB reserves space for the data files to avoid file fragmentation. The size of the data files do therefore not necessarily represent the size of the data in the database. Another reason can be that the data directory contains journal files. These files store write operations on disk before they are applied to the database.

2.3 Client-server architectures

The client-server architecture classifies devices in clients and servers. The clients make requests and the server provides responses to these requests. It is a program that listens to the connections to receive requests. The World Wide Web and Email use the client-server architecture. Advantages of using the client-server architecture are that the workload between the server and the client can be divided, the data is stored centrally and it provides more data integrity. The HTTP protocol is almost always used for the communication between the server and the client. There are different HTTP methods (i.e. GET, POST, PUT and DELETE) to access data on the specified resource. The most common methods are HTTP GET and HTTP POST. They can both be used to request the specified resource with a query. HTTP GET contains the query in the URI and HTTP POST in the body of the request.

2.3.1 REST

REST is an architecture style that sets structured and documented principles for the communication between applications. REST APIs serve as an API layer between the clients and servers. The interface is mostly available on the developer domain of the website. The developer domain shows the available URIs. These URIs give access to specified resource endpoints. Endpoints contain data from the underlying data structures. Although objects and field can be queried using HTTP POST and GET, REST APIs usually use different HTTP methods for different Create, Read, Update, Delete (CRUD) operations. REST APIs require versioning. This means that the major version number must be included in the URI/ request header and in the response header. Versioning is needed when the database schema changes, because the clients have to integrate the new version in their application code during the transition period. The transition period is the period in which the old version is still available. In case the code is not changed during the transition period, the changed database schema might break the application on the client-side [Schellevis et al., 2019].

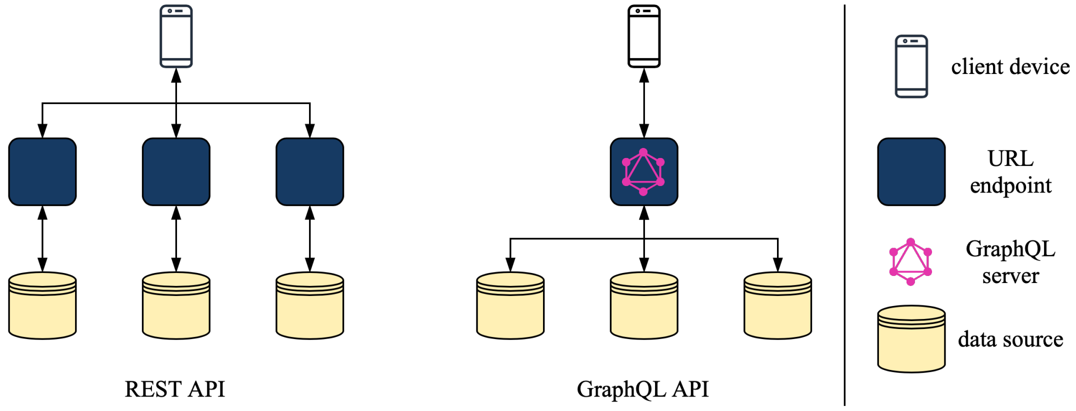


Figure 2.8: A REST API and a GraphQL API

2.3.2 GraphQL

GraphQL is a query language and a server-side runtime. The runtime executes instructions to perform the queries while the application is running. The GraphQL server usually operates on one endpoint and it handles HTTP GET and POST methods.

GraphQL can increase the network performance by reducing the amount of transferred data and the number of requests. REST APIs on the other hand turn easily into multiple non-specific endpoints as can be seen in figure 2.8, because they try to reduce the number of request and they try to avoid slightly different endpoints. GraphQL often uses only one endpoint and the returned `fields` can be specified by the client. This means that no redundant `fields` have to be returned [Gleison Brito, 2019]. Even though REST APIs also allow field filtering, the number of requests could still be more for data from multiple endpoints [Taskula, 2019]. An higher amount of requests increases the pressure on the network and therefore decreases the network performance [Guo et al., 2018].

Other advantages are the possibility to analyze the requests of the clients. These analyses can give inside in the specific needs of the users. Also new fields can be added to GraphQL object types and deprecated without breaking the architecture. Additionally, the GraphQL schema can be used directly for introspection. Introspection means that information about the queries that the current schema supports can be returned. This allows the client to inspect the GraphQl object types of the current GraphQL Schema specifically [Gleison Brito, 2019]. It is however more difficult in GraphQL to implement catching mechanisms and to avoid expensive queries. GraphQL could therefore be unnecessary complex for applications that contain structured and consistent data over time. Another disadvantage of GraphQL is that handling errors can be more difficult due to the fact that queries always return a HTTP status code of 200. The JSON response will have a key named `errors` in case of an unsuccessful request.

The client can query the servers from the underlying data structures represented by the GraphQL schema via the GraphQL server. It enables the server and client to precisely describe which data from their data model the client and servers need from each other. The GraphQL schema is a multi-graph with nodes and edges. Nodes are GraphQL object types with fields. Fields have a name and data type. The data type can be Int, Float, String, Boolean, Null, Enum, List and a GraphQL object type. The data type Enum specifies the valid responses for a field. An edge appears when a field has another GraphQL object type as data type. The GraphQL query type is for the client the entry point of the GraphQL server and it could expose which GraphQL object types can be queried and which arguments can be provided.

```
 { PostByTitleAndAuthor{
    post(id: ”1000”){
        title
        author{
            name
        }
}}}
```

Figure 2.9: Example of a GraphQL request [Gleison Brito, 2019]

```
 { ”data”: {
    ”post”:{
        ”title”: ”GraphQL: A data  query language”
        ”author”:{
            ”name”: ”Lee Byron”
        }
}}}
```

Figure 2.10: Example of a GraphQL response [Gleison Brito, 2019]

The example query in figure 2.9 contains the field `title` of the GraphQL object type `post` and the field `name` of the GraphQL object type `Author`. There is a relationship or also called an edge between the GraphQL object type `Author` and the object type `post`. The request returns the GraphQL object type `post` where the `id` = ‘1000’, because the `id` is provided as argument. The response can be seen in figure 2.10. The GraphQL schema needs resolver functions to specify the returned data. Resolver functions resolve fields of the GraphQL object type. These functions are called each time the field is queried. A resolver function can receive arguments that could be used to resolve the fields.

Besides the GraphQL query type, the GraphQL mutation type can be used to insert or modify the underlying data structure of the GraphQL object types. GraphQL also contains the interface type and the union type. The interface type is similar to a GraphQL object type, but it is no GraphQL object type on its own. The GraphQL object types can implement an interface type to inherit the fields of the interface type. The union type indicates that a field can return more than one GraphQL object type. Only GraphQL object types are allowed to be members of the union type [Foundation, 2020].

2.3.3 Location-based applications for 3D city models

3D city models can be used in combination with the user's position on mobile devices. These large 3D city models can however not entirely be stored on mobile devices due to the limited hardware. Only a preselection of the 3D city model can be stored. Blut et al. (2019) established a selection process to enable AR applications [Blut et al., 2019]. AR uses the existing environment and overlays new digital information on top of it in real-time. Examples of using a 3D city model in combination with AR are the visualization of planned buildings on parcels of land, hidden building parts like cables and historical buildings. When poses become tightly coupled to the 3D city model, new spatial data can also be captured.

They established a selection process and used a XML pull parser and a local spatiaLite database, because the network connection can be unstable. However, they also mentioned the possibility to use a client-server architecture in combination with a local database.

The selection process can be seen in figure 2.11. The selection process can be used to visualize the surroundings of the user based on the user's position. The user's position on a mobile device is provided in a global reference system. The sensor information is however not easily available in a global reference system [Blut and Blankenbach, 2020]. This is probably the reason that the selection process only uses the user's position.

According to figure 2.11, the user's position is used to select the relevant city model. The radius around the user's position is used to reduce the number of buildings. After that, the question is whether the user is inside or outside a building. This can be detected based on the user's position and the 3D city model. If the user is outside, all exterior parts of the buildings within the radius are returned. If the user is inside, the interior parts of one building are returned such as a room. These geometries are visualized.

After the returned geometries are visualized, a ray casting algorithm can be used to select or mutate an object. The user can cast a ray from a particular point in 3D space and a ray casting approach can determine the intersections between the ray and the object. The intersection returns the ID of the object, which enables an ID-based reference to the database. In this way, additional information about the object can be retrieved [Blut et al., 2019].

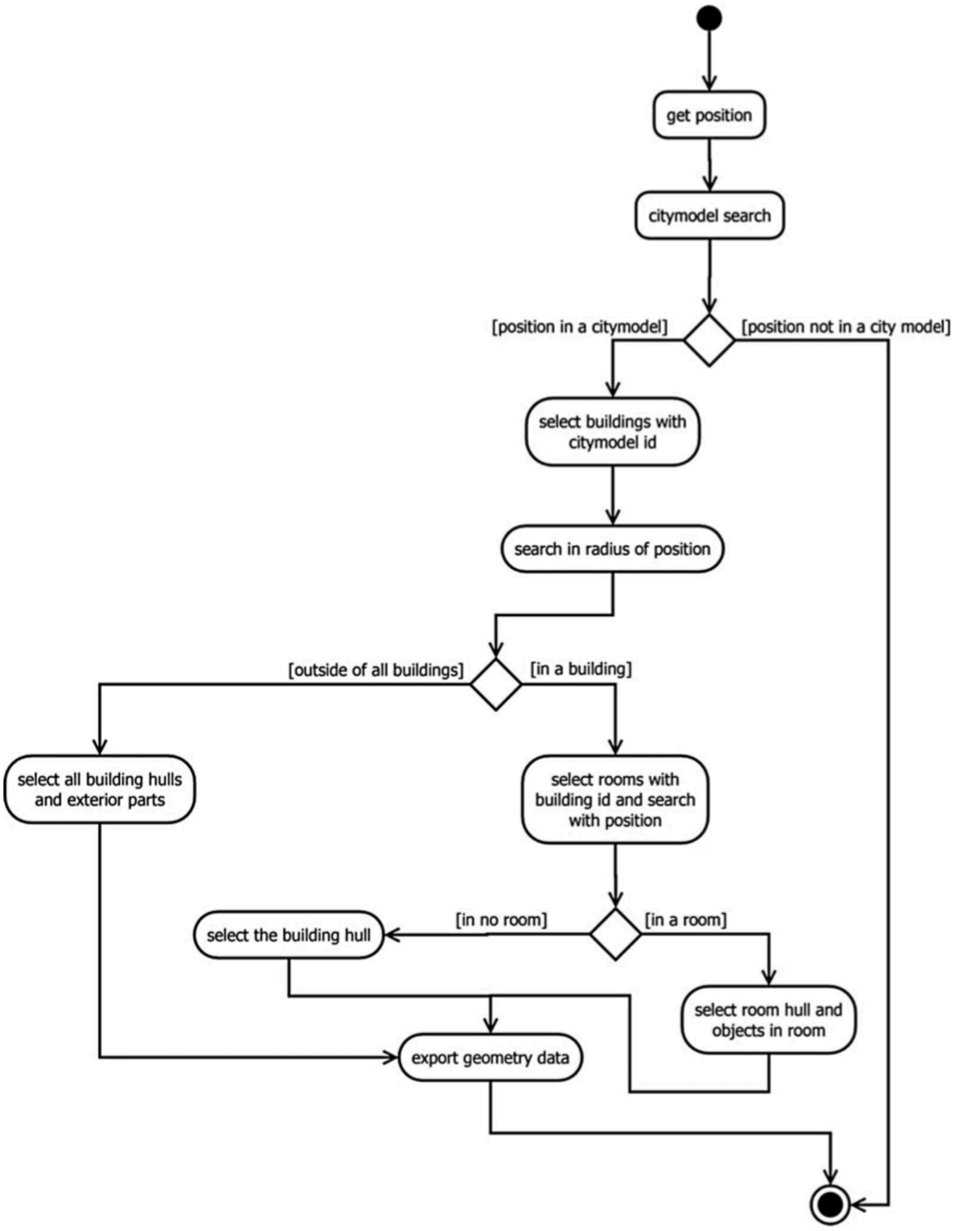

get position
citymodel search
[position in a citymodel]
[position not in a city model]
select buildings with citymodel id
search in radius of position
[outside of all buildings]
[in a building]
select all building hulls and exterior parts
select rooms with building id and search with position
[in no room]
[in a room]
select the building hull
select room hull and objects in room
export geometry data

Figure 2.11: The selection process for an AR application [Blut et al., 2019]

3 Implementation analysis of CityGML in 3DCityDB

The CityGML data model can be implemented differently. The mapping with 3DCityDB is analysed to understand the mappings and the corresponding reasons to simplify certain parts of the CityGML data model. This section describes the mapping of the CityGML data model to relational databases with 3DCityDB. 3DCityDB supports the commercial software Oracle and the open source software PostgreSQL. The implementation in PostgreSQL is used to analyse the mapping. The tables according to the scope of this research and the theoretical background of 3DCityDB are therefore used. The tables can be seen in figure 3.1 and more detailed in appendix C.

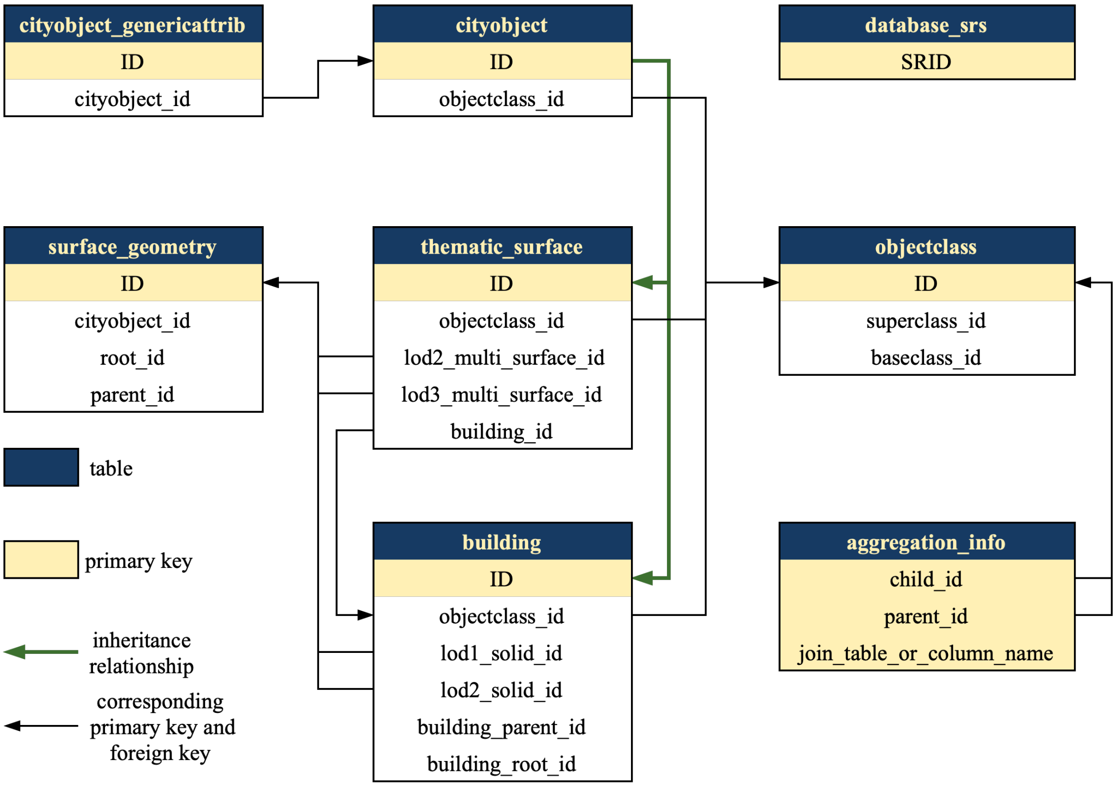


Figure 3.1: Overview of the simplified CityGML data model mapping to the relational database PostgreSQL with 3DCityDB.

Every CityGML class is mapped to another table. The `objectclass` table registers all CityGML class names with the corresponding table name. The table also contain the inheritance relationships between the classes. The `aggregation_info` table describes on the other hand the aggregate and composite relationships. The columns `child_id` and `parent_id` refer to the classes of the `objectclass` table. The column name to join them on is specified in the column `join_table_or_column_name` in order to establish the relationship. Additional information about the relationship can be stored in the columns `is_composite`, `min_occurs` and `max_occurs`. The columns `is_composite` distinguishes if the relationship is an aggregate or a composite. The multiplicity of the relationships can be specified with the column `min_occurs` and `max_occurs`. In case of a 0..* relationship (UML notation), `min_occurs` is zero and `max_occurs` is null.

The city objects are mapped to the `cityobject` table. Every city object contains a bounding box. The bounding box is stored as a `PolygonZ` geometry in the `envelope` column. There are two other columns that represent the ID of the city object, because it is otherwise impossible to guarantee that the GML ID is unique over multiple city models. There are therefore two columns: the gml_id and the id. The column id uses therefore a sequence and is therefore used as primary key.

The mapping contains an inheritance relationship. The inheritance relationship reuses the primary key of the base-class in the sub-classes. The sub-classes of the cityobject table are here the building table and the thematic_surface table. This means that the cityobject table contains the same primary keys as the building and thematic_surface tables.

The building table merges three CityGML classes (AbstractBuilding, Building and BuildingPart). The CityGML class can be identified with the objectclass_id. The objects of the building table can belong to an aggregate of objects. The aggregate relationship is represented with the attribute columns building_parent_id and building_root_id. These IDs are used to create an aggregate tree structure. The building_parent_id refers to the predecessor and the building_root_id refers to the top level of the tree. The building table also contains attribute columns and geometry columns. The attribute columns correspond to the ones that belong to the CityGML building class. The geometry columns are columns such as lod1_multi_surface_id, lod2_multi_surface_id, lod1_solid_id and lod2_solid_id. These IDs refer to a surface geometry in the surface_geometry table.

The thematic_surface table contains thematic boundary surfaces, which are called semantic surfaces in CityJSON. The column objectclass_id refers to the name of the thematic surface:

- 30 (CeilingSurface)
- 31 (InteriorWallSurface)
- 32 (FloorSurface)
- 33 (RoofSurface)
- 34 (WallSurface)
- 35 (GroundSurface)
- 36 (ClosureSurface)
- 60 (OuterCeilingSurface)
- 61 (OuterFloorSurface)

The thematic_surface table also contains the columns lod2_multi_surface_id, lod3_multi_surface_id or lod4_multi_surface_id. They refer to the surface_geometry table. A thematic surface can also be related to a building with the column building_id. A thematic surface can beside a building also reference to a room or a building installation.

The geometry is stored in the surface_geometry table. The geometry consists of surfaces. The geometries are represented with a parent/child structure. The parent/child structure that is used could be seen in figure 3.2. It specifies to which solid, shell or surface the object belongs. The hierarchy requires a unique id column. A sequence ID is used to guarantee a unique ID. The root_id is added to avoid recursive queries. The objects that represent a surface contain a PolygonZ geometry in the geometry column. A PolygonZ geometry can include holes. The other objects only contribute to the parent/child structure.

The column is_solid in figure 3.2 identifies if it is a solid. The column is_composite identifies if it is a composite, such as a CompositeSurface, or an aggregate, such as a MultiSurface. The geometry types MultiSurface and MultiSolid are implemented the same, but they can be distinguished based on their children. While the child of a MultiSolid is a Solid, the child of a MultiSurface is a surface.

CityGML allows to share geometries to avoid redundancy with the XLink concept as explained in section 2.1.1. It is not possible to share a geometry in the surface_geometry table due to the parent/child structure. This means that it is possible to share geometries with CityGML, but not in PostgreSQL with 3DCityDB.

Furthermore, the column solid_geometry is used to perform 3D spatial operations. It stores the outer shell of a volume with the geometry type PolyhydralsurfaceZ.

The cityobject_genericattrib table is used to implement the concept of generic attributes as described in section 2.1.1. An attribute contains a name and a value. The name is mapped to the column attrname. The value

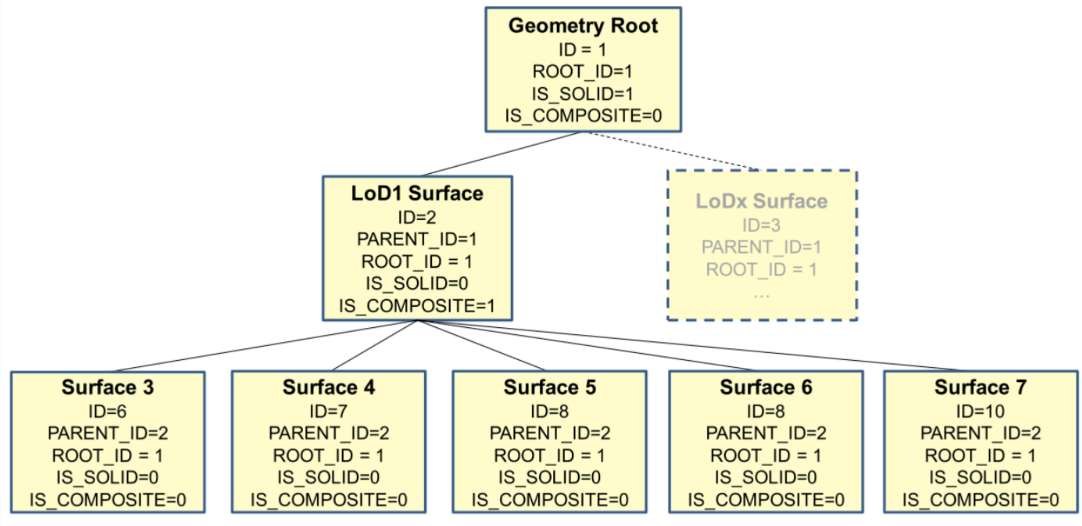


Figure 3.2: Hierarchy to represent a solid geometry [Kolbe, 2019]

is more difficult to map. The value has to be mapped to a specific column due to the varying data type of the value. The `datatype` column contains therefore an integer. Each integer refers to another data type and each data type has its own column. This means that the `datatype` column refers to the column in which the value is stored [Kolbe, 2019].

The reference system is mapped to the `database_srs` table using the `srid` and the `gml_srs_name` attribute columns, which are defined during the database setup [Kolbe, 2019].

4 Methodology

CityJSON is mapped to an entity-relationship model. This model is mapped to a document and relational schema. CityGML is directly mapped with 3DCityDB. After that, the database schemas are converted to GraphQL object types. The queries are then described and defined as GraphQL query types. The GraphQL object types and GraphQL query types form together the GraphQL schema. Queries can then be sent to the GraphQL endpoint to perform the experiments.

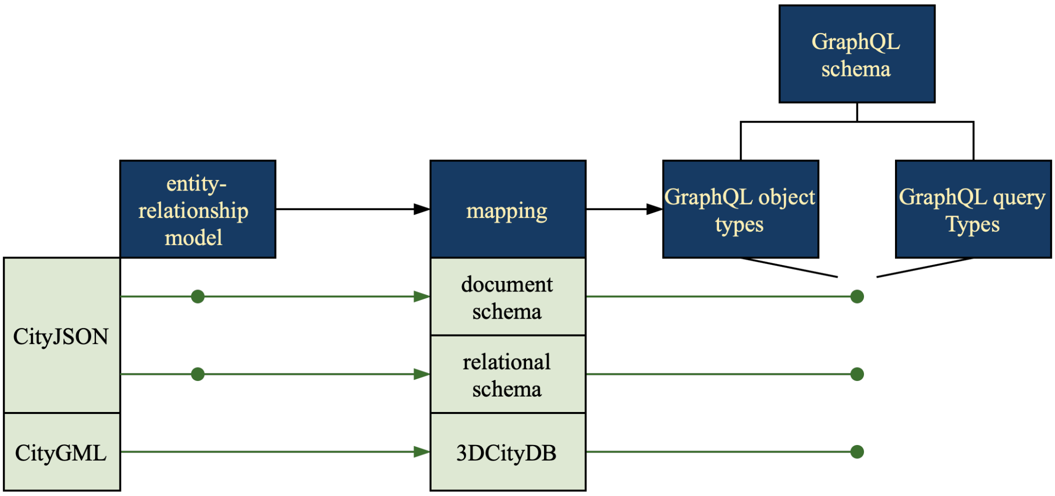


Figure 4.1: Overview of the methodology

4.1 Data model mapping

The objects of CityJSON and the relationships between them are described in section 2.1.3. This section proposes methods to map them to relational and document databases. A distinction between them is made, because they use different mapping rules as described in section 2.2.2 and 2.2.3.

The mapping of CityGML with 3DCityDB is already analysed in section 3. The mapping of CityJSON to PostgreSQL is compared to the mapping of CityGML to PostgreSQL with 3DCityDB, because they implement the CityGML data model differently and they are therefore mapped differently to the relational database PostgreSQL.

4.1.1 Entity-relationship analysis and challenges

The objects of CityJSON and the relationships between them of section 2.1.3 can be mapped to an entity-relational model. The CityJSON objects of figure 2.3 and the nested objects in figure 2.4, 2.5, 2.6 and 2.7 are first converted to entities.

However, this results in many entities. Some of them contain only one attribute-value pair. The property `type`, for example {“type”:“CityJSON” }, and `version`, for example {“version”:“1.0” }, are properties with one attribute-value pair. They are therefore added to the `metadata` entity, because they can be considered metadata. The `transform` entity, for example {”transform”: {”scale”: [0.001, 0.001, 0.001], ”translate”: [78248.66, 457604.591, 2.463]}}, also contains only two attribute-value pairs, but it operates on the vertices and it is therefore no `metadata`. This results in the following entities:

- `metadata` entity including the `type` and `version`, which are the properties `metadata`, `type` and `version` of the CityJSON object in figure 2.3

- `city object` entity, which is the object of figure 2.4
- `attributes` entity, which is the property `attributes` of the `city object` in figure 2.4
- `geometric` entity, which is the object of figure 2.5.
- `semantics` entity, which is the object of figure 2.6
- `semantic surface` entity, which is the object of figure 2.7
- `vertices` entity, which is the property `vertices` of the CityJSON object in figure 2.3
- `transform` entity, which is the property `transform` of the CityJSON object in figure 2.3

The relationships between the objects of section 2.1.3 are translated in an entity-relationship model as could be seen in figure 4.2. There is one relationship between the `city object` entity and the `metadata` entity added, because the `metadata` entity is associated with the city objects of the CityJSON file. This is especially important for databases that contain multiple 3D city models.

There is also a division made between dependent and independent entities. The difference between them is explained in section 2.2.1. The properties of the CityJSON object in figure 2.3 are independent, because they are part of the first level of the CityJSON file. The `attributes` entity, the `geometric` entity, the `semantics` entity and the `semantic surface` entity are part of a city object. These entities can not exist without the `city object` entity and they are therefore dependent entities. A few relationships are not modelled yet. They are marked in figure 4.2 as challenges. The decisions for these challenges are explained in section 4.1.2 and in section 4.1.3.

The first challenge is the mapping of the geometry, which influences the `geometric` entity, the `vertices` entity and the `transform` entity as could be seen in figure 4.2. The boundaries in figure 2.5 do not contain their real coordinates yet, but indexes. A referenced structure with the `vertices` and the `transform` object is needed to obtain the real coordinates. The advantages of the `vertices` and the `transform` object are described in section 2.1.3.

Another option would be to resolve the indexes and to transform the integer coordinates. The advantage would be that no referenced structure is needed anymore and that the real coordinates could be used to map the boundaries of figure 2.5 to a geometry type that is supported by the database. This is an advantage, because databases support spatial operations on certain geometry types. Therefore, these geometry types might simplify spatial operations and support spatial indexes.

The second challenge is the mapping of the semantics, which influences the `geometric` entity, the `semantics` entity and the `semantic surface` entity as could be seen in figure 4.2. The property `boundaries` of figure 2.5 exists of individual surfaces. A surface is related to a semantic surface object of figure 2.7 through the semantics object of figure 2.6. The first option is that the three entities have to be referenced to each other inside or outside the database. The other option is that the `boundaries` have to be mapped as individual surfaces and these surfaces relate to semantic surface objects.

4.1.2 Relational schema

The entity-relationship model of figure 4.2 is mapped to a relational database using the relational database rules of section 2.2.2. First are the dependent and independent entities mapped to separate tables. After that the challenges are mapped, then the relationships and lastly the IDs.

Challenges

The first challenge is the mapping of the geometry, which influences the `geometric` entity, the `transform` entity and the `vertices` entity as explained in section 4.1.1. The `geometric` entity must be mapped to a separate `geometry` table due to the many-to-one relationship between the `geometric` entity and the `city object` entity. It is called the `geometry` table instead of the `geometric` table, because the city object of figure 2.4 also contains the property ğeometry. This property contains an array of geometric objects as can be seen in figure 2.4

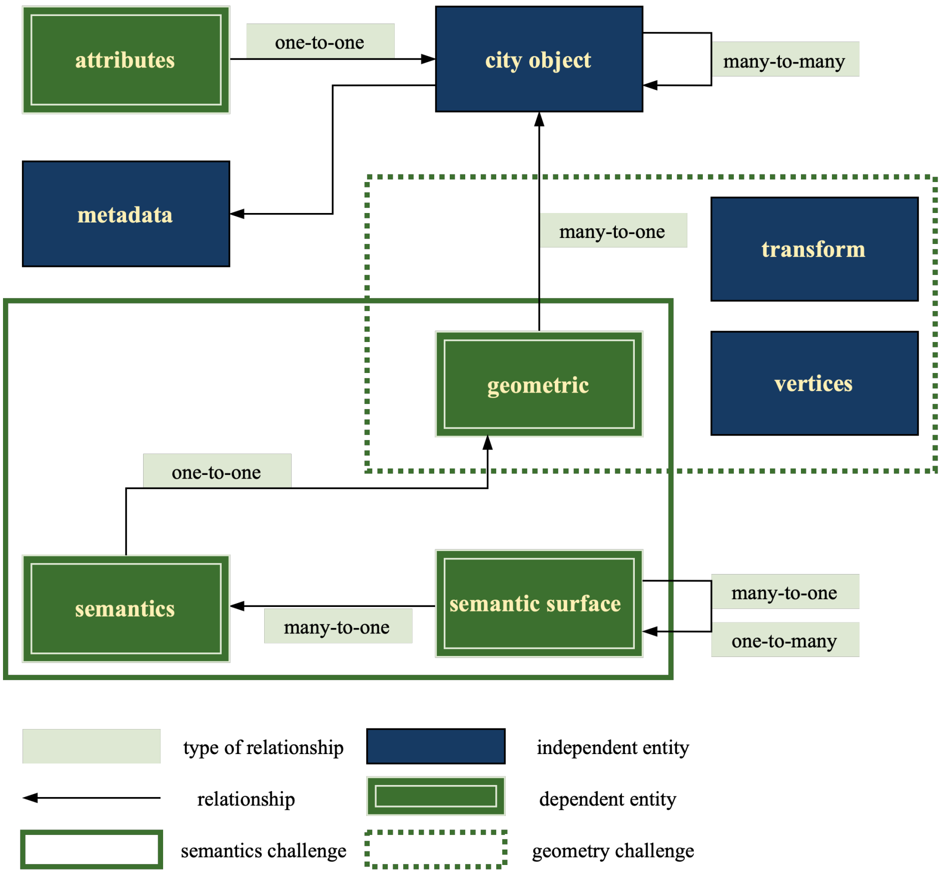


Figure 4.2: The defined entity-relationship model

The indexes are resolved and the integer coordinates are transformed to obtain the real coordinates. The `vertices` entity is therefore not needed anymore, but the `transform` entity has to be mapped in order to query and reconstruct a new file that is approximately the same as the original. The real coordinates are used to create a supported geometry type. The relational database PostgreSQL is investigated for this, but it probably applies to other relational databases as well. The supported geometry types of the relational database PostgreSQL are an advantage, because the database supports many reference systems and 3D operations on 3D geometry types. However, not all geometry types of the CityGML data model are supported. PostgreSQL does not support `voids`, `MultiSolids` and `MultiComposites`. In order to represent all geometry types of the CityGML data model, another mapping has been established. This mapping does not apply to the geometry types `MultiPoint` and `LineString`, but this is not an issue since they are already supported by PostgreSQL. They can be stored as an extra geometry.

There are also other options to establish the mapping. 3DCityDB uses for instance a parent/child structure as explained in section 3 instead. Here, the geometry is represented as individual surfaces. This is not only done to create the geometry types that are not supported, but also because each individual surface can in this way refer to a semantic surface object. The difficulty is the reconstruction of the whole geometry. 3DCityDB uses a parent/child structure for this, but this structure needs a recursive function to understand the position of the surface in the hierarchy. Therefore, another mapping to understand the position in the hierarchy is established. The geometries are represented as an hierarchy of surfaces as could be seen in the figure 4.4 and table 4.1. The surfaces contain the data type `PolygonZ`. A `PolygonZ` geometry is allowed to contain holes. The columns `solid_num`, `shell_num` and `surface_num` represent the position of the surface in the hierarchy. The surfaces are mapped to a separate `surfaces` table. They also have to contain a reference to the `geometry` table, because of the many-to-one relationship between them. It is unknown how many surfaces a geometry is going to contain.

The surfaces also reference to the `semantic_surface` table. The `semantic surface` entity is mapped to a separate `semantic_surface` table, because one semantic surface can refer to multiple surfaces and to other semantic

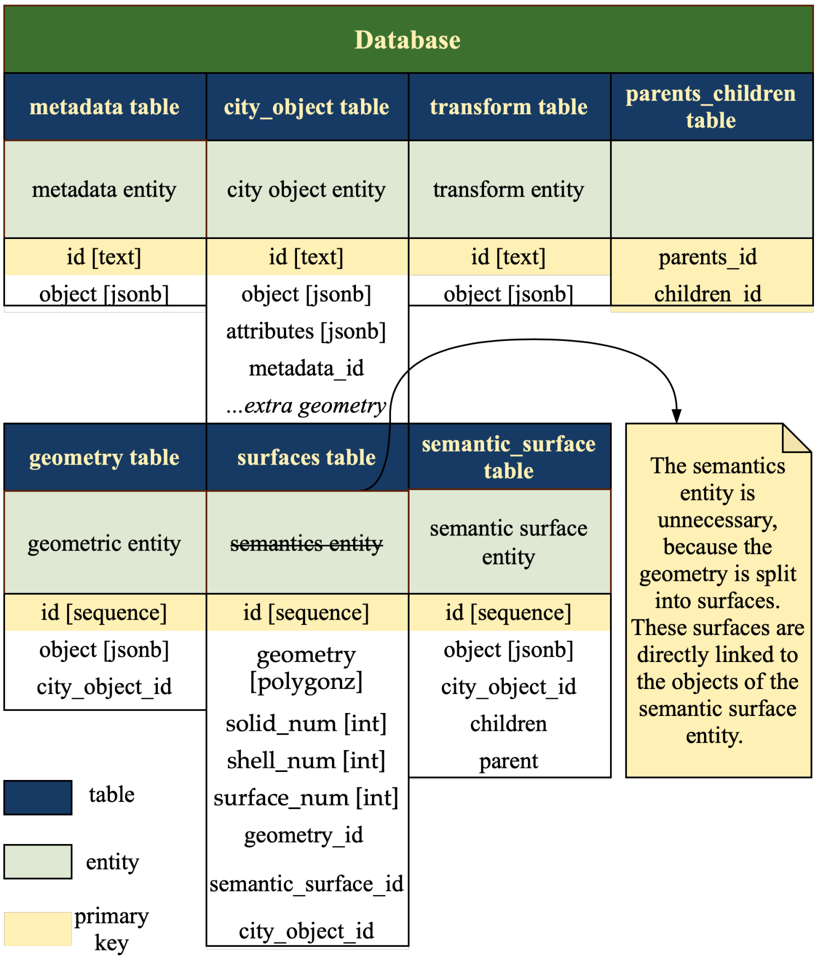


Figure 4.3: An overview of the relational database structure based on the entity-relationship model of figure 4.2.

surface objects. The `semantics` entity is on the other hand not needed anymore, because the surfaces are referenced to the semantic surfaces directly. Extra geometries can be added to the `city_object` or the `geometry` table to support `MultiPoints`, `Linestrings`, `MultiPolygons`, `PolyhedralSurfaces`, `ConvexHulls`, `geographicalExtents` or `3D boxes` depending on the use case.

Relationships

The mapping of the relationship between the `metadata` entity and the `city object` entity in figure 4.2 is established with a foreign key. The column `metadata_id` refers to the primary key of the `metadata` table.

The `attributes` entity is even as the `geometric` entity a dependent entity of the `city object` entity. Although the entity can be mapped to a separate table, this would require a join between the table and the `city_object` table. The `attributes` entity is therefore mapped to a JSONB column in the `city_object` table instead. A JSONB column is used to store all attributes in one column and to avoid a join. JSONB is used instead of JSON, because of the reasons described in section 2.2.2.

The other properties of the `city object` entity are mapped together in a JSONB column called `objects`. Different types of city object may have different properties. A JSONB column can contain different properties. The usage of the JSONB column makes it therefore possible to map all types of `city objects` to the same `city_object` table. The same principle is used for the properties of the `metadata` entity in the `metadata` table, the `transform` entity in the `transform` table, the `geometric` entity in the `geometry` table and the `semantic surface` entity in the `semantic_surface` table.

```
[
    [ //-- 1st Solid
      [ //-- 1st Shell
      [[0, 3, 2, 1, 22]], [[4, 5, 6, 7]], [[0, 1, 5, 4]], [[1, 2, 6, 5]]
      ],
      [ //-- 2nd Shell (void)
      [[240, 243, 124]], [[244, 246, 724]], [[34, 414, 45]], [[111, 246, 5]]
      ]
    ],
    [ //-- 2nd Solid
      [ //-- 1st Shell
      [[666, 667, 668]], [[74, 75, 76]], [[880, 881, 885]], [[111, 122, 226]]
      ]
    ]
]
```

Figure 4.4: An example of the property boundaries in case of a `MultiSolid`

id	geometry	solid_num	shell_num_void	surface_num
1	polygonz (0 3 2 1 22)	0	0	0
2	polygonz (4 5 6 7)	0	0	1
3	polygonz (0 1 5 4)	0	0	2
4	polygonz (1 2 6 5)	0	0	3
5	polygonz (240 243 124)	0	1	0
6	polygonz (244 246 724)	0	1	1
7	polygonz (34 414 45)	0	1	2
8	polygonz (111 246 5)	0	1	3
9	polygonz (666 667 668)	1	0	0
10	polygonz (74 75 76)	1	0	1
11	polygonz (880 881 885)	1	0	2
12	polygonz (111 122 226)	1	0	3

Table 4.1: An example of the surfaces in the `surfaces` table before resolving the indexes and transforming the integers

The many-to-many relationship of the `city object` entity requires an additional table as described in section 2.2.2. The additional table is the `parents_children` table. Additionally, an extra relationship is added between the `semantic_surface` table and the `geometry` table. This is done in order to link the `semantic surface` entities directly to a `geometric` entity.

IDs

Each table needs a primary key. The tables use the primary keys of figure 4.5.

```
metadata table = metadata_ + file name
city_object table = city object ID
transform table= transform_ + file name
geometry table = sequential
surfaces = sequential
semantic_surface table = sequential
```

Figure 4.5: The primary keys of the relational database tables

4.1.3 Document schema

The entity-relationship model of figure 4.2 is mapped to a document database using the document database rules of section 2.2.3. First are the dependent entities theoretically mapped to a separate collection and the independent entities as nested documents inside the `city_object` collection. After that, the challenges are mapped, then the relationships and lastly the IDs.

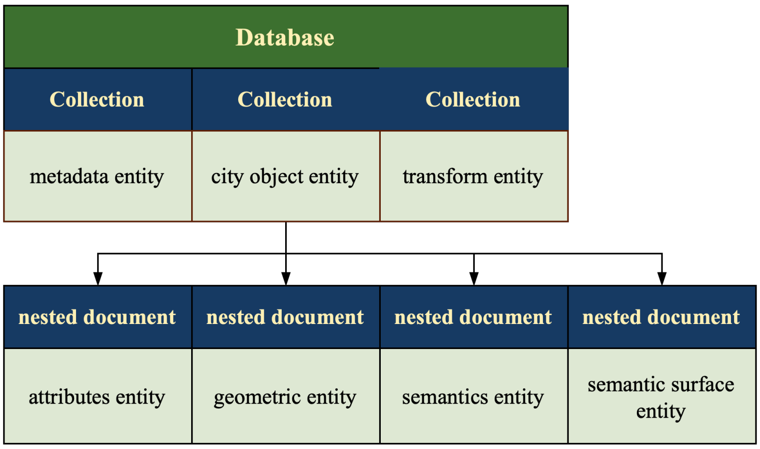


Figure 4.6: An overview of the document database structure based on the entity-relationship model.

Challenges

The first challenge is the mapping of the geometry, which influences the `geometric` entity, the `vertices` entity and the `transform` entity. The entity `vertices` is not mapped to a collection, because the vertices list exceeds the maximum BSON document size. Another option is to query each vertex separately, but this requires many references, which is not preferred in a document database. The indexes are therefore resolved during the mapping.

They are also transformed during the mapping in order to obtain the real coordinates directly, but this is not necessary. It is not necessary, because the boundaries of the `geometric` entity are stored as they are. They can not be stored as GeoJSON objects, which is the only geometry type that the document database MongoDB supports. This probably applies to other document databases as well. It is problematic to map the geometries of the CityGML data model to GeoJSON objects, because it is only possible to do spatial operations on GeoJSON objects when the coordinates are related to the WGS84 reference system. The coordinates in a CityJSON file can be related to other reference systems than WGS84. Another reason is that not all geometry types of a CityJSON file are supported by GeoJSON. For instance, the geometry type `MultiSurface` of the CityGML data model is not supported by GeoJSON. The `MultiSurface` must be converted to a `MultiPolygon` to become a GeoJSON Object. GeoJSON is originally in 2D instead of 3D and no spatial operations in 3D are supported. Although MongoDB is investigated, we expect that these reasons also apply to other document databases.

The second challenge is the mapping of the semantics, which influences the `geometric` entity, the `semantics` entity and the `semantic surface` entity. They are stored as they are, because the boundaries are also stored as they are. The semantic surface objects are not nested inside the property `values` of figure 2.6, because semantic surface objects can be referenced to each other.

Relationships

The dependent entities that belong to a city object are mapped as nested documents. The parent and child city objects are not nested, because of the many-to-many relationship. One city object can have multiple children and multiple parents. This means that the same city object can be present in multiple documents. It is therefore not preferred to nest them, because this would lead to implications concerning the maintenance. They are therefore referenced to each other with their IDs.

One database could contain multiple 3D city models and therefore every city object has to contain a reference to the corresponding metadata document. The relationship between the `metadata` entity and the `city object` entity is also established with references, because duplicating the metadata for every city object would lead to implications concerning the maintenance. Additionally, the transform document also contains a reference to the corresponding metadata document.

IDs

Each document of each collection needs a primary key. The collections use the primary keys of figure 4.7.

```
metadata object = metadata_ + file name
city object = city object ID
transform object = transform_ + file name
```

Figure 4.7: The primary keys of the document database tables

4.1.4 Validation and comparison

The validation is used to investigate if the objects of CityJSON and the relationships between them are correctly stored in the database. After the CityJSON file is stored in the database, the database can be queried in order to reconstruct a new CityJSON file. The original and new CityJSON file make it possible to compare them as could be seen in figure 4.8. They are compared based on their keys and values. It is investigate if the values of the keys are the same and if the same keys are present. It is also investigated if the file sizes are the same and additional software is used to validate CityJSON files.

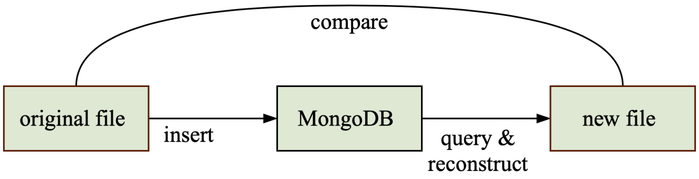


Figure 4.8: Validation method

A descriptive comparison between the mappings of section 3, 4.1.2 and 4.1.3 is done to investigate the differences between the general mapping, the mapping of the challenges, the mapping of the relationships, the mapping of the IDs and the mapping of multiple city models in section 4.1.5.

4.1.5 Mapping of multiple city models

The selection process of figure 2.11 selects the relevant city model. This process requires a mapping that includes multiple city models. Multiple city models can have different reference systems.

Relational databases do not map them to different databases, because cross-database references are not directly supported. They can map them to one database with different database schemas. Each schema defines the reference system of the geometries on forehand. 3DCityDB of section 3 does this, but GraphQL has to access the different schemas in this way.

This was not possible in this research as described in section 5.4.4 and therefore only one database schema in one database is used for the relational database schema of section 4.1.2. This means that the database schema does not define the reference system of the geometries on forehand.

Document databases map multiple city models to the same database in the same collection. No geometry types are used and therefore no reference systems are defined.

4.2 Use case

The use case enables location-based applications on mobile devices and it is used to detect whether relational and document databases are suitable for the use case. The databases are used to store the 3D city models as described in section 4.1. The GraphQL API layer defines the interactions between the application and the database as could be seen in figure 4.9. The selection process of figure 2.11 is used to define the queries. Although the selection process is used in combination with a XML pull parser and a local database, the query algorithm can also be used for a semi client-server architecture as described in section 2.3.3. A semi client-server architecture is the combination of a client-server architecture and a local database. This research only uses a client-server architecture, because no local databases are tested, and it uses CityJSON files as 3D city models.

The selection process pre-selects the buildings within a certain radius, but this might not be logical to do without a local database. The pre-selected buildings would have to be provided as argument again when using a client-server architecture. A for- loop over the building IDs would be required to obtain the building objects and the requested IDs would have to be transferred over the network. These queried are used for testing, but they might not be efficient without a local database.

The selection process also returns the interior parts, but the usage of CityJSON files as 3D city models affects this. The interior parts can in the selection process be returned as a room, but CityJSON does not support LoD 4 features such as a room as described in section 2.1.3. The only option is therefore to return one building. Additionally, one city object can contain geometries with multiple LoDs. However, only one geometry is needed to visualize a city object and the amount of transferred data has to be limited. It is therefore convenient to return the geometry with the highest LoD as default, because each city object can contain different LoDs. The highest LoD might optimize the user's experience for visualizing objects nearby. The highest LoD is not always the best choice, because of performance issues such as the processing time and errors related to the complexity of a higher LoD.

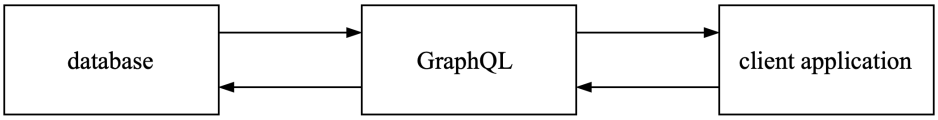


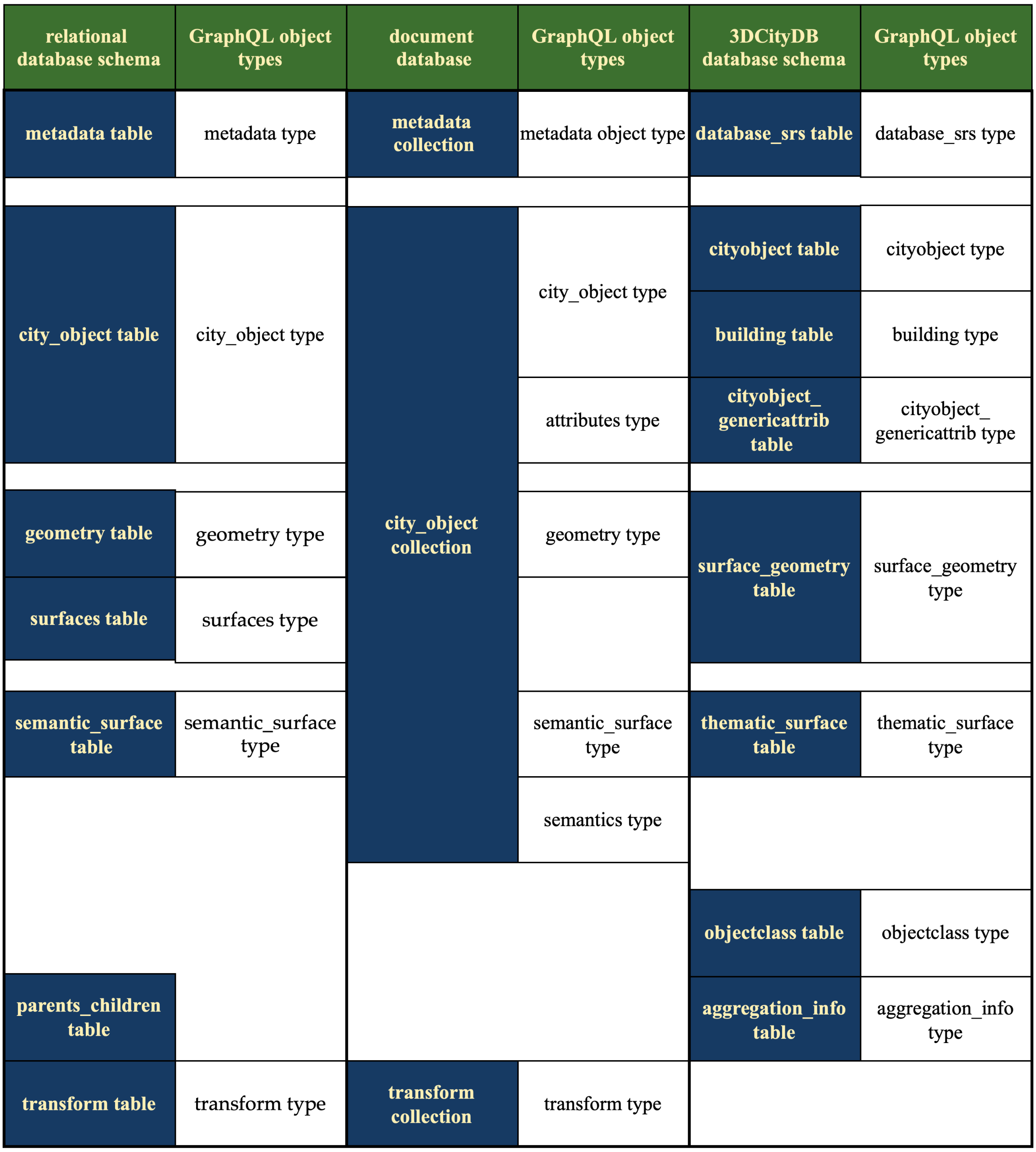
Figure 4.9: The architecture of the use case

4.2.1 GraphQL as API layer

Section 2.3.2 describes the specification of GraphQL. GraphQL has to be connected to the application and the databases. The defined database schemas in section 4.1 are mapped to GraphQL object types as could be seen in figure 4.11. GraphQL object types map to nested documents and collections for document database schemas and to tables for relational database schemas.

```
MultiSurface:
boundaries = ListField(ListField(ListField(IntField())))
MultiSolid:
boundaries = ListField(ListField(ListField(ListField(ListField(IntField())))))
```

Figure 4.10: An example of the `boundaries` field that can contain varying data types, because the data type of the property `boundaries` varies based on the geometry data



relational database schema	GraphQL object types	document database	GraphQL object types	3DCityDB database schema	GraphQL object types
metadata table	metadata type	metadata collection	metadata object type	database_srs table	database_srs type
city_object table	city_object type	city_object collection	city_object type	cityobject table	cityobject type
				building table	building type
			attributes type	cityobject_ genericattrib table	cityobject_ genericattrib type
geometry table	geometry type		geometry type	surface_geometry table	surface_geometry type
surfaces table	surfaces type				
semantic_surface table	semantic_surface type		semantic_surface type	thematic_surface table	thematic_surface type
			semantics type		
				objectclass table	objectclass type
parents_children table				aggregation_info table	aggregation_info type
transform table	transform type	transform collection	transform type		

Figure 4.11: The database collections and tables mapped to GraphQL object types

An object type has fields. A field has a name and a data type. The data type of these fields must be specified exactly, which is an issue for fields with varying data types. The data types of the properties `boundaries` and `values` vary based on the geometry type. While `MultiSurfaces` have an hierarchy of three arrays, `MultiSolids` have an hierarchy of five arrays as could be seen in figure 4.10. Document databases allow fields with varying data types even as JSONB columns in relational databases. It is difficult to implement varying data types with GraphQL. Solutions can be to parse a string or to support only one geometry type.

The GraphQL queries and the GraphQL object types are together the GraphQL schema. The queries of section 4.2.2 are defined as field in GraphQL object types. These fields have to be resolved with resolver functions as described in section 2.3.2.

4.2.2 Queries

Number	Name	Argument	Operations	Filter	Returned data
1	**location**	position with the field lat, long and alt			location with the fields latitude, longitude and altitude
2	**citymodel**	position with the field lat, long and alt	intersection		city model ID
3	**radius100**	position with the field lat, long and alt	within 100 meter	city object type = Building	city object IDs
4	**inside**	position with the field lat, long and alt	intersection	city object type = Building	city object ID
5	**maxlod**	city object ID			ID of the geometry with the highest LoD
6a	**cityobjects**	city object ID		ID-based	ID of the city object
6a	**cityobjects**	city object ID		ID-based	ID and attributes of the city object
6a	**cityobjects**	city object ID		ID-based	ID and geometry of the city object
7	**surfaces**	surface ID or other		ID-based or other	the surface with the related semantic surface object

Table 4.2: The options to perform query 2, 3 and 4

The selection process of figure 2.11 is split into seven queries. These queries are used to detect whether the relational and document databases are suitable for the queries of the use case. The mobile device provides the user's position in the 3D version of the WGS84 reference system (latitude: degrees, longitude: degrees, altitude: meters). This is EPSG:4979. The altitude is the height above the ellipsoid.

The first query as could be seen in figure 4.2 investigates whether the user's position is provided and returned correctly. The mobile device provides the user's position in EPSG:4979 as argument and the resolver function returns the user's position in EPSG:4979. EPSG:4979 uses geographical coordinates, while most reference systems for 3D city models use geometrical coordinates.

The second, third and fourth query require spatial operations. These spatial operations can be geographical and geometrical. These queries investigate the following requirements:

- Multiple city models with different reference system can be queried through GraphQL.
- The database supports spatial reference transformations.
- The database supports geometrical and geographical spatial operations such as intersection and within and indexes on them.
- The database supports the geometry definitions of the CityGML data model.
- The city objects can be filtered on the type of city object.

There are many options to implement the queries. The geometry of the city model or city object can be represented in multiple ways as can be seen in table 4.3. The necessary steps can be performed before the the data is inserted in the database, during the query in the database or during the query outside the database. It is preferred to perform them in the database, because databases can use spatial indexes and the data does not have to be altered. The implementation of query 2, 3 and 4 differ based on the requirements of the relational and document databases.

The second query returns the ID of the relevant city model. An intersection is used with a geometric representation of the city model as input.

Options	Advantages	Disadvantages	Necessary steps
The advantages, disadvantages and necessary steps are based on a city model with geometrical coordinates and the user's position with geographical coordinates.			
geographical coordinates	It requires less queries to the database than geometrical coordinates.	It might require more transformations than geometrical coordinates.	The geometries of the city model or city objects have to be transformed to the geographical coordinates of EPSG:4979.
geometrical coordinates	It requires less transformation than geographical coordinates.	It requires more queries to the database than geographical coordinates.	The user's position has to be transformed to the reference system of the 3D city model.
envelope (2D)	It is probably faster than the usage of a convex hull.	It does not represent the surface of the city model or city object correctly.	The geometry of a city object or the geographical extent of a city model can be to create an envelope.
convex hull (2D)	It represents the surface of a city model or city object better than an envelope.	It is probably slower than the usage of an envelope.	The geometry of a city object can be used to create a convex hull.
3D box (3D)	It is probably faster than the usage of the original geometry.	It represent the city model or city object incorrectly.	The geometry of a city object or the geographical extent of a city model can be used to create a 3D box.
original geometry (3D)	It represent the city model or city object correctly.	It is probably slower than the usage of the original geometry.	The geometry of the city object is used.

Table 4.3: The options to perform query 2, 3 and 4

The third query returns the IDs of the buildings around the user's position. The IDs within a 100 meter radius are returned. The unit type of the radius is set according to the measurement units of the reference system. In case geographical coordinates are used, the 100 meters are converted to degrees. The spatial operation `within` is used with the geometric representation of the city objects as input. Additionally, the city objects are filtered on the city object type building.

The fourth query is used to decide whether user is inside or outside a building. It uses the geometric representation of the city object as input to perform the spatial operation intersects. The query also filters on buildings and returns the ID of the city object.

The fifth query investigates whether it is possible to access and operate on the LoD attribute. The city object in which the user is located is provided as argument. The resolver function returns the ID of the geometry with the highest LoD.

The sixth and seventh query investigate whether it is possible to retrieve the attributes, the geometries and semantic data. The sixth query is split in three parts: one query only returns the ID, another only the ID and the attributes and another the ID and the geometry. These queries show the field filtering possibilities of GraphQL and the way in which the geometries are returned. The ID of the city object is provided as argument. The seventh query returns a surface, which is related to a semantic surface object. The provided argument might be dependent on the database schema.

4.3 Preliminary analysis

Experiments investigate the performance of the databases using GraphQL as API layer. Every part of the environment influences the performance. The environment exists of the following parts:

- The device including its memory and storage space

- The databases with the used data
- The connection(s) from the database to GraphQL
- GraphQL
- The connection(s) to the client(s)

The network performance depends on the amount of transferred data and the number of requests. These components optimize the network performance independently of the amount of users, the available network bandwidth and the browser. The amount of transferred data and the number of requests optimize together the usage of the network bandwidth. The amount of transferred data also optimizes the amount of data that is potentially stored on the mobile device.

The performance is divided in the network performance and the performance between the database and GraphQL. GraphQL reduces the amount of transferred data to the client, but it does not necessarily reduce the amount of transferred data from the database as could be seen in figure 4.12. The performance is also measured in terms of retrieval times.

The retrieval times indicate whether the queries are executed efficiently. They can be divided in the execution time in the database and the execution time in the API layer. The connections between the database, GraphQL and the client also influence the retrieval times, but those are neglected in this thesis.

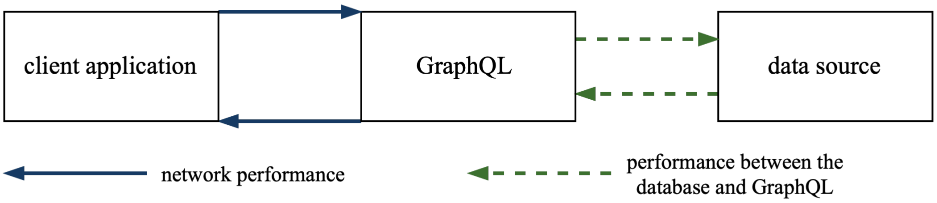


Figure 4.12: Division of the performance experiments

4.3.1 Network performance

The network performance depends on the amount of transferred data and the number of requests. This research mainly focuses on the amount of transferred data, because the requests are constructed in this research and the number of needed requests per query is therefore likely to be one.

GraphQL is designed to reduce the amount of transferred data to the client, but it does not necessarily reduce the amount of transferred data from the client or in other words the request sizes. The total amount of transferred data is dependent on request sizes from the client and the response sizes to the client. The total of the HTTP request size and the HTTP response size are calculated, because larger HTTP request sizes might nullify the reduction of the HTTP response sizes.

4.3.2 Performance between the database and GraphQL

The amount of transferred data from the database is dependent on the queries that are sent to the database. These queries are for instance SQL queries for relational databases and MQL queries for document databases. These SQL/MQL queries can be identified in logfiles for example. The response sizes of these SQL/MQL queries can be used to identify over-fetching from the database. Over-fetching means that more than the required data is returned. One of the causes might be that the resolver functions are not designed properly. The response sizes from the database are therefore compared to the HTTP response sizes from GraphQL to the client.

4.3.3 Retrieval times

The execution times in the databases can be identified with the execution times of the SQL/MQL queries. The execution times of these queries dependent on the amount of data that is stored in the database, the used operations and the used indexes.

The retrieval times of the API are not separated from the connections in this thesis, but the retrieval times dependent on the amount of data that has to be processed in the resolver functions and the efficiency of the resolver functions in general.

5 Implementation

5.1 Data

5.1.1 Den Haag

The dataset is retrieved from https://3d.bk.tudelft.nl/opendata/cityjson/1.0/DenHaag_01.json. The size is 2.6 MB. The dataset contains a transform object and the reference system of the dataset is EPSG:7415. The dataset contains parent and child city objects. Every city object contains one geometric object of LoD 2. The geometry types are CompositeSurface and Solid. The geometric objects contain the property semantics. The semantic surface objects do not have parents and children. The city objects are of type Building, BuildingPart and TINRelief. The metadata contains the geographicalExtent. The corresponding CityGML file is 21.2 MB.

5.1.2 Delfshaven

The dataset is retrieved from https://3d.bk.tudelft.nl/opendata/cityjson/1.0/3-20-DELFSHAVEN.json. The size is 1.4 MB. The dataset contains a transform object and the reference system of the dataset is EPSG:28992. Every city object contains one geometric object of LoD 2. The geometry types are MultiSurface. The geometric objects contain the property semantics. The semantic surface objects do not have parents and children. The city objects are of type Building. The metadata contains the geographicalExtent. The corresponding CityGML file is 10.1 MB.

5.1.3 Potsdam

The dataset is retrieved from https://de.ftp.opendatasoft.com/potsdam/Gebmodell3D_CityGML/Potsdam3D_3_6.zip. The size is 15.7 MB. The reference system of the dataset is EPSG:25833. Some of the city objects contain two geometric objects of LoD 1 and 2. The geometry types are MultiSurface and Solid. The geometric objects contain the property semantics. The semantic surface objects do not have parents and children. The city objects are of type Building. The metadata contains the geographicalExtent and PresentLoDs. The corresponding CityGML file is 80.5 MB.

5.1.4 User's location

The user is located in a building. The building is located in the delfshaven dataset. The position on a mobile device is given in latitude and longitude. The user is located in EPSG:4979 at latitude: 4.450846, longitude: 51.906183 and meters: 0.

5.2 Software

MongoDB version 4.2.8 is used as document database and PostgresQL version 12.3 as relational database, because they are open source. They are downloaded via brew from https://brew.sh/index_nl. The PostgreSQL database is downloaded with the postgis and postgis_sfcgal extensions. MongoDB compass from https://www.mongodb.com/try/download/compass is downloaded as Graphical User Interface (GUI) to view the data in MongoDB. PGadmin from https://www.postgresql.org/ftp/pgadmin/pgadmin4/v4.20/macos/ is downloaded as GUI to view the data in PostgreSQL.

3DCityDB is developed to efficiently insert and process CityGML files in relational databases. The CityGML data model is therefore simplified and mapped to a relational database schema [Stadler et al., 2009]. The 3DCityDB-importer-exporter version 4.2.0 is downloaded from https://www.3dcitydb.org/3dcitydb/downloads/. This software is used to create the 3DCityDB database schema in the existing PostgreSQL database with postgis, postgis_sfcgal and postgis_raster extensions. After that, the software is used to insert the CityGML files in the database.

The software val3dity from https://github.com/tudelft3d/val3dity is used to validate the geometries of the CityGML data model. It is used to validate the following geometry types: MultiSurface, CompositeSurface, Solid, MultiSolid, CompositeSolid. The Python library cjio of section 5.2.1 is used to validate CityJSON files against the CityJSON schema in particular.

The implementation are tested with Apache JMeter version 5.3 from https://jmeter.apache.org/download_jmeter.cgi. Apache JMeter has the ability to do performance measurements and load testing. JMeter simulates a group of users, thread groups, which are sending requests to a target server. JMeter collects statistic information about the requests and responses. The target server is the development server of the Flask app with GraphQL support. The SQL responses are also measured with JMeter. JMeter uses a Java Database Connectivity (JDBC) driver to send queries to the database. The database is in this way represented with an Uniform Resource Locator (URL). The JDBC driver can be downloaded from https://jdbc.postgresql.org/download.html and must be added to the lib folder of the JMeter directory.

5.2.1 Python libraries

The Python libraries are used to implement the methodology of chapter 4. The Python libraries are installed using pip. An overview of them can be seen in figure 5.1.

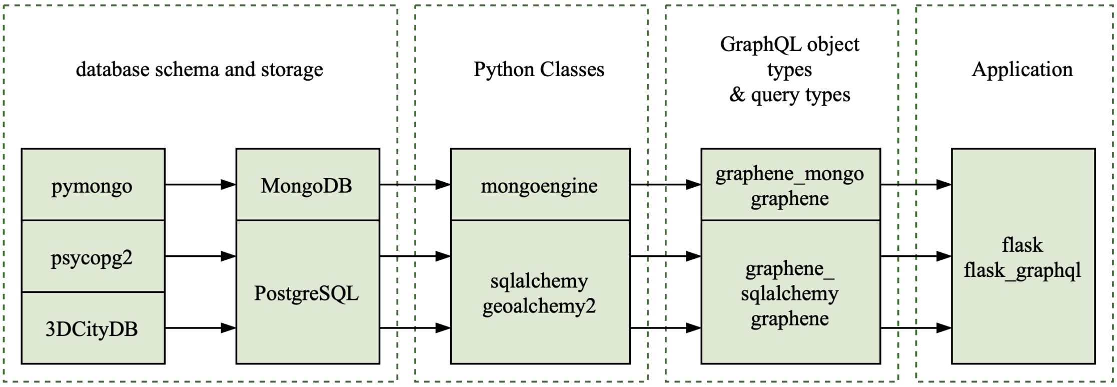


Figure 5.1: An overview of the used Python libraries

The libraries, that are used to create the database schemas and to store the data of section 5.1 in the databases, are psycopg2 and pymongo. psycopg2 connects the Python programming language to PostgreSQL. It is in this way possible to interact with the database via Python. The other library pymongo connects the Python programming language to MongoDB. They are used to create a database schema, to create indexes, to insert data, to alter data and to query data.

The libraries, that are used to support spatial operations and transformations, are pyproj, shapely and scipy.spatial. The library pyproj is a Python interface to PROJ, which is a coordinate transformation system to perform spatial reference transformations. The library is used to convert the user's location to the reference system of the city model in some cases. shapely is used to perform geometric PostGIS operations outside the database. It uses therefore spatial operations from the GEOS library. The spatial package from the scipy library is able to compute triangulations, convex hulls and to generate a k-d tree. The library is used to compute the convex hull of the city objects based on the x, y coordinates of the vertices. No indexes are built outside the database, because they would have to be rebuilt for every request separately.

The libraries, that are used to map the data in the database to Python classes, are `sqlalchemy`, `geolalchemy2` and `mongonegine`. The library `sqlalchemy` is an object-relational mapper for relational databases that use SQL. It presents a method of associating user-defined Python classes with database tables. The `geolalchemy2` library is an extension of `sqlalchemy` to support spatial databases. It is used to perform spatial operations on PostGIS geometry types, geography types and raster types. The `mongonegine` library is a Python object-document mapper for MongoDB. It associates user-defined Python classes with the documents in the collection of MongoDB.

The Python classes are also called models, which are used to develop the GraphQL API. The library `graphene` is used to build the GraphQL API in Python with the GraphQL object types, the GraphQL query types and the GraphQL schema. The libraries, that are used to create the fields of the GraphQL object type with the models, are `graphene_mongo` and `graphene_sqlalchemy`. `graphene_mongo` is graphene with built-in support for `mongoengine`, which makes it easier to operate with the models. The Meta class inside a GraphQL object type enables the developer to modify the fields of the GraphQL object type. The `MongoengineObjectType` specifically has the option `model`. This option is used to inspect the model and to create the fields of the model automatically. `graphene_sqlalchemy` is graphene with built-in support for `sqlalchemy`. The `SQLAlchemyObjectType` also has the option `model`.

Finally, the library `Flask` is used to build the web application in Python that is able to handle HTTP requests. The library `Flask-GraphQL` is used to add GraphQL support to a Flask application and it exposes the GraphQL schema through the API endpoint. GraphiQL is set to True to load the GraphQL endpoint in the browser.

SQLAlchemy documentations

```
dbschema=’cityjsondb, public’
connection = ”postgres+psycopg2://postgres:1234@localhost:5432/insertdb”
engine = create_engine(connection, connect_args=
{’options’: ’−csearch_path={}’.format(dbschema)})
made_session = sessionmaker(autocommit=False, autoflush=True, bind=engine)
db_session = scoped_session(made_session)
Base = declarative_base(cls=DeferredReflection)
Base.query = db_session.query_property()
```

Figure 5.2: Python script that creates Sessions to manage the interactions with the database through the Base class

The function `create_engine` is used to create a connection with the database as could be seen in figure 5.2. It creates an engine that is the core interface to the database. The function `sessionmaker` creates a `Session` that manages the interactions with the database. The `Session` automatically starts new transaction when a connection with the database is needed (`Automcommit` = False). The statement `Session.flush()` does not have to be called to retrieve results from the database (`autoflush` = True) and all SQL operations are executed via this `Session` (`bind = engine`). The `scoped_session` starts a new `Session` for every request. The function `declaritive_base` is used to create a Base class from which all mapped classes inherit.

The library is also used to create the Python classes or also called models. The relationships between the mapped classes of the relational databases of chapter 3 and section 4.1.2 are also mapped to the Python classes. The default behavior of a `sqlalchemy.Relationship` is that it joins the primary key of a class on the one-side and the corresponding foreign key of a class on the other-side. The function also accepts the following additional parameters:

- The parameter `backref` builds two individual `Relationship` constructors that refer to each other. It is possible to use it for multiple foreign keys that refer to the same primary key, but it is not possible to use the same named `backref`.
- The many-to-many relationship needs a primary join, a secondary join and the `secondary` table. The `secondary` table is for instance the `parents_children` table of section 4.1.2.
- The `remote_side` is used for self-referencing relationships. It specifies the column(s), which are on the `remote_side` of the relationship.

The `__mapper_args__` field is used to specify the inheritance relationship of section 3. It is not possible to specify a `sqlalchemy.Relationship` between the base-class and the sub-classes when the fields to join them on have both primary keys. The IDs of the sub-classes must therefore also obtain a foreign key constraint. The `__mapper_args__` field must also be specified for each involved Python class. The base-class specifies the field with the arguments `polymorphic_identity` and `polymorphic_on` and the sub-classes with `polymorphic_identity` and `inherit_condition`. The `polymorphic_identity` specifies its own table name. The `polymorphic_on` specifies the table name of the sub-class. The `inherit_condition` defines how the base-class and sub-class are joined. It is not allowed that the sub-class has the same attributes as the base-class, because the sub-class inherits the attributes from the base-class.

cjio

It is a Python command line utility to process, manipulate and validate CityJSON Files. The library is not only used to validate CityJSON files, but also to remove duplicate vertices from the reconstructed vertices list.

BSON

The `bson` library can be used to decode and encode BSON data. The library is used to generate `ObjectIds` for embedded documents. It is not a good solution, because the IDs change each time the field of the the GraphQL object type is queried.

5.3 PostgreSQL

5.3.1 Storage

The methodology of section 4.1.2 is implemented in the relational database PostgreSQL as described in section 5.2. The library `psycopg2` is used to connect PostgreSQL to Python. The connection is used at first to create a database with `postgis` and `sfcgal` extensions. The database schema is created as described in section 4.1.2. The database schema exists of tables, sequences and possibly the indexes of section 5.3.2. Information about the tables in PostgreSQL can be seen in appendix B. There are three changes relative to the methodology in section 4.1.2:

- The name of the `geometry` table is changed to `geometries` as could be seen in table description B.5, because the name `geometry` is already reserved for the `geometry` data type. The data types of the columns in a table have to be specified in the database schema.

- Two additional columns are added to the `city_object` table as could be seen in table description B.3 to store the convex hull of the city object in the `convexhull` column and the global convex hull of a city object in the `globalconvexhull` column. The reasons for this are described in section 5.3.7. The difference between them is that the convex hull is stored in the reference system of the city model and the global convex hull in the global reference system EPSG:4979.

- The relationships between `semantic surface` entities are not established, because they were not available in the data of section 5.1.

After the creation of the database schema, the CityJSON data of section 5.1 has been inserted in the database.

5.3.2 Indexes

PostgreSQL automatically creates a B-Tree index on every primary key.

However, it is in the created Python script for this thesis also possible to add additional indexes to the database schema. There are four additional indexes used. Three GIST indexes on the column `geometry` of the `surfaces` table, the column `convexhull` of the `city_object` table and the column `globalconvexhull` of the `city_object` table. The fourth index is a B-Tree index on the column `semantic_surface_id` of the `surfaces` table. The column `semantic_surface_id` refers to the corresponding semantic surface object.

5.3.3 Validation

Datasets	Denhaag	Delfshaven	Potsdam
Size differences	0.21 MB	3.719e-0.5 MB	0.876 MB

Different key-value pairs	`boundaries,`
	`semantics`
		`boundaries`	`boundaries,`
			`semantics`

Different cjio and val3dity warnings	No, they are the same.	No, they are the same.	No, they are the same.
The reason(s) behind the difference(s)	The vertices are inserted in a different order and the original CityJSON file duplicates the semantic surface object for every surface, which is not needed.	The vertices are inserted in a different order.	The vertices are inserted in a different order and the original CityJSON file duplicates the semantic surface object for every surface, which is not needed.

Table 5.1: Validation of the implementation of CityJSON in PostgreSQL

The stored CityJSON files are then validated as described in section 4.1.4. The key-value pairs of the CityJSON object of figure 2.3, the city objects of figure 2.4 and the geometry object of figure 2.5 are compared to narrow down the values. This makes it easier to clarify the reason(s) behind the different values and therefore the size differences of table 5.1. The sizes of the original and the new file are compared using the `os.path` module with the `getsize` method. The library `cjio` and the software `val3dity` are used as additional software to validate the original and new CityJSON file in their own way as described in section 5.2.

5.3.4 Database sizes

The database sizes are investigated for each dataset in section 5.1. The created databases exists of four schemas, which are the `information_schema`, the `pg_catalog` schema, the pg_statistic schema and the `cityjsondb` schema as could be seen in table 5.2.

The `information_schema` is always 0.35 MB without indexes. The `pg_catalog` schema is approximately 11 MB, because the size of the pg_statistic table slightly differs. The `public` schema is always 7.27 MB. It contains the spatial reference systems. The sizes of them together are approximately 18.62 MB of which 4.86 MB are indexes. The size of the `cityjsondb` differs per dataset.

5.3.5 Multiple city models

Although the database sizes are determined per CityJSON dataset, the use case requires multiple city models. The CityJSON datasets of section 5.1 are therefore also stored in one PostgreSQL database with one database schema as described in section 4.1.5.

Datasets	Denhaag	Delfshaven	Potsdam
information_schema of which are indexes	0.35 MB 0 MB	0.35 MB 0 MB	0.35 MB 0 MB
public schema of which are indexes	7.27 MB 0.3 MB	7.27 MB 0.3 MB	7.27 MB 0.3 MB
pg_catalog of which are indexes	10.8 MB 4.56 MB	10.74 MB 4.56 MB	10.93 MB 4.56 MB
cityjsondb of which are indexes	13.08 MB 3.62 MB	7.12 MB 1.85 MB	48.98 MB 11.16 MB
total database of which are indexes	31.51 MB 8.48 MB	25.48 MB 6.71 MB	67.53 MB 16.02 MB

Table 5.2: The schema sizes of CityJSON in PostgreSQL

5.3.6 Access

A connection with the database is established as described in section 5.2.1. The tables in the database are mapped to Python classes using sqlalchemy and geoalchemy2. The relationships of the relational database of section 4.1.2 are also mapped to the Python classes as described in section 5.2.1. These classes are also called models and they are mapped to GraphQL object types with the libraries graphene_sqlalchemy graphene.

Two changes are made to the GraphQL object types. The first one is made to return an attribute not with the data type JSONB, but with their own data type. A new field is therefore created in the GraphQL object type with one of the data types of GraphQL and the value is resolved with the data from the JSONB column. The second one resolves the values of the fields, that contain a Extended Well-Known Text (EWKT) geometry, as human readable text. The SQL query ST_AsText is therefore used.

5.3.7 Query

The GraphQL query types define the queries of section 4.2.2.

1. **location:** This query is defined with the locationQuery. The user provides its location and creates a GraphQL object type. The GraphQL query type accepts the user input and returns the created GraphQL object type. The input is not stored in the database.

2. **citymodel:** This query is defined with the citymodelQuery. There are many options to implement this query as described in section 4.3. The spatial operation is performed in the database, because it is possible to use spatial operations and to perform reference transformations in PostgreSQL. It was however not possible to use geometrical coordinates, because the library graphene_sqlalchemy ensures that the resolver function returns a SQL query instead of objects. A SQL query can only perform on geometries with one reference system. The geometric representation of every city model must therefore be transformed to the geographical coordinates of EPSG:4979. The geometric representation of the city model uses the geographicalExtent. The geographicalExtent is part of the metadata GraphQL object type. It is not possible in PostgreSQL to perform spatial operations in 3D on geographical coordinates without a reference transformations to a geometric reference system. The geometry is therefore not a 3D Box, but an envelope in 2D. The function ST_Intersects is used.

3. **radius100:** This query is defined in two ways with the radius100Query and the radius100indexQuery. There are many options to implement this query as described in section 4.3 as well.

   One of the options is to select the relevant city objects based on the ID of the city model. The ID of the city model would be provided as argument. The selected city objects would have the same reference system. However, it was not possible to the best of my knowledge to create one spatial index on the column, because the column contains city objects with multiple reference system. Another option might be to create multiple indexes on one column, one for every reference system, but this would require that the system knows which index to use. This option would therefore not work. Another option might be to create an index on the transformation. However, the index was not used. A reason might be that the sizes of the used datasets were too small, but this is not confirmed with larger datasets.

The geographical coordinates must be used just like the second query. Because it is not possible in PostgreSQL to perform spatial operations in 3D on geographical coordinates, the geometric representation of the city object must be 2D. A convex hull is therefore created based on the property `boundaries` of figure 2.5 and the function `ST_DWithin` is used.

Although it is preferred to perform all spatial operations in the database, the spatial index might require one reference system. There are therefore to different kinds of convex hulls inserted and added to the database schema as described in section 5.3.1. One of them contains the reference system of the city model and must therefore be transformed in the database. The other one is already transformed to the global reference system EPSG:4979 before the convex hull is inserted in the database. The `globalconvexhull` is used for the `radius100indexQuery` and the `convexhull` is used for the `radius100Query`. The city objects are also filtered based on the city object type and therefore the argument 'Building' is provided. The name `type`, which is used to represent the type of geometry and the type of city object, can not be used as argument when querying, because it is a built-in method in Python.

4. **inside:** This query is defined with the `insideQuery` and the `insideindexQuery`. They are almost the same as the queries of query 3 **radius100**. The difference is that they use the spatial operation `ST_Intersects` instead of `ST_DWithin`.

5. **maxlod:** The query is defined with the `cityobjectsQuery`. The resolver functions of individual GraphQL query types return SQL queries instead of the geometry objects themselves. The SQL query would have to be an aggregate query, because the geometry with the highest LoD would have to be returned. Another option is to access the geometry objects through the `city_objectType`. An additional field named `maxlod` can be added to the `city_objectType`. The `geometriesType` with the highest LoD is in this way resolved without an aggregate SQL query.

6. **cityobjects:** These queries are defined with the `cityobjectsQuery`.The attributes can be specified separately or together as a JSONB column as described in section 5.3.6. The geometries are retrieved as geometries with semantic surfaces with surfaces. The surfaces are not returned with the parent/child structure. An additional resolver would therefore be needed, but this resolver is not implemented in this research.

7. **surfaces:** This query is defined with the `surfacesQuery`. The surfaces have IDs, which makes it possible to filter the surface based on the ID. The attributes of the semantic surface are retrieved through the `object` field as described in section 4.1.2.

5.4 3DCityDB

5.4.1 Storage

The CityGML files are inserted in PostgreSQL with 3DCityDB as described in section 5.2. The tool creates the `citydb` schema, that contains tables, and the `citydb_pkg` schema, that contains functions. The details of the implementation of CityGML in 3DCityDB are described in section 3.

5.4.2 Indexes

3DCityDB uses many indexes as can be seen in appendix C. It uses the B-Tree index on the primary keys, foreign keys and other IDs, and the GIST index on geometries and the `envelope` of the city object.

5.4.3 Database sizes

The sizes of the schemas in the database are investigated for each dataset as can be seen in table 5.3. The selection of the used citydb tables are database_srs, cityobject, objectclass, building, surface_geometry, thematic_surface, cityobject_genericattrib and aggregation_info. They are described in section 3. The unused tables take up between 3 and 4 MB.

The pg_catalog schema sizes of CityGML in PostgreSQL with 3DCityDB differ from the pg_catalog schema sizes of section 5.3.1, because of the following tables: pg_shdepend, pg_depend, pg_proc, pg_attribute, pg_rewrite, pg_statistic, description, pg_class, pg_type, pg_index, pg_aggregate, pg_constraint and pg_trigger.

Datasets	Denhaag	Delfshaven	Potsdam
information_schema	0.35 MB	0.35 MB	0.35 MB
of which are indexes	0 MB	0 MB	0 MB
public schema	7.27 MB	7.27 MB	7.27 MB
of which are indexes	0.3 MB	0.3 MB	0.3 MB
pg_catalog	16.4 MB	16.3 MB	16.4 MB
of which are indexes	6.4 MB	6.4 MB	6.4 MB
citydb	61.85 MB	18.78 MB	154.3 MB
of which are indexes	38.66 MB	12.15 MB	92.56 MB
selection of the used citydb tables	58.21 MB	15.26 MB	150.78 MB
total database	85.9 MB	42.71 MB	178.34 MB
of which are indexes	45.45 MB	18.94 MB	99.35 MB

Table 5.3: The schema sizes of CityGML in PostgresQL with 3DCityDB

5.4.4 Multiple city models

It is possible with 3DCityDB to store different city models in one database with different schemas. The schema obtains the reference system of the first schema in the database. After that, the reference system of the schema can be changed using a SQL query.

Nevertheless, every database schema needs its own GraphQL schema. It might therefore be impossible or at least difficult to use these different schemas in combination with GraphQL. In order to use them in one GraphQL schema, the different GraphQL schemas have to be stitched together. However, the Python library graphene of section 5.2 does not support GraphQL schema stitching. It is therefore unknown how to implement GraphQL schema stitching and how it influences the data model mapping.

It is therefore not possible to use the three city models of section 5.1 in one GraphQL schema. The city models have to be accessed and queried separately with 3DCityDB.

5.4.5 Access

The basic implementation is approximately the same as section 5.3.6, but here the tables of section 3 are mapped to Python classes and the inheritance relationship is specified as described in section 5.2.1.

One change is made to the GraphQL object types. It resolves the values of the fields, that contain a EWKT geometry, as human readable text. The SQL query ST_AsText is therefore used.

5.4.6 Query

The city objects are not queried with their original ID as described in section 3.

1. **location:** This query is the same as in section 5.3.7.
2. **citymodel:** This query can not be implemented, because only one city model can be used at the same time as described in section 5.4.4. Additionally, the `citymodel` table remains empty when importing the CityGML files of section 5.1.
3. **radius100:** This query is defined with the `radius100Query`, which works approximately the same as the `radius100Query` of section 5.3.7. The are two differences. The first difference is that the `envelope` of the city object is used and the second one is that the city object type is not filtered with the argument 'Building'. The sub-class `buildingType` is queried instead of the base-class `cityobjectType`. Because the sub-class inherits the attributes from the base-class, the attributes of both the `buildingType` and the `cityobjectType` can be queried.
4. **inside:** This query is defined with the `insideQuery`. The query is almost the same as the query 3 **radius100**. The difference is that it uses the spatial operation `ST_Intersects` instead of `ST_DWithin`.
5. **maxlod:** This query is defined with the `buildingQuery`. The query is approximately the same as the fifth query of section 5.3.7. There are two differences. The first difference is that the field `maxlod` is added to the sub-class `buildingType` instead of the `cityobjectType`. The second difference is related to the resolver. The resolver returns the value based on multiple columns, because of two reasons. The first one is that the geometry can be represented in multiple ways as described in section 2.1.3. It is therefore possible that the geometry is not returned as one root-surface, but as multiple surfaces. The surfaces can be related to the sub-class `buildingType` and the sub-class `thematic_surfaceType`.
6. **cityobjects:** These queries are defined with the `cityobjectQuery`. The attributes are mapped to different GraphQL object types. The specified attributes are mapped to the base-class, which is the `cityobjectType`, and the sub-classes, which are the `buildingType` and the `thematic_surfaceType`. The generic attributes as described in section 2.1.2 are mapped to the `cityobject_genericattribType`. This means that the query contains the same attributes and only specifies those fields. Section 3 describes how generic attributes are mapped to PostgreSQL with 3DCityDB. The field named `value` is therefore defined and resolved based on the data type and the corresponding column. A field can only contain one data type and therefore every value is returned as a string, which is most commonly done as described in this blog `https://kamranicus.com/posts/2018-07-02-handling-multiple-scalar-types-in-graphql`. The implementation of query 6b and query 6c are adjusted to the `Delfshaven` dataset of section 5.1.2 and the implementation of CityJSON in PostgreSQL of section 5.3.7. The implementation of query 6c might not be suitable for other datasets, because the geometry can be represented in multiple ways as described in section 2.1.3
7. **surfaces:** This query is defined with the `surface_geometryType`. The surfaces have IDs, which makes it possible to filter the surface based on the ID. The surface exists of a parent and a child due to the parent/child structure. The parent contains the reference to the `thematic_surface` and the child contains the geometry. The attributes of the `thematic_surface` are not retrieved through a JSONB field as described in section 5.3.7, but the `thematic_surface` is related to the `cityobject_genericattribType`, which contains the attributes of the thematic surface.

5.5 MongoDB

5.5.1 Storage

The methodology of section 4.1.3 is implemented in the document database MongoDB as described in section 5.2. The library `pymongo` is used to connect MongoDB to Python. The connection is used at first to create a database. The collections are created as described in section 4.1.3. After the creation of the collection, the CityJSON data of section 5.1 is inserted in the database. There is one change made relative to the methodology of section 4.1.3. The attribute `presentLoDs` could not be inserted in MongoDB, because MongoDB does not allow to insert a dot as part of a key. The key ‘2.0’ can be seen in figure 5.3. This attribute informs the user about the LoDs that are present in the city model.

```
"presentLoDs":{"2.0":145865}
```

Figure 5.3: The attribute “presentLoDs”

5.5.2 Indexes

Every collection has a primary key. The primary key is the `_id` field. MongoDB creates automatically an index with a B-Tree structure on the primary key of every collection.

5.5.3 Validation

Datasets	**Denhaag**	**Delfshaven**	**Potsdam**
Size differences	0.0076 MB	3.719e-05 MB	4.10e-05 MB
Different key-value pairs	`boundaries`	`boundaries`	`presentLoDs`
Different cjio and val3dity warnings	No, they are the same.	No, they are the same.	No, they are the same.
The reason(s) behind the difference(s)	The vertices are inserted in a different order.	The vertices are inserted in a different order.	The attribute could not be stored in MongoDB.

Table 5.4: Validation of the implementation of CityJSON in MongoDB

The stored files are then validated as described in section 4.1.4. The comparison is done in the same way as the implementation of section 5.3.3. The results of the validation can be seen in table 5.4. The differences are smaller than the implementation in PostgreSQL of table 5.1, because the original file does not remove duplicated `semantic surface` object and MongoDB does not do that either.

Schema validation

Schema validation is possible in MongoDB using the $jsonSchema operator, but not all features of a JSON schema are supported as described in section 2.2.3. It is therefore necessary to adjust the CityJSON schemas of `https://3d.bk.tudelft.nl/schemas/cityjson/` before they can be used in MongoDB. This has only been implemented for the `transform` and `metadata` collection in this thesis.

5.5.4 Database sizes

The sizes of the collections are investigated for each dataset as can be seen in table 5.5. The database schema is related to the collections. The size of the database is however not only related to the collections, but also to other files as described in section 2.2.3. The data sizes are relatively small, because data replication as described in section 2.2.3 is not enabled.

5.5.5 Multiple city models

The three city models can be stored in the same database in the same collections as described in section 4.1.5.

Datasets	Denhaag	Delfshaven	Potsdam
CityObjects collection			
data size	5.58 MB	3.65 MB	28.04 MB
storage size	1.20 MB	0.93 MB	6.79 MB
index size	0.07 MB	0.06 MB	0.152 MB
metadata collection			
data size	0.0002 MB	0.0002 MB	0.0002 MB
storage size	0.02 MB	0.02 MB	0.02 MB
index size	0.02 MB	0.02 MB	0.02 MB
transform collection			
data size	0.0002 MB	0.0002 MB	0 MB
storage size	0.02 MB	0.02 MB	0.004 MB
index size	0.02 MB	0.02 MB	0.0176 MB
total			
data size	5.58 MB	3.65 MB	28.04 MB
storage and index size	1.35 MB	1.07 MB	6.986 MB
journal files	104.9 MB	104.9 MB	104.9 MB

Table 5.5: The file sizes of CityJSON in MongoDB

5.5.6 Access

A connection with the database is established as described in section 5.2. The collections in the database are mapped to Python classes using mongoengine. The relationships of the document database of section 4.1.3 are also mapped using a ReferenceField or an EmbeddedDocumentField. The ReferenceField is able to dereferences the reference to a document automatically. However, it is not possible to the best of my knowledge to implement them with the Python libraries due to the different types of references in MongoDB as described in section 2.2.3. The references are therefore mapped to a field with a string as data type and resolved manually. It was not possible to map the property boundaries as described in section 2.1.3. Many options have been tried, but no solution has been found. Although it might be possible to return the boundaries as a string, it was decided to return only the MultiSurface geometry type. This means that only this geometry type is stored, accessed and queried.

These classes are also called models and they are mapped to GraphQL object types using the library graphene_mongo and graphene.

5.5.7 Query MongoDB

The queries are performed on the three city models, but they do not contain all their geometries, only their MultiSurfaces. The reason is that only the geometry type MultiSurface can be queried as described in section 5.5.6.

1. **location:** The first query is the same as in section 5.3.7.
2. **citymodel:** This query is defined with the citymodelQuery, which queries the CityObjectsType. There are many options to implement this query as described in table 4.3. The spatial operation is performed outside the database, because it is not possible to perform reference transformation in MongoDB as described in section 4.1.3. Because of this, the resolver has to iterate over all objects of the MetadataType. Each MetadataType contains the geographicalExtent of the city model. The library shapely is used to perform the spatial operations outside the database, but they only perform them on geometric reference systems in 2D. The geographicalExtent is therefore converted to a Polygon and the user's location is transformed to the reference system of the city model using the library pyproj. The spatial operation intersects of the library shapely is used between the city model and the user's location.
3. **radius100:** This query is defined with the radius100Query, which queries the CityObjectsType. The same options are considered as in query 2 **citymodel**, but now the resolver function has two iterations. The first one is over the objects of the MetadataType to get the reference system and to transform the user's location to it. The second one is over the city objects of the CityObjectsType. The vertices of

the boundaries are used to form a convexhull, because it represents the surface of the city object better. The city objects of which the distance between the convexhull and the user's location are less than 100 meters are returned. The city objects are also filtered based on the city object type and therefore the argument 'Building' is provided. The name type, which is used to represent the type of geometry and the type of city object, can not be used as argument when querying, because it is a built-in method in Python. The name objtype or geomtype are therefore used.

4. **inside:** This query is defined with the insideQuery. It works approximately the same as the third query, but it uses an intersection between convexhull and the user's location instead of the distance.

5. **maxlod:** This query is defined with the MaxLoDQuery. The objects of the geometryType are accessed through the CityObjectsType, but the geometry with the highest LoD is returned on its own. It is possible to use an additional Graph query type, because the resolver can operate on the objects and does therefore not require an aggregate MQL query. The embedded documents of MongoDB do not necessarily have an ID. It is therefore possible to create an ObjectId in the model of the the embedded document, but the ObjectId changes every time the object is queried as described in section 5.2.

6. **cityobjects:** These queries are defined with the CityObjectsQuery. The attributes are defined in the embedded AttributesType. The attributes are specified individually in the request and returned with their original data type. The geometries from the GeometryType can contain semantics object from the SemanticsType, which can contain the semantic surface objects from the Semantic_surfaceType.

7. **surfaces:** This query is not implemented, because the embedded documents do not have their own IDs that are stored in the database. Although the surfaces are not stored separately, a resolver function can return an individual surface together with its semantic surface object.

5.6 Experiments

The experiments are broadly explained in section 4.3. An overview of them can be seen in figure 5.6. The number of city models differs, because the implementations differ. The implementation of CityJSON in MongoDB and the implementation of CityGML in PostgresQL with 3DCityDB are both compared to the implementation of CityJSON in PostgreSQL.

Sometimes there are also additional indexes added to improve the retrieval times. The additional indexes of the implementation of CityJSON is PostgreSQL are described in section 5.3.2. The implementation of CityGML in PostgreSQL with 3DCityDB already has many additional indexes and the implementation of CityJSON in MongoDB has not. Furthermore, MongoDB only handles `MultiSurfaces` as described in section 5.5.6.

Comparison							
Experiment 1				**Experiment 2**			
Implementation	Number of city models	Additional indexes	Thread group	Implementation	Number of city models	Additional indexes	Thread group
CityJSON in MongoDB	3	no	1 user	**CityJSON in PostgreSQL**	3	no	1 user
CityJSON in PostgreSQL	3	**no**	1 user	CityJSON in PostgreSQL	3	**yes**	1 user
CityJSON in PostgreSQL	**3**	no	1 user	CityJSON in PostgreSQL	**1**	no	1 user
CityJSON in PostgreSQL	**3**	yes	1 user	CityJSON in PostgreSQL	**1**	yes	1 user
CityJSON in PostgreSQL	1	yes	1 user	**CityGML in PostgreSQL with 3DCityDB**	1	yes	1 user

Table 5.6: An overview of the retrieval time experiments (1 user sends the query a 100 times)

The thread groups are created with `JMeter`. The software `JMeter` is described in section 5.2.

5.6.1 Retrieval times

The retrieval times are measured for each experiment of figure 5.6 and for each defined query of appendix D, E and F individually. This has been done as explained in figure 5.7.

Step 1.	Query GraphQL with JMeter for each defined query
Step 2	Store the logfile of the database
Step 3.	Detect errors based on the HTTP code and response sizes
Step 4.	Analyse the SQL/MQL retrieval times using the logfiles
Step 5.	Link the results of step 1 and 4
Step 6.	Remove outliers based on the results of step 1
Step 7.	Re-calculate the mean and standard deviation of the GraphQL retrieval times
Step 8.	Calculate the mean and standard deviation of the SQL/MQL retrieval times

Table 5.7: An overview of the retrieval time experiments (1 user sends the query a 100 times)

The software `JMeter` is used in step 1 of figure 5.7 to measure the GraphQL retrieval times. Separate logfiles are created to analyse them for each query individually. The database server must therefore be started and stopped after each experiment as can be seen in figure 5.4.

```
PostgreSQL:
$ pg_ctl –D /usr/local/var/postgres stop
$ pg_ctl –D /usr/local/var/postgres start

MongoDB:
$ mongod ––config /usr/local/etc/mongod.conf
(CTRL + C) to shut down MongoDB
$ mongod ––logpath /usr/local/var/log/mongodb/mongo.log
```

Figure 5.4: Start and stop database servers

After that, it possible to analyse the results in Excel. First, the errors are detected using the HTTP codes that are monitored with `JMter` and using deviating response sizes as explained in section 2.3.2. A deviating response size indicates an error message. This is done in step 3. The logfiles of PostgreSQL with SQL queries are analysed in Python, but the logfiles of MongoDB are not. They do not store the duration of the MQL query. The logfile of PostgreSQL contains the duration of each SQL query. In case, 1 user sends each query a 100 times, 100 durations are measured. The durations for each SQL query are stored in a .csv file. There are 100 results and therefore 100 rows if there are no detected errors. The results of step 1 and step 4 are related to each other in Excel with the detected errors in mind.

The detection of outliers is based on the GraphQL retrieval times, because they measure the entire architecture. An outlier is considered a result that deviates more than two times the standard deviation from the mean. These outliers are removed from the linked results. After that, the mean and standard deviation of the GraphQL retrieval times are re-calculated. In addition, the mean and standard deviation of the SQL retrieval times are calculated for the first time.

5.6.2 Request and response sizes

The request and response sizes are also measured for each query of appendix D, E and F individually. They are only measured once, because they stay approximately the same regardless of the number of city models or additional indexes. The experiment to measure the transferred data can be seen in figure 5.5. The request and response sizes are measured in bytes.

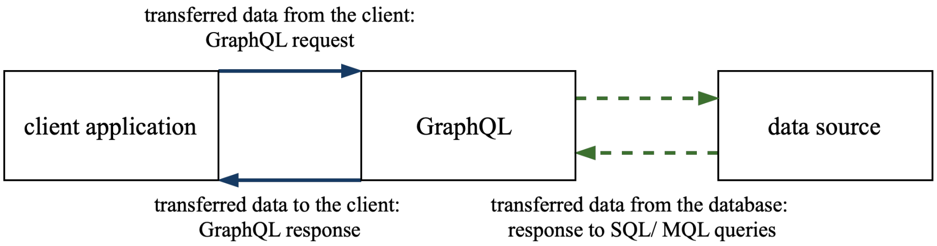

transferred data from the client:
GraphQL request
client application
GraphQL
data source
transferred data to the client:
GraphQL response
transferred data from the database:
response to SQL/ MQL queries

Figure 5.5: Experiment to measure the transferred data

Step 1 of figure 5.7 is used, because `JMeter` also returns statistic information about the network performance. The network performance is described in section 4.3. The logfiles are used to identify the queries that are send to the database in order to answer the GraphQL query of the client. SQL queries are used for relational databases. MQL queries are used for the document database MongoDB. The identified queries can be seen in appendix D, E and F.

The response sizes are investigated with `JMeter` for each SQL and MQL query individually. `JMeter` uses a JDBC driver to represent PostgreSQL with an URL. The number of threads does not matter, since the response sizes stay the same. `JMeter` returns the size of the transferred data from the database to GraphQL. The MQL queries could not be measured using `JMeter`. The sizes of the MQL queries are therefore measured with the MongoDB Shell script of figure 5.6. These response sizes of the SQL and MQL queries are smaller than the responses to the client,

```
var cursor = db.metadata.find();
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

Figure 5.6: MongoDB Shell script to measure the response size of the MQL query

because the responses to the client have a request header of 145 bytes. The 145 bytes have to be subtracted from the GraphQL responses in order to measure over-fetching from the database.

Sometimes multiple queries are send to the database in order to answer the GraphQL query of the client. The response sizes of the SQL/MQL queries are then added together.

The HTTP request and the HTTP response sizes are added together in order to identify the network performance. Also, the transferred data from the database and the transferred data to the client are compared to each other in order to identify over-fetching. The results can be explained by means of the queries in appendix D, E and F.

5.6.3 Computer environment

	13-inch MacBook (64-bit) Pro
Processor	2,7 GHz Dual-Core Intel Core i5.
Memory	8 GB 1867 MHz DDR3

Table 5.8: Laptop specifications

6 Results

6.1 Database size differences

The implementations store different files. The implementations of CityGML in PostgreSQL with 3DCityDB and of CityJSON in PostgreSQL are compared first. After that, the implementations of CityJSON in MongoDB and in PostgreSQL.

The schema sizes of the `information_schema` and the `public_schema` are the same. The schema size of the `pg_catalog` slightly differs, because the size of CityJSON in PostgreSQL is approximately 11 MB and CityGML in PostgreSQL with 3DCityDB is approximately 16 MB. The main difference is based on the created database schemas. The created schema for CityJSON is called `cityjsondb` and the created schema for CityGML with 3DCityDB is called `citydb`. The datasets are more compressed with CityJSON as exchange format than with CityGML as described in section 5.1. The reductions of the `cityjsondb` schema relative to the `citydb` schema is less than the reduction of CityJSON relative to CityGML as can be seen in table 6.1. One of the reasons that the reductions are less might be that CityJSON uses an indexing mechanism for the vertices and the `transform` object to represent the real coordinates as integers as described in section 2.1.3. The indexes are however resolved and the vertices are transformed to real coordinates before they are inserted in the database as described in section 4.1.2.

Datasets	Denhaag	Delfshaven	Potsdam
`citydb` (3DCityDB)	61.85 MB	18.78 MB	154.3 MB
`cityjsondb` (PostgreSQL)	13.08 MB	7.12 MB	48.98 MB
Reduction `cityjsondb` %	79 %	62 %	68 %
CityGML	21.2 MB	10.1 MB	80.5 MB
CityJSON	2.6 MB	1.4 MB	15.7 MB
Reduction CityJSON %	88 %	86 %	80 %

Table 6.1: Schema size comparison between CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB per dataset

The implementations of CityJSON in MongoDB and in PostgreSQL use the same exchange format. The database size of MongoDB exists of different files and sizes as explained in section 2.2.3. It is more efficient to store the CityJSON files compressed on disk than in a CityJSON file, but it is less efficient to store them uncompressed in the database. The results show that it is more efficient to store CityJSON in MongoDB than in PostgreSQL. One of the reasons might be that the geometries are stored less efficient in PostgreSQL and that more foreign keys are stored. However, the comparison is incomplete. The journal files take up more than 100 MB in MongoDB and data replication is not enabled yet.

Datasets	Denhaag	Delfshaven	Potsdam
MongoDB (compressed)	1.35 MB	1.07 MB	6.986 MB
MongoDB (uncompressed)	5.58 MB	3.65 MB	28.04 MB
`cityjsondb` (PostgreSQL)	13.08 MB	7.12 MB	48.98 MB
Reduction MongoDB %	58 %	49 %	43 %

Table 6.2: Storage size comparison between CityJSON in PostgreSQL and MongoDB per dataset

6.2 Access and query differences

Section 4.2.2 explained which requirements the queries investigate. The results of these requirements are summarized in table 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8 for each implementation.

Query 1	The user’s position is provided and returned correctly.	
PostgreSQL	Yes	
3DCityDB	Yes	
MongoDB	Yes	

Table 6.3: Access and query differences per implementation for query 1

Query 2	Multiple city models with different reference systems can be queried through GraphQL	
PostgreSQL	Yes	The city models are stored in one database with one schema as described in section 5.3.5, but this results in difficulties concerning the spatial indexes and spatial operations. The spatial indexes are built on a specific reference system and spatial operations can only be executed on geometries with the same reference system.
3DCityDB	No	One database can have multiple database schemas. It is possible to access them using dot.notations in the database, but not in one GraphQL schema using the Python library `graphene` as described in 5.4.4.
MongoDB	Yes	The city models can be stored in one database using the same collection as described in section 5.5.5, but reference transformation and spatial operations are performed outside the database.
Query 2, 3, 4	The database supports spatial reference transformations.	
PostgreSQL	Yes	
3DCityDB	Yes	
MongoDB	No	It only supports GeoJSON objects in the WGS84 reference system with geographical coordinates.
Query 2, 3, 4	The database supports geometrical and geographical spatial operations	
PostgreSQL	Yes/No	Yes for 2D operations and no for 3D operations as explained in section 5.3.7. 3D operations can only be performed on geometric coordinates.
3DCityDB	Yes/No	Yes for 2D operations and no for 3D operations as explained in section 5.3.7. 3D operations can only be performed on geometric coordinates.
MongoDB	No	Only geographical spatial operations in 2D
Query 2, 3, 4	The database supports the geometry definitions of the CityGML data model.	
PostgreSQL	No	Many geometry types can be stored with their reference system, but there is no option to store `MultiSolids` and `MultiComposites` with voids as explained in section 4.1.2. Another issue is that the link with the semantic surface objects can not be maintained when using the supported geometry type `MultiSurfaces` for instance.
3DCityDB	No	
MongoDB	No	The reasons are described in section 4.1.3
Query 3, 4	It is possible to filter on the type of city object in GraphQL	
PostgreSQL	Yes	The name `objtype` is used instead of `type`, because `type` is a built-in method in Python as described in section 5.3.7.
3DCityDB	Yes	The sub-class `buildingType` is queried instead of the base-class `cityobjectType` as described in section 5.4.6. The sub-class contains multiple CityGML classes (AbstractBuilding, Building and BuildingPart). A filter on the `objectclass_id` would be needed as well, but this is not implemented.
MongoDB	Yes	The name `objtype` is used instead of `type`, because `type` is a built-in method in Python as described in section 5.3.7.

Table 6.4: Access and query differences per implementation for query 2, 3 and 4

Query 5	It is possible to access and operate on the LoD attribute.	
PostgreSQL	Yes	
3DCityDB	No	There is no separate LoD attribute as described in section 5.4.6.
MongoDB	Yes	

Table 6.5: Access and query differences per implementation for query 5

Query 6b	It is possible to retrieve the attribute data of the city object based on the implementation.	
PostgreSQL	Yes	
3DCityDB	Yes	
MongoDB	Yes	
Query 6b	**How are the attributes returned based on the implementation?**	
PostgreSQL		They can be returned separately by specifying the fields, but they can also be returned together as JSONB data.
3DCityDB		There are specified attributes and generic attributes as described in section 3. They are resolved as described in section 5.4.6 and returned as a string.
MongoDB		They are returned separately by specifying the fields.

Table 6.6: Access and query differences per implementation for query 6b

6c	How are the geometries returned based on the implementation?	
PostgreSQL		The geometries are returned as `geometries` with `semantic_surfaces` with `surfaces` or in a different order.
3DCityDB		The geometries are returned as a building with thematic surfaces with surfaces. However, the geometries can be represented in multiple ways as explained in 2.1.3. This makes the it more difficult to implement the query and therefore less reliable.
MongoDB		The geometry is returned as the properties described in figure 2.5, 2.6 and 2.7.

Table 6.7: Access and query differences per implementation for query 6c

Query 7	It is possible to filter based on the IDs of a surface based on the implementation.	
PostgreSQL	Yes	
3DCityDB	Yes	
MongoDB	No	The geometries and surfaces do not automatically have their own ID, because they are embedded documents. It might be possible to add them.
Query 7	**It is possible to retrieve the surface and the semantic surface object together.**	
PostgreSQL	Yes	There is a reference between them.
3DCityDB	Yes	There is a reference between them.
MongoDB	Yes	A resolver function is needed to retrieve the surfaces and to obtain the index value of the semantic surface object. The semantic surface object and the surface can be returned together.

Table 6.8: Access and query differences per implementation for query 7

6.3 Number of SQL/MQL queries

The SQL/MQL queries that are send to the database in order to answer the GraphQL query of the client can be seen in appendix D for CitJSON in PostgreSQL, appendix E for CityGM in PostgreSQL with 3DCityDB and appendix F for CityJSON in MongoDB. These SQL/MQL queries are identified with the logfiles of the databases as described in section 5.5. Query 1 does not access the database and therefore no SQL/MQL query is send to the database.

6.3.1 Comparison between CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB

Query 2 is excluded, because it is not implemented with 3DCityDB as described in section 5.4.6.

The number of SQL queries is lower for CityJSON in PostgreSQL in case of query 5, 6b, 6c and 7. The number is the same for the other queries as can be seen in figure 6.1.

The lower number has multiple reasons. The first one is related to query 5. The implementation with 3DCityDB has to investigate two sub-classes and many surfaces as described in section 5.4.6. This is due to the fact that

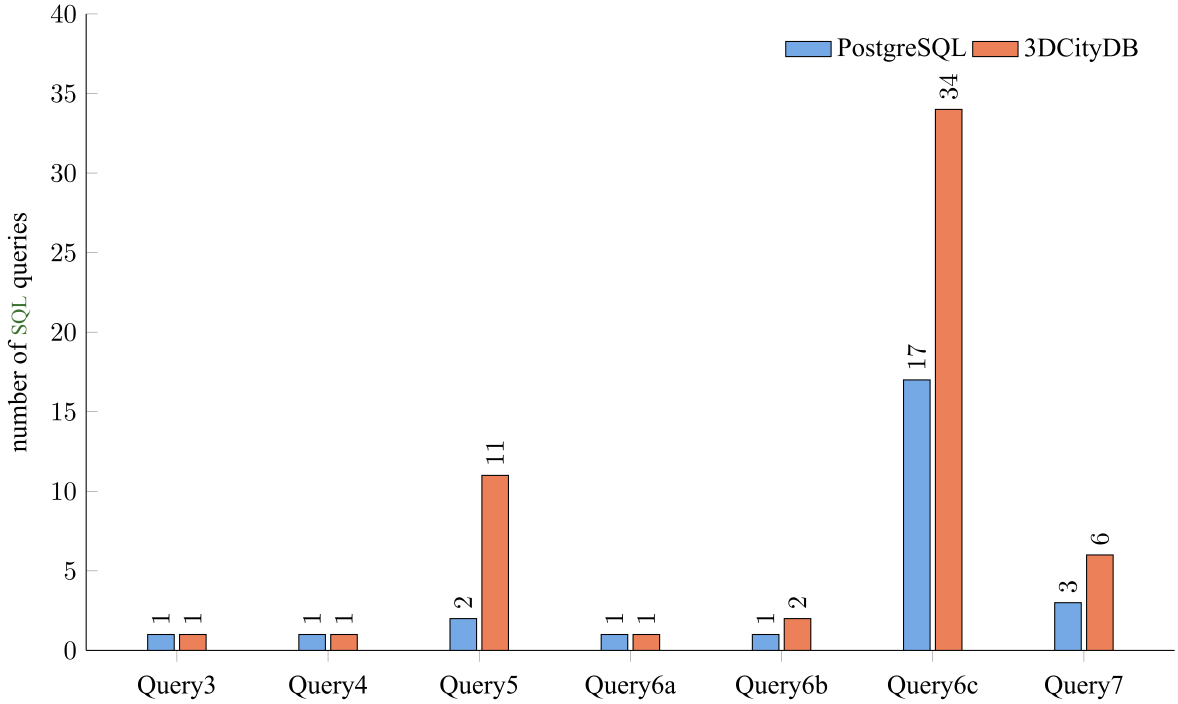


Figure 6.1: Comparison between the number of SQL queries for CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB per defined query

CityGML can represent geometries in many ways as described in section 3. The implementation of CityJSON in PostgreSQL investigates on the other hand the whole geometries related to a city object.

The next one is related to query 6b. The implementation with 3DCityDB queries the generic attributes separately from the `cityobjectType` with the `cityobject_genericattribType`. The implementation of CityJSON in PostgreSQL queries the JSONB column of the `city_objectType`. This column already contains the generic attributes.

Query 6c is higher for the implementation with 3DCityDB, because the implementation with 3DCityDB queries surfaces multiple times due to the parent/child structure, while the implementation of CityJSON in PostgreSQL only queries a surface once. Additionally, the implementation with 3DCityDB has to query the name of the thematic surface separately. The name is stored in the `objectclass` table.

Query 7 combines some of the reasons that are mentioned earlier. The implementation in 3DCityDB has three queries more, because the name of the thematic surface is queried separately, the generic attributes are queried separately and a child has to be queried due to the parent/child structure.

6.3.2 Comparison between the CityJSON in PostgreSQL and in MongoDB

Query 7 is excluded, because it is not implemented in MongoDB as described in section 5.5.7.

The number of MQL queries is higher for MongoDB in case of query 3 and 4. The number of SQL queries is higher for PostgreSQL in case of query 5 and 6c. The number is the same for the other queries as can be seen in figure 6.2.

The number of MQL queries is higher for query 3 and 4, because the city objects of each city model are queried separately. They are queried separately, because the spatial operations of the Python library `shapely` can only be performed on geometric coordinates and geometries with the same reference system as described in section 5.5.7. The spatial operations can not be performed in the database, because the database is not able to do spatial reference

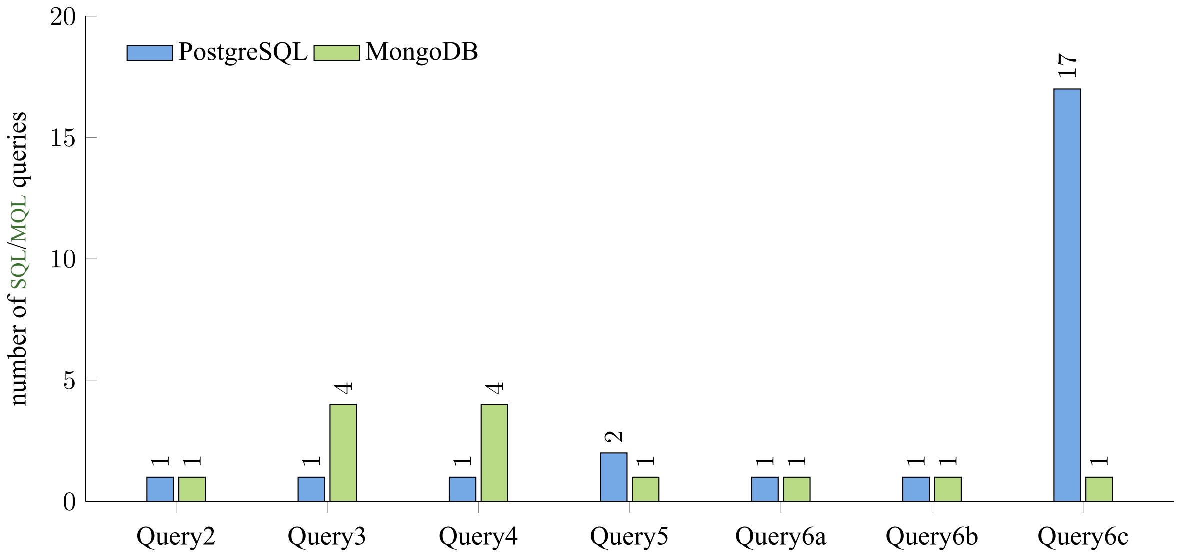


Figure 6.2: Comparison between the number of SQL/MQL queries for CityJSON in PostgreSQL and in MongoDB per defined query

transformation. However, PostgreSQL is able perform spatial reference transformations as described in section 5.3.7.

On the other hand, the number of SQL queries is higher for query 5 and 6c, because MongoDB uses embedded documents. The geometries, the surfaces and the semantic surface objects are embedded in the documents of the `CityObjects` collection and therefore queried at once. The PostgreSQL implementation on the other hand needs to query these objects separately, which increases the number of queries.

6.4 Request and response sizes

The experiment to measure the request and response sizes is explained in section 5.6.2. Each subsection describes the transferred data between the client and GraphQL, and over-fetching from the database, because GraphQL does not use all the data that it gets from the database.

6.4.1 PostgreSQL

The HTTP request sizes are quite small, but the HTTP response sizes to the client vary as could be seen in figure 6.3. The responses of query 3 and query 6C are relatively large. The reasons for this are that query 3 returns the IDs of multiple city objects instead of one ID and query 6c returns the whole geometry of a city object. The geometry is returned as individual surfaces. These surfaces are represented as a `PolygonZ` geometry with additional data about the position of the surface in the hierarchy.

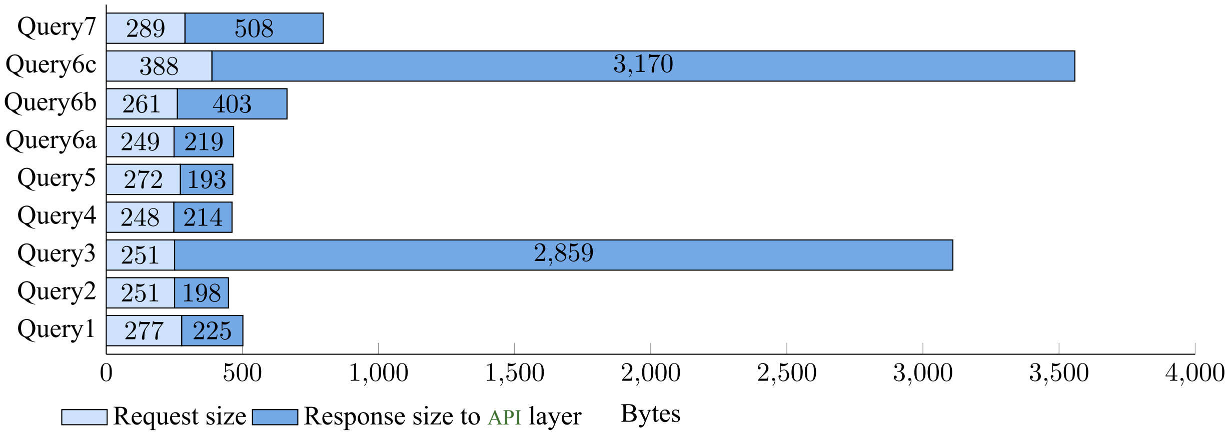


Figure 6.3: Transferred data between the client and GraphQL for CityJSON in PostgreSQL per query

The transferred data that is used for the GraphQL response can be seen in figure 6.4. The implementation obtains more data from the database than it uses for the GraphQL response. The GraphQL response only uses the specified fields, but the database returns all attributes from the queried tables. This means more precisely that the queries only use the ID of the object, while all the fields from the `city_object` table are returned. Approximately the same applies to the other queries. When more fields are specified in the GraphQL request, less fields are redundantly returned from the database. Although all fields seem to be specified for query 7, the database still returns redundant fields such as foreign keys and it returns the geometry twice.

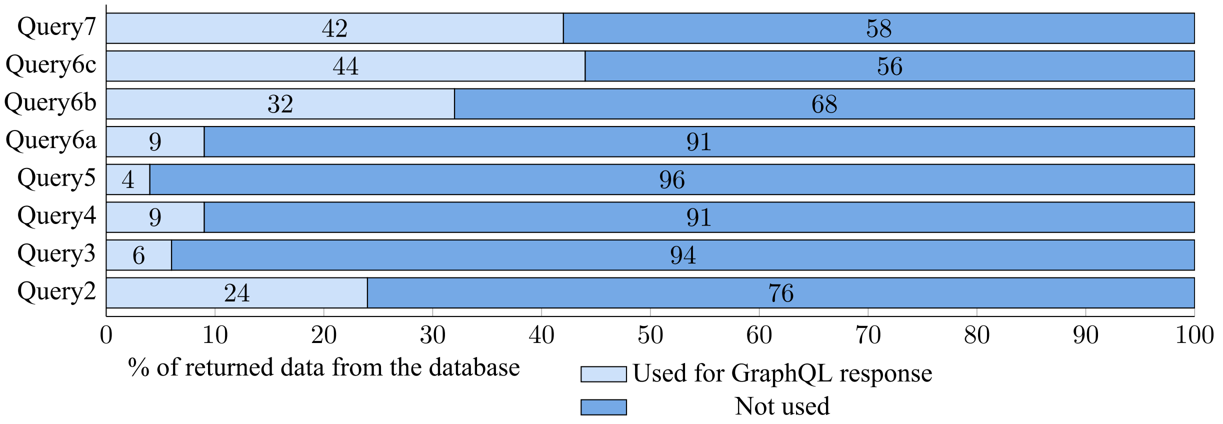


Figure 6.4: Over-fetching from the database per query for the PostgreSQL implementation

6.4.2 3DCityDB

The HTTP request sizes are quite small, but the HTTP response sizes to the client vary as could be seen in figure 6.3. The responses of query 3 and query 6c are relatively large. The reasons are the same as described in section 6.4.1, but the surfaces are represented as a PolygonZ geometry with additional data about the the parent/child structure and the type of geometry. Other reasons are that the objectclassType contains the name of the thematic surfaces class and the buildingType has to be specified inside the cityobjectType to access the attributes that belong to the building class.

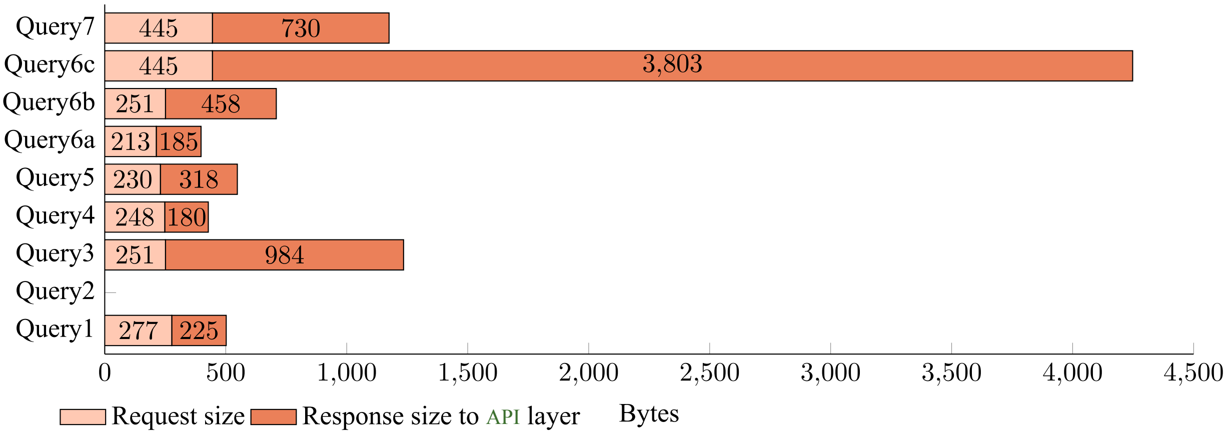


Figure 6.5: Transferred data between the client and GraphQL of CityJSON in PostgreSQL with 3DCityDB per query

The transferred data that is used for the GraphQL response can be seen in figure 6.6. Query 2 can not be executed with 3DCityDB as explained in section 5.4.6. The GraphQL response only uses the specified fields, but the database returns all attributes from the queried tables. This is not ideal, because the implementation scatters the data model over multiple tables and the tables have many attributes. Examples are:

- The cityobject_genericattribType stores generic attributes. The values of the attributes have to be resolved based on many columns.
- The objectclassType contains the name of the thematic surfaces class.
- The sub-classes automatically inherit the attributes from the base-class due to the inheritance relationship. The sub-classes also store many XML tags as columns such as all the geometry type features of CityGML.

Another reason is the parent/child structure for geometries. The structure can cause that geometries are requested multiple times.

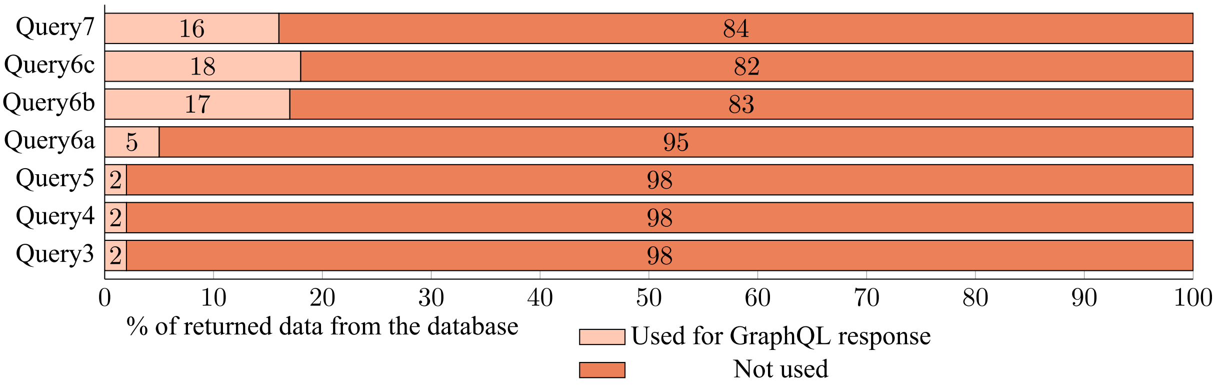


Figure 6.6: Over-fetching from the database per query for CityGML in PostgreSQL with 3DCityDB

6.4.3 MongoDB

Query 7 could not be implemented as described in section 5.5.7. The HTTP request sizes are quite small, but the HTTP response sizes to the client vary as could be seen in figure 6.7. The responses of query 3 and query 6c are relatively large. The reasons are the same as described in section 6.4.1. The whole geometry is returned as the properties described in figure 2.5, 2.6 and 2.7.

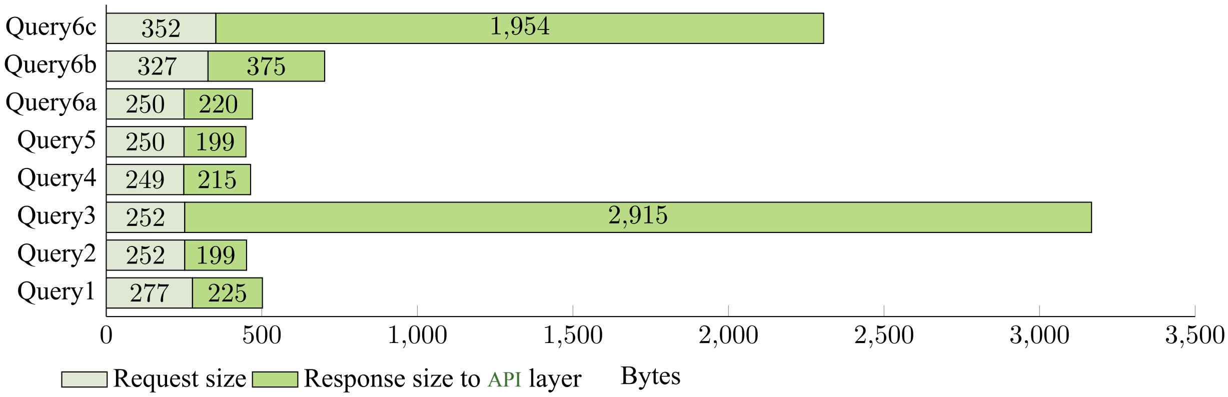


Figure 6.7: Transferred data between the client and GraphQL for CityJSON in MongoDB per query

The transferred data that is used for the GraphQL response can be seen in figure 6.8. Query 1 does not query the database and therefore no data is transferred between the database and GraphQL. When only a few fields of the documents are specified, then a lot of data is not used for the GraphQL response. The reason for this is that the MongoDB implementation always returns the entire document with its embedded documents regardless of the fields that are queried. Query 3 and query 4 use even less than 1 percent of the data from the database. On the contrary, query 6c specifies most of the city object fields and therefore a significant part is used for the GraphQL response.

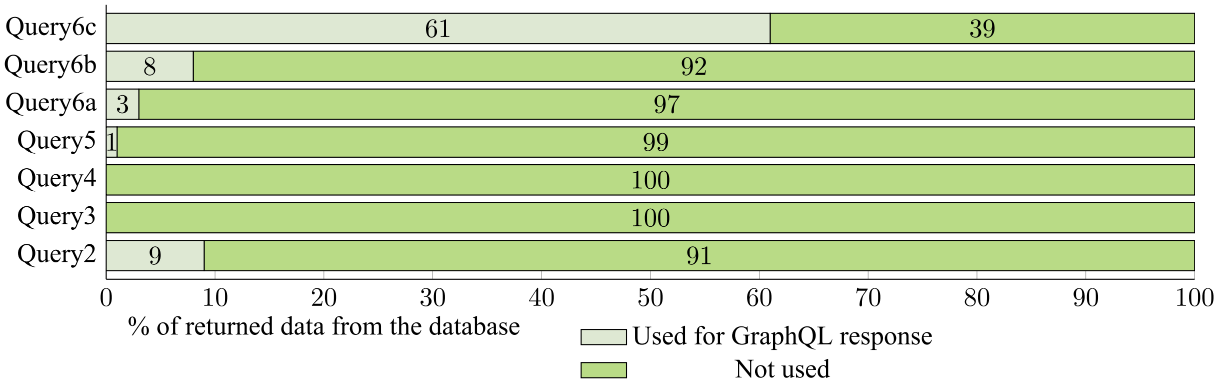


Figure 6.8: Over-fetching from the database per query for CityJSON in MongoDB

6.4.4 Comparison between CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB

The HTTP request sizes are compared as well in figure 6.9. The request sizes of query 3 and 4 are exactly the same. The requests of query 5, 6a and 6b are larger for CityJSON in PostgreSQL, because the implementation with 3DCityDB uses a sequence with shorter IDs for the city objects. The difference would likely be negligible when the implementation with 3DCityDB would also use the original ID, the `gml_id`, of the city object. The requests of query 6c and 7 are larger for the implementation with 3DCityDB. While the implementation with 3DCityDB and the one of CityJSON in PostgreSQL use the same number of fields for query 6c, the small difference can be explained due to the length of the field names. The request size difference of query 7 can be explained due to the number of requested fields. The implementation with 3DCityDB requests 15 fields and the one of CityJSON in PostgreSQL requests 9 fields.

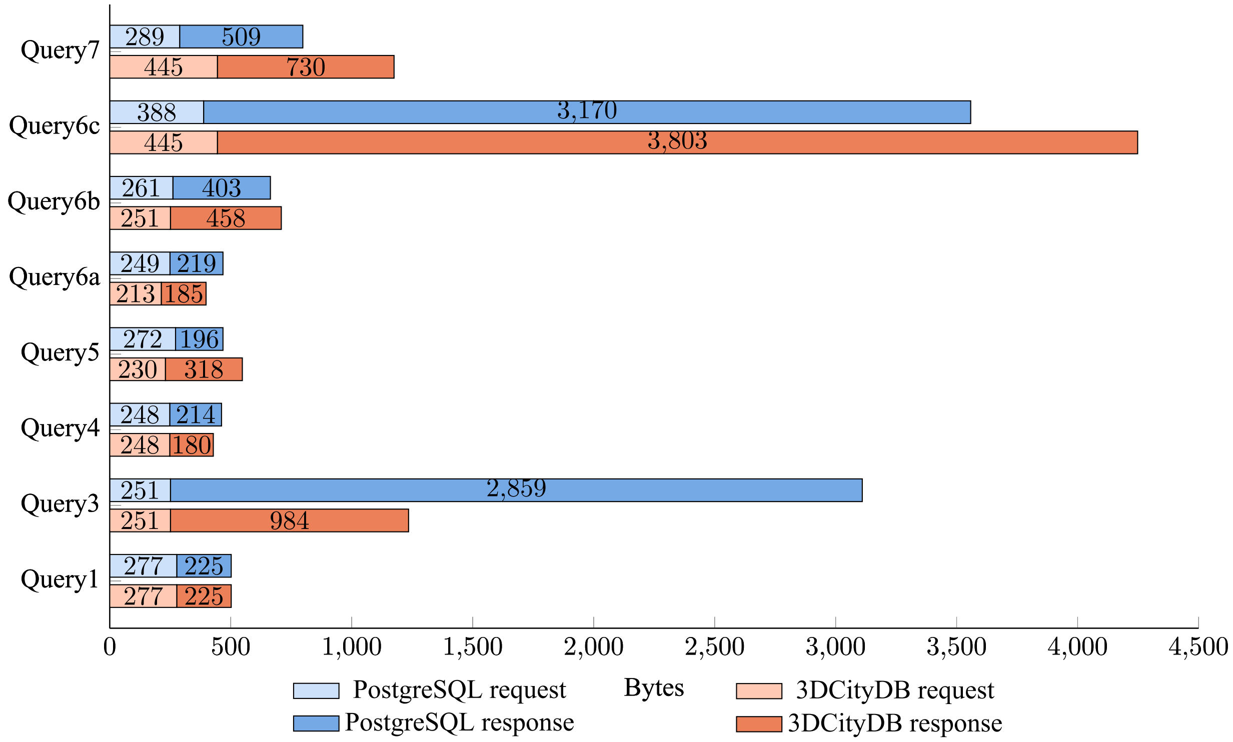


Figure 6.9: Comparison of the GraphQL request and response sizes between CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB

The HTTP response sizes are also compared in figure 6.9. The responses of query 3, 4 and 6a are larger for the implementation of CityJSON in PostgreSQL, because the one with 3DCityDB uses a sequence with shorter IDs for the city objects. The responses of the other queries are smaller for the implementation of CityJSON in PostgreSQL. Query 5 only returns one geometry ID for CityJSON in PostgreSQL, but the one with 3DCityDB returns multiple surface IDs. Query 6b returns a larger ID for the implementation of CityJSON in PostgreSQL, but the one with 3DCityDB uses two fields to return one attribute instead of one JSONB field for all attributes. The response size of query 6c is larger with 3DCityDB, because the surfaces return an extra field named children. Query 7 is also larger for the 3DCityDB implementation, because of two reasons. The first reason is that the 3DCityDB implementation returns two surfaces: one root surface, which relates to a thematic surface, and one child surface, which contains the `PolygonZ` geometry. The other reason is that the PostgreSQL implementation returns the semantic surface as JSONB field, while the 3DCityDB implementation has to request the `objectclassType` to get the name of the thematic surface and the `cityobject_genericattribType` to obtain the attributes of the thematic surface.

The percentage of data that the implementation with 3DCityDB returns redundantly to the API layer is more for each query than the one of CityJSON in PostgreSQL as can be seen in figure 6.10. The reason for this is that the implementation with 3DCityDB scatters the data model over more tables, it uses more columns and a parent/child structure as described in section 6.4.2.

6.4.5 Comparison between CityJSON in PostgreSQL and in MongoDB

The HTTP request sizes are compared in figure 6.11. The request sizes of query 1, 2, 3, 4 and 6c are approximately the same. The request of query 6b is larger for the MongoDB implementation, because PostgresQL can return the data in one field as JSONB data, while MongoDB has to specify the attributes individually. The requests of query 5

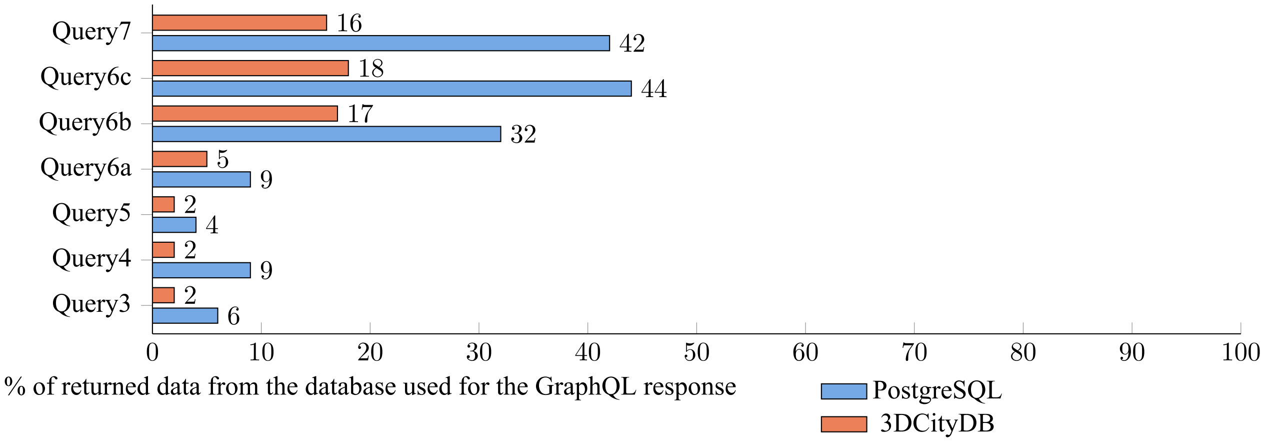


Figure 6.10: Comparison of the percentage of data used for the GraphQL response between CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB

and 6c are larger for the PostgreSQL implementation, because of two reasons. The first one is that the MongoDB implementation queries the geometry directly with the `MaxLoDQuery` and the PostgreSQL implementation queries the geometry as `maxlod` field indirectly through the `city_objectType`. The other reason is that the geometries are differently implemented and therefore different fields are queried. The embedded documents of the MongoDB implementation do not automatically have their own IDs, while tables must have their own IDs in PostgreSQL. These IDs are also specified when the geometry is queried.

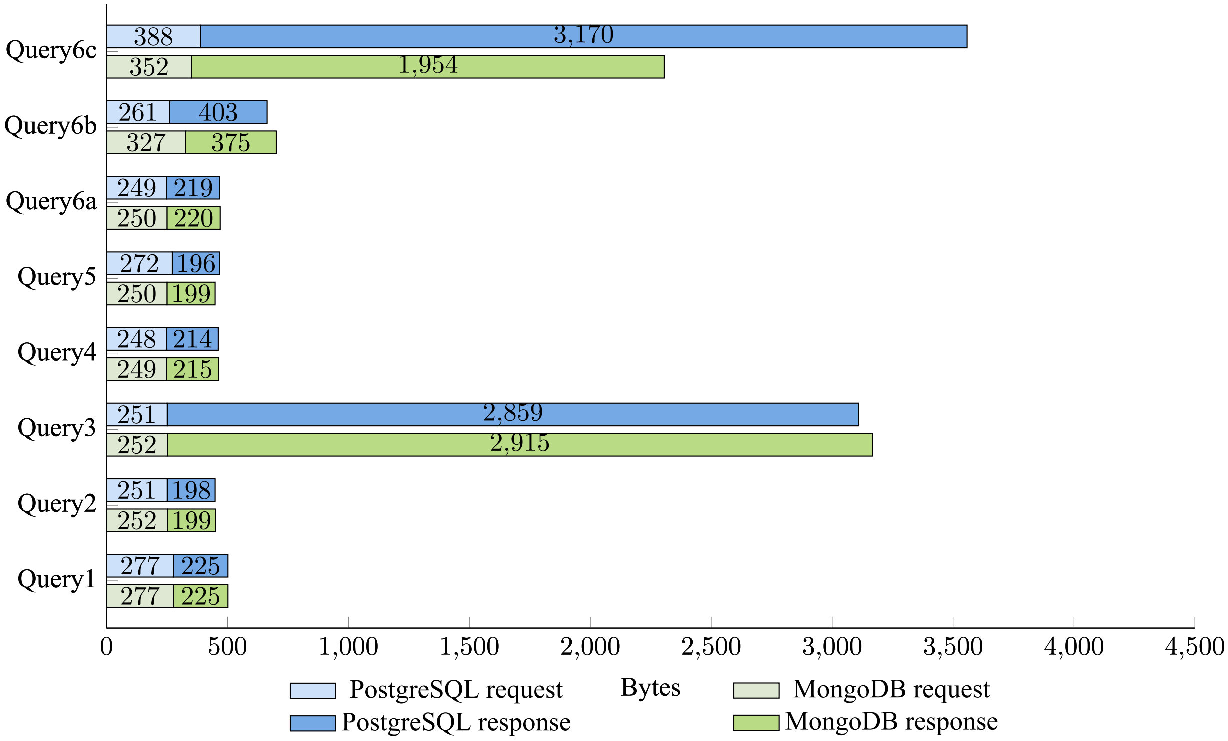


Figure 6.11: Comparison of the GraphQL request and response sizes between CityJSON in PostgreSQL and in MongoDB

The HTTP response sizes are also compared. The response of query 3 is slightly larger for the MongoDB implementation, because it returns the ID of the city objects with the `_id` field instead of the `id` field. The responses of query 6b and 6c are larger for the PostgreSQL implementation, because PostgreSQL returns the geometry as individual surfaces instead of an hierarchy of arrays and the attributes are returned as JSONB data instead of individual attribute fields. Individual attribute fields use a few bytes less.

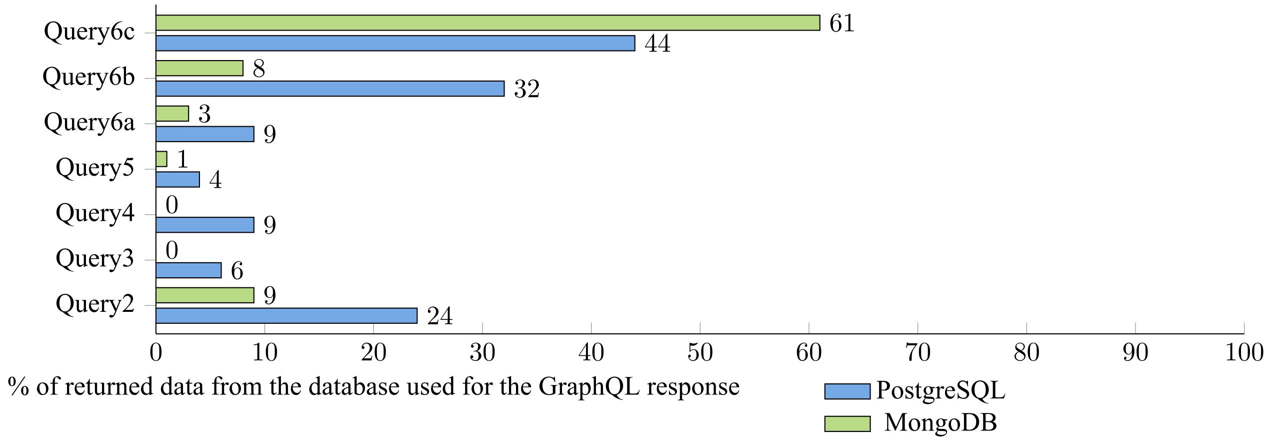


Figure 6.12: Comparison of the percentage of data used for the GraphQL response between the implementations of PostgreSQL and MongoDB

The PostgreSQL implementation uses almost always an higher percentage of the data than the MongoDB implementation as can be seen in figure 6.12, except for query 6c. The MongoDB implementation always queries the whole city object with its embedded documents such as the attributes and the geometries. A lot of data is therefore redundantly returned. Especially in the case that the geometry is not queried. The geometry is only queried with query 6c. The MongoDB implementation uses for query 6c even more data for the GraphQL response than the PostgreSQL implementation. The reason for this is that the PostgreSQL implementation returns some of the foreign keys redundantly and the geometries of the surfaces are queried twice: ones in EWKT which is the form in which the geometry is stored and ones as text to become readable for the client.

6.5 Retrieval times

The results are based on the experiments explained in section 5.6.1. The implementation of CityJSON in PostgreSQL divides query 3 and 4 into A and B, because the queries are implemented in two ways. Query A transforms the `convexhull` of the city object in the SQL query and query B uses a `globalconvexhull`, which is transformed before the `globalconvexhull` is used. Query 3B and query 4B are used for the comparisons with 3DCityDB and MongoDB, but not necessarily with an index on the `globalconvexhull`.

6.5.1 One and three CityJSON files without additional indexes in PostgreSQL

The database with the three city models contains more data than the database with one city model. A comparison between them is made, because the amount of data in the database influences the performance of the database. Query 2 is only performed on three city models and therefore excluded. The retrieval times of expensive queries are influenced the most according to figure 6.13. This is the case for query 3A, 3B (and therefore 4B), 4A and 6C.

The reason is that they are not using an efficient index. The city models without indexes only contain the automatic B-Tree indexes on the primary keys. They might therefore benefit from an additional index. Query 3A, 3B, 4A and 4B execute spatial operations on the `convexhull` and the `globalconvexhull` of the city objects. Query 6C queries the surfaces that relate to a semantic surface based on the `semantic_surface_id`. Additional indexes on the `convexhull`, the `globalconvexhull` and the `semantic_surface_id` might reduce the retrieval times.

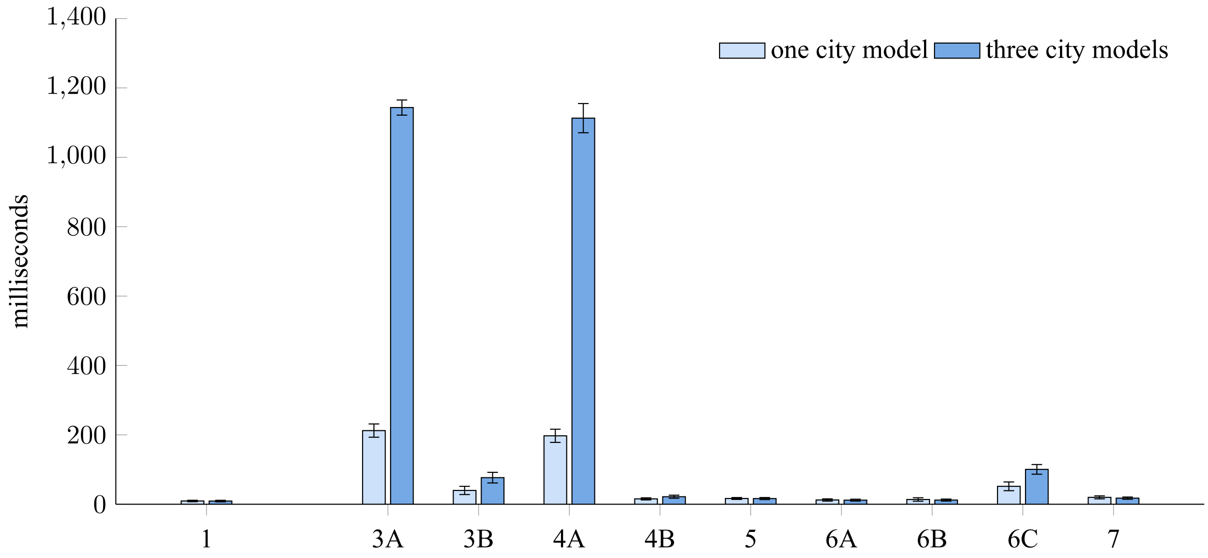


Figure 6.13: The GraphQL retrieval times with one and thee CityJSON files without additional indexes in PostgreSQL for each query

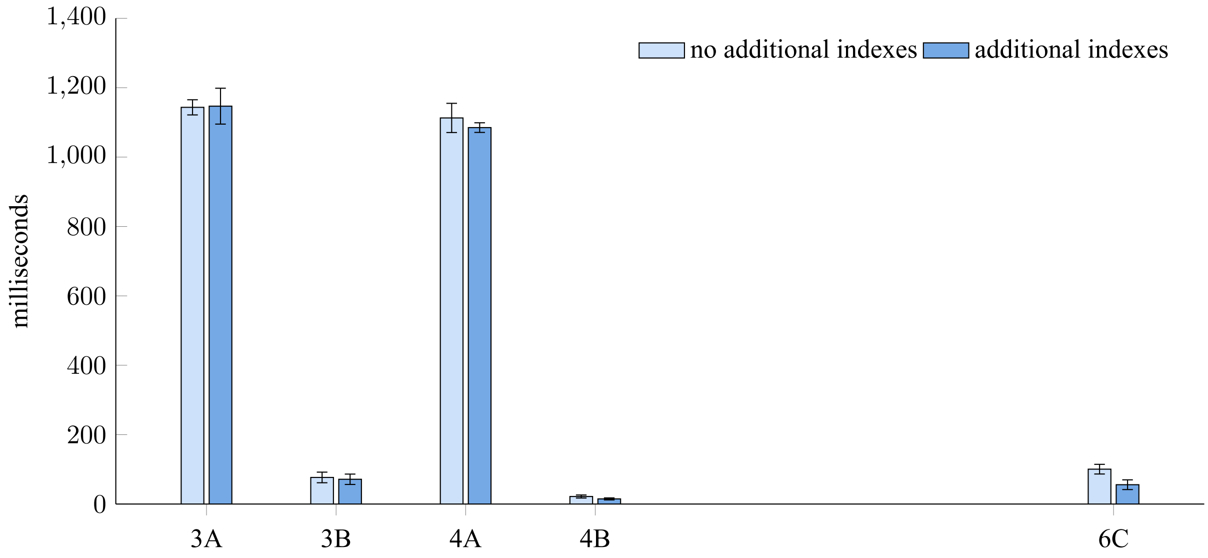


Figure 6.14: The GraphQL retrieval times of three city models with and three without additional indexes in PostgreSQL for each query

6.5.2 Three CityJSON files with and three without additional indexes in PostgreSQL

The additional indexes are added on the `convexhull`, the `globalconvexhull` and the `semantic_surface_id`. The indexes on the `convexhull` and the `globalconvexhull` are only working for query 4B and 6C as can be seen in the query plans of appendix G. It can also be seen in figure 6.14, because the retrieval times of query 6C are reduced with 50 % and the retrieval times of query 4B were already relatively low. The indexes are not working on the other queries. query 3A, 3B and 4A. It might be that the used city models are not large enough for the query planner to use the indexes on those queries. Another reason might be that the spatial operation `ST_DWithin` needs another index.

6.5.3 One and three CityJSON files in PostgreSQL with additional indexes

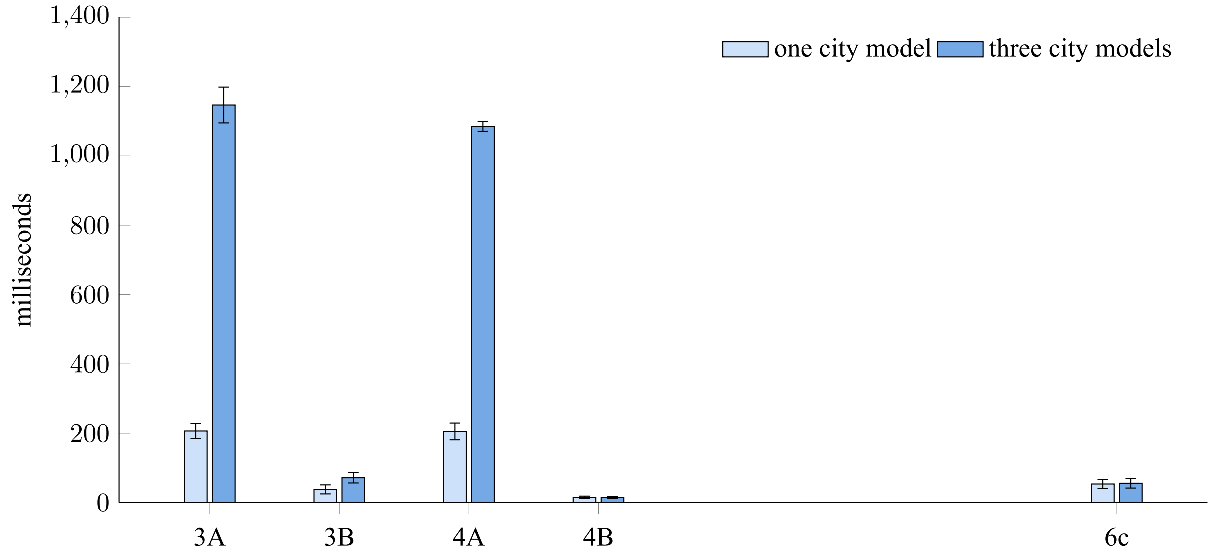


Figure 6.15: The GraphQL retrieval times of one and thee city models with additional indexes for the queries with additional indexes

An index ensures that the data volume in the database influences the retrieval times less, because indexes are implemented to efficiently search the data in the database. This happens to query 4B and 6C, because the data volume increases, but the retrieval times only slightly increase. This is an indication that the indexes are being used and they are used as can be seen in appendix G.

6.5.4 One CityJSON file in PostgreSQL with additional indexes and one CityGML file in PostgreSQL with 3DCityDB

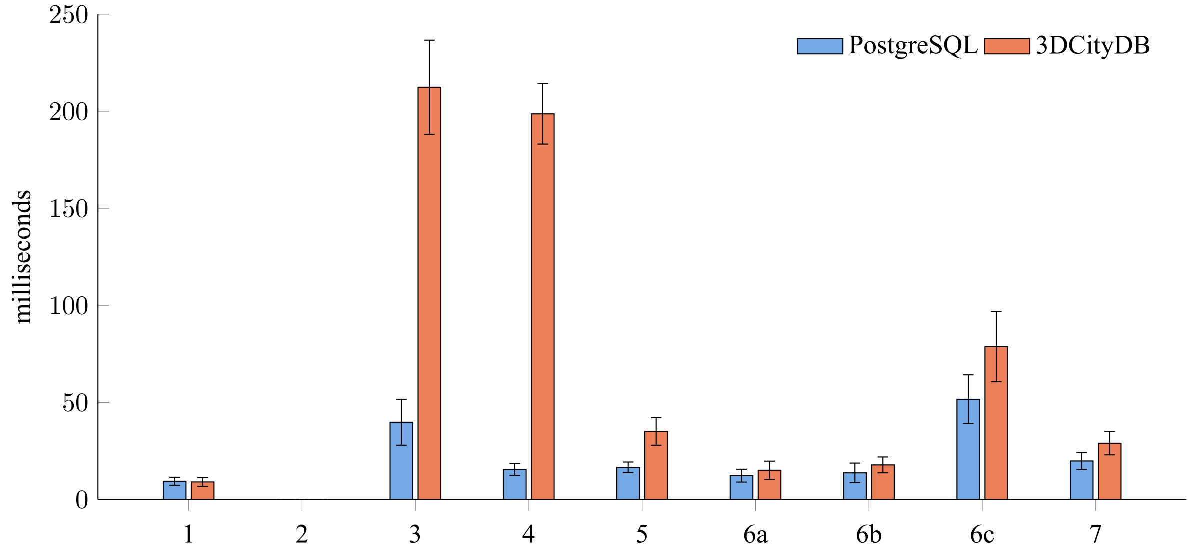


Figure 6.16: Comparison of the GraphQL retrieval times between CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB per defined query

The implementation of CityGML with 3DCityDB and the one of CityJSON in PostgreSQL contain additional indexes as can be seen in appendix B and appendix C. One city model is stored for each query as explained in section 5.6. Query 2 is therefore not compared.

The implementation with 3DCityDB does not efficiently implement query 3 and 4. The reason is likely the same as the reason for query 3A and 4A of CityJSON in PostgreSQL as described in section 6.5.2. The index is created on the `envelope` with the reference system of the city model.

The retrieval times of the other queries are lower for CityJSON in PostgreSQL as well. This might be due to the number of SQL queries to the database. The number of SQL queries is especially higher for the implementation with 3DCityDB in case of query 5, 6c and 7 as can be seen in figure 6.1. Small fluctuations might be due to the transferred data between GraphQL and the database, but they might also be due to the network bandwidth at a certain time.

6.5.5 Three CityJSON files in PostgreSQL and MongoDB without additional indexes

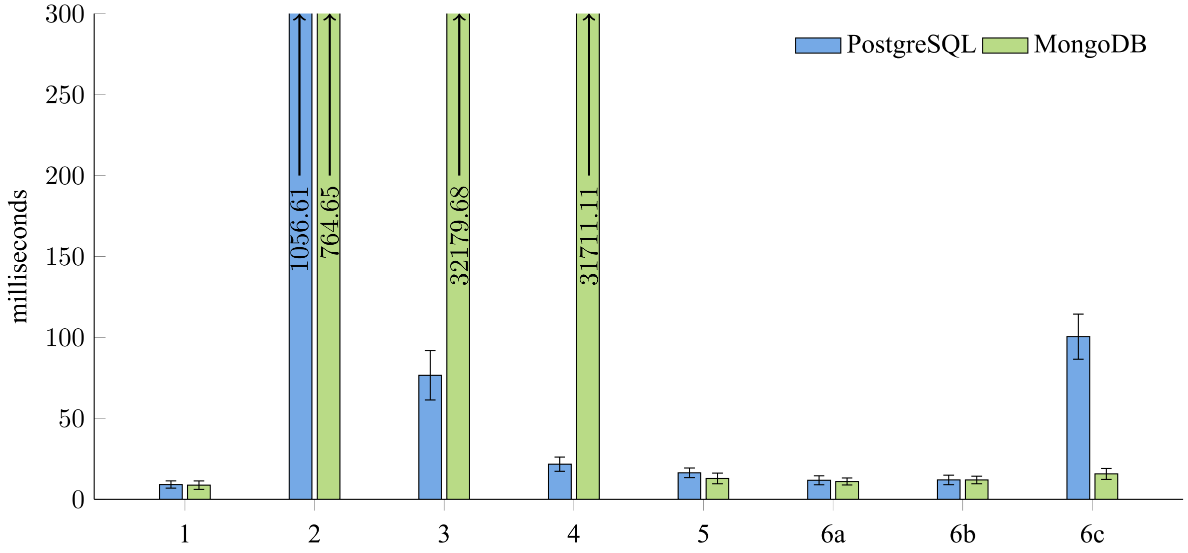


Figure 6.17: Comparison of the GraphQL retrieval times between PostgreSQL and MongoDB per defined query

The implementations do not contain additional indexes. Query 7 is not compared, because it was not possible to select the ID of a surface with the MongoDB implementation. Although the MongoDB database contains less geometries as described in section 5.5.6, the retrieval times of query 2, 3 and 4 are still too high for the implementation of CityJSON in MongoDB. Query 3 and 4 take even more than 31000 ms to execute, because the geometric representation of the city object, the `convexhull`, is created outside the database in the resolver and therefore no indexes are used. Although it might be possible to create an index in the resolver, this is not feasible based on the calculations that would have to be made each time. MongoDB only supports geometry indexes on GeoJSON objects as explained in 4.1.3. It is not feasible to perform spatial operations that require spatial indexes outside the database.

The MongoDB implementation is faster in case of query 5 and query 6c, because the number of queries is lower for the MongoDB implementation as could be seen in figure 6.2. Less processing is required, which results in faster retrieval times.

6.5.6 Three smaller and one large CityJSON file in PostgreSQL

This section is an additional section, which has not be used for the results of this thesis. The large dataset is retrieved from https://3d.bk.tudelft.nl/opendata/cityjson/1.0/Zurich_Building_LoD2_V10.json and it contains the city Zürich. The file contains 198699 city objects and is 292.8 MB, while the three smaller files contain together 6258 city objects and are together 19.7 MB.

Query 3B does not use the index, because the 100 meters have to be converted to degrees. 0.001 degrees is approximately 111 meters. In this case, the retrieval times will drop to the retrieval times of query 4B. The relatively high retrieval times of query 6C are due to the fact that there is no index on column `city_object_id` of the geometries table and no index on column `geometries_id` of the `semantic_surface` table. After adding these indexes and the usage of 0.001 degrees, the results of these adjustments can be seen in figure 6.18.

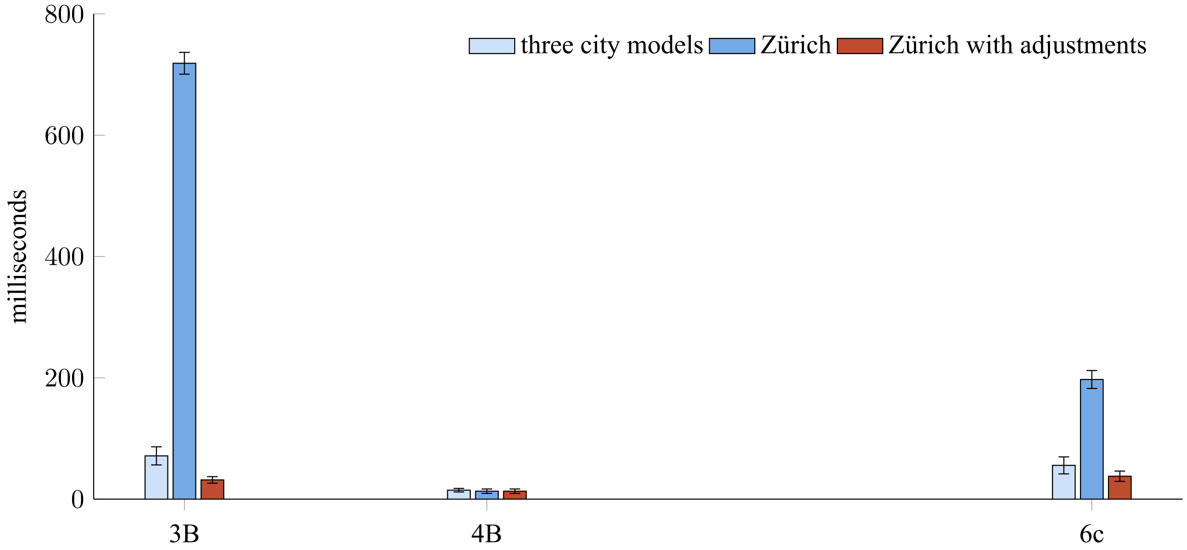


Figure 6.18: The GraphQL retrieval times thee city models and Zürich with additional indexes for the queries with additional indexes

7 Conclusions

7.1 Discussion

In the end, CityJSON has been mapped to the relational database PostgreSQL and the NoSQL database MongoDB. CityGML has also been mapped to PostgreSQL with 3DCityDB to clarify the impact of different exchange formats. The databases are after that accessed and queried through GraphQL. The queries are based on the selection process of an AR application. The architecture is tested based on the number of queries between the databases and GraphQL, the request sizes, the respond sizes and the retrieval times. Three relatively small datasets have been used to do the performance measurements. These datasets were chosen to test the mappings, because the datasets have different characteristics. However, they were less suitable for the performance measurements due to their relatively small sizes. Besides this, the queries were only tested with one argument and in a relatively unstable environment. It would have been better to test the queries with different arguments and in a more stable environment. This would provide more reliable results.

Both databases are able to store the data of CityJSON (except the attribute presentLoDs in MongoDB), but their suitability can be described based on three aspects that are discovered after trying different mappings. Additionally, the databases were accessed and queried through GraphQL. In the next paragraphs, the difficulties related to the implementation of GraphQL will also be described.

The first aspect is that the indexing mechanism and the hierarchical structure of the boundaries is not necessarily maintained in the databases, because the hierarchical structure is often converted to supported geometry types. Databases support spatial indexes on supported geometry types. However, the indexing mechanism has more chance to succeed in PostgreSQL than in MongoDB. The maximum BSON document size in MongoDB is 16 MB and the vertices list is almost always larger than that. The hierarchical structure can be implemented due to the integration of JSON in both databasess. GraphQL does on the other hand not support fields with varying data types. Solutions can be to parse a string or to support only one geometry type.

The second aspect is related to the structure of the databases. MongoDB is able to query the entire document based on the ID of the document, which can contain embedded documents. This means that a city object is queried with its geometries and attributes all together. On the contrary, PostgreSQL stores the geometries and attributes in separate tables. This means that a join between the tables is needed to retrieve the entire city object. However, the queries were asked through GraphQL. GraphQL sends SQL/MQL queries to the database to answer those queries. More specifically, the ORM or ODM between the database and GraphQL translates the GraphQL query in SQL/MQL queries. The software that carries out the translation does not often use joins, which means that the translation might not result in the most optimal SQL/MQL queries. The results show that the usage of less tables result in better retrieval times, because less queries are sent to the database. The least queries are sent to MongoDB, because the city object with its embedded documents is retrieved with one query. Multiple queries must therefore be sent to PostgreSQL. However, it is unknown whether the Python libraries are causing this issue or whether the GraphQL schema is incapable of sending efficient queries.

The third aspect is related to the spatial functionalities of the databases. PostgreSQL has more spatial functionalities than MongoDB. The main difference is that PostgreSQL supports reference transformations, while MongoDB only supports GeoJSON objects in WGS84. The defined use case did not necessarily require reference transformation, but it was not properly investigated which database performs better in a global reference system with simplified 2D representations of the city objects.

The mappings of CityJSON and CityGML are also compared to clarify the impact of the different exchange formats and to identify the suitability of CityJSON. 3DCityDB scatters CityGML over more tables than the mapping of CityJSON, because JSON is able to map normal and generic attributes without specifying them on forehand. In this way, all types of city objects can be easily mapped to the same table or collection. Both reasons result in a lower amount of tables. Consequently, it results in less joins and smaller respond and request sizes, because less tables have to be accessed as separate GraphQL object types. On the other hand, querying on a JSON attribute might result in higher retrieval times, but this is not tested in this research.

CityJSON can be used in combination with PostgreSQL, MongoDB and GraphQL. The suitability of these technologies depends on the use case, but the technologies are not tested on all aspects. It is therefore difficult to draw conclusions about which database is the most suitable for the defined use case based on this research.

7.2 Answer research questions

Q1. What are the differences between the storage of CityJSON in PostgreSQL and the storage of CityGML in PostgreSQL with 3DCityDB?

The differences related to CityGML and CityJSON are described in section 2.1.2 and 2.1.3. The differences related to the mappings are described in section 3 and section 4.1.2. The implementations are described in section 5.3.1 and section 5.4.1

CityJSON	**CityGML**
The geometry and semantic surface objects can only be stored in one way.	They can be stored in multiple ways.
The different types of city objects can be identified with the field `type`. This field specifies the sub-class.	The different types of city objects are inherited as sub-classes in the base-class.
Only one reference system for a city model is allowed.	Every city objects can have its own reference system. However, 3DCityDB does not allow it.
Semantic surface objects are not regarded as city objects, but as a dependent entity of the semantics.	Thematic surfaces are regarded as city objects
It is not allowed to share geometries	It is allowed to share geometries. However, 3DCityDB does not allow it due to the parent/child structure

Table 7.1: Differences between CityJSON and CityGML

While CityJSON uses JSON, CityGML uses XML. CityJSON also encodes the CityGML data model differently to increase the simplicity and efficiency of the data model. Other differences are explained in table 7.1. These differences influence the mappings of the exchange formats to PostgreSQL.

The base-class and sub-classes are mapped to separate tables with an inheritance relationship between them. On the other hand, the implementation of CityJSON maps them to one table and uses the attribute `type` to identify the sub-class. However, the attributes can differ per city object type. The implementation of CityJSON does not have to map all attributes to one table on forehand, because they can be mapped flexibly to one JSONB column. On the contrary, the attributes are mapped to separate columns with 3DCityDB to specify each attribute with its data type on forehand. Consequently, the concept of generic attributes is mapped to a separate table as described in section 3.

Secondly, the geometries are mapped differently. The column names are based on the `type` and LoD of the geometry with 3DCityDB. This results in many columns that can contain the geometry. CityJSON provides information about the `type` and LoD of the geometry through two separate attributes: one contains the `type` and one contains the LoD, and the corresponding surfaces contain a link to the city object. This makes it together with the fact that CityJSON only represents the geometries and semantics in one way easier to understand the mapping of the geometries. Additionally, semantic surface objects are not regarded as city objects as described in table 7.1.

Q2. What are the differences between the storage of CityJSON in PostgreSQL and in MongoDB?

An entity-relationship analysis is performed on the data and relationships of CityJSON as described in section 4.1.1.The differences related to the mappings are described in section 4.1.2 and section 4.1.3. The implementations are described in section 5.3.1 and section 5.5.1.

Most data can be stored in PostgreSQL and MongoDB, except the data and relationships of table 7.2. The spatial functionalities differ per table as can be seen in table 7.3. PostgreSQL has more spatial functionalities than MongoDB, but not all geometry definitions of the CityGML data model are fully supported. Voids are for instance not supported and surfaces are often stored individually to maintain the link with the semantic surface objects.

PostgreSQL	**MongoDB**
The variable name `geometry` has to be changed to `geometries`, because it is reserved for the geometry data type.	The variable name `geometry` does not have to be changed to `geometries`, because it uses `$geometry` as data type.
The attribute `presendLoDs` can not be stored as described in section 5.5.1.	The attribute `presendLoDs` can be stored.
Although the indexing mechanism is not implemented in this research, the vertices list can be stored for most city models. The maximum field size is 1 GB.	The indexing mechanism is not maintained, because the vertices list exceeds the maximum BSON document size of 16 MB in most cases.

Table 7.2: The data and relationships of CityJSON that can not be stored in PostgreSQL and/or MongoDB

PostgreSQL	**MongoDB**
Support for spatial reference transformations	No support for spatial reference transformations
Support for spatial operations and geometry types with geometrical coordinates in 2D and 3D	No support for spatial operations and geometry types with geometrical coordinates in 2D and 3D
Support for spatial operations and geometry types with geographical coordinates in 2D	Support for spatial operations and geometry types with geographical coordinates (GeoJSON objects) in 2D
Support for indexes on geometry types with geometrical and geographical coordinates in 2D and 3D	Support for indexes on geometry types with geographical coordinates (GeoJSON objects) in 2D

Table 7.3: The spatial functionalities of PostgreSQL and MongoDB

The metadata object, the city objects and the transform object are considered independent entities in this research, and the attributes, geometric objects, the semantics object and the semantic surface objects are considered dependent entities. PostgreSQL maps the independent entities and most of the dependent entities to different tables, but MongoDB only maps the independent entities to different collections. The dependent entities are mapped as embeddded documents. Embedded documents are one of the options to refer documents to each other in MongoDB. The tables in PostgreSQL are referenced to each other with the usage of primary and foreign keys. Although the databases handle the relationships between the entities differently, both databases integrated JSON. The attributes are for instance mapped differently, because PostgreSQL maps them to one JSONB column and MongoDB as an embedded BSON document.

Q3. What are the differences between accessing and querying CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB using GraphQL?

The implementations to access and query the databases are described in section 5.3.6, 5.3.7, 5.4.5 and 5.4.6.

The different types of city objects are stored differently and therefore accessed and queried differently. As a consequence, the filter on the type of city object is implemented differently for each exchange format. The implementation of CityGML filters on the column `objectclass_id` and the implementation of CityJSON filters on the column `type`. However, the implementation with CityGML has to define the sub-class as GraphQL object type in the GraphQL query in order to access the attributes of the sub-class.

Also, the mapping of the geometries is easier to understand for the implementation of CityJSON in PostgreSQL, which makes the implementation of the queries easier as well. Besides this, the generic attributes were mapped to a separate column with 3DCityDB. They had to be resolved in order to return them in approximately the same way as normal attributes. The implementation of CityJSON did not have this problem, because CityJSON regards generic attributes as normal attributes.

Q4. How do these differences influence the performance of PostgreSQL when querying CityJSON and CityGML with 3DCityDB using GraphQL?

The results between CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB can be seen in section 6.3.1, 6.4.4 and 6.5.4.

The results indicate that the implementation of CityGML in PostgreSQL with 3DCityDB uses more SQL queries compared to the implementation of CityJSON in PostgreSQL. The first reason is that the data model is scattered over more tables due to the inheritance relationship, the generic attributes and the class names are mapped to a separate table. The other reason is more due to the implementation, but also due to the fact that the semantics and geometries can be implemented in multiple ways. This makes it more difficult for the developer to implement the queries and therefore more tables and columns are requested. Less SQL queries result in lower retrieval times for the implementation of CityJSON in PostgreSQL.

The scattered data model can result in larger response and request sizes, because more tables have to be accessed as separate GraphQL object types. The implementation of CityJSON uses relatively more data from the database than the implementation of CityGML. This is also due to the fact that the data model is scatted over more tables and it uses more columns to specify the attributes.

Q5. What are the differences between accessing and querying CityJSON in MongoDB and PostgreSQL using GraphQL?

The methodology to access the database is described in section 4.2.1. The implementations to access and query the databases are described in section 5.3.6, 5.3.7, 5.5.6 and 5.5.7.

Both databases integrated JSON. JSON allows fields with varying data types. GraphQL does on the other hand not support fields with varying data types. Solutions can be to parse a string or to support only one geometry type.

Although it is dependent on the implementation, it must be mentioned that the variable name `type` can not be used as argument in the GraphQL schema when querying, because it is a built-in method in the programming language Python.

CityJSON is stored slightly different in MongoDB than in PostgreSQL and they are therefore also accessed and queried differently. The MQL/SQL queries can use the spatial functionalities of table 7.3. If the database does not have the required spatial functionalities, the spatial functionalities have to be resolved outside the database.

Furthermore, the databases query an entire city object differently. While a MQL query retrieves the entire city object with its embedded documents, a SQL query has to query each table that is related to the city object separately or it has to use a join between them.

Q6. How do these differences influence the performance of CityJSON in MongoDB and PostgreSQL when querying CityJSON using GraphQL?

The results between CityJSON in PostgreSQL and CityGML in PostgreSQL with 3DCityDB can be seen in section 6.3.2, 6.4.5 and 6.5.5.

The implementation of CityJSON in MongoDB uses less queries than the implementation of CityJSON in PostgreSQL, because the implementation does not have to query separate tables. Consequently, this will result in better retrieval times for the implementation in MongoDB. However, it will also result in relatively less data that is used from the database for the GraphQL response when only a few fields of the document are requested by the client.

On the other hand, PostgreSQL supports more geometry types than MongoDB. This can improve the performance, because the databases support spatial indexes and spatial operations on the supported geometry types. MongoDB only supports geometry types with geographical coordinates in 2D, but PostgreSQL also supports geometry types with geometrical coordinates in 2D and 3D.

How suitable are MongoDB and PostgreSQL for the storage and querying of CityJSON using GraphQL?

The main difference between CityJSON and CityGML is the usage of JSON. MongoDB and PostgreSQL integrated JSON. The results show that the usage of JSON maps attributes more flexibly than the mapping of 3DCityDB, which can result in less tables/ collections and therefore less joins or queries. On the other hand, querying on a JSON attribute might result in higher retrieval times, but this is not investigated. Additionally, the usage of JSON makes it possible to store field with varying data types such as the hierarchy of arrays. A difficulty can be that software such as GraphQL is less flexible.

In general, there are no real signs yet that MongoDB and PostgreSQL are not suitable for the storage and querying of CityJSON using GraphQL. Possible signs are that the indexing mechanism with the vertices list can not be stored in MongoDB and only to some extent in PostgresQL, but this might not be a problem since the indexes can be resolved to integer or real coordinates. The vertices list easily exceeds the maximum document size of MongoDB, which is 16 MB. PostgreSQL is on the other hand able to store a vertices list up to 1 GB. The attribute `presentLoDs` can not be stored in MongoDB as well, but this can probably be solved with a small adjustment.

Although PostgreSQL supports more spatial functionalities than MongoDB, it remains difficult just as with CityGML to map the geometries and semantics to the database. When spatial queries require spatial indexes, the geometries should be mapped to the database. The indexing mechanism and the hierarchical structure of CityJSON provide no solution for this.

7.3 Issues related to the implementations

The methodology and the implementation have influenced the results of this thesis in multiple ways. The issues are described according to different parts of the implementation.

Mapping of the IDs

The implementations have influenced the mapping of the IDs in two ways. The mapping between CityGML with 3DCityDB and CityJSON use different kind of IDs for the city objects. The implementation of CityGML uses a sequence and the implementation of CityJSON uses the original IDs. The usage of sequence IDs reduces the request and response sizes. Nevertheless, it is a design choice independent of the used exchange format.

The implementation of CityJSON and MongoDB also differ, because embedded documents in MongoDB do not automatically have their own ID. Although it might be possible to add them, the mapping of this research did not add them. As a result, query 7 has not been implemented in MongoDB.

Mapping of the geometry

The implementations have influenced the mapping of the geometry in many ways. The first reason is related to the number of queried city models. The storage of multiple city models in one database has the advantage that they can be queried together, but the disadvantage is that the implemented queries might be less efficient. The implementations focus on these different aspects. While the implementation of CityJSON in PostgreSQL focuses on the advantage of multiple city models in one database, the implementation of CityGML in PostgreSQL with 3DCityDB focuses on the disadvantage and stores therefore one city model in one database. The usage of multiple city models is also influenced by the usage of the Python library `graphene`. Although GraphQL allows schema stitching, the Python library does not. The number of city models influences query 2, 3 and 4.

The number of city models influences query 3 and 4, because spatial operations can only be performed on geometries with the same reference system. The usage of multiple city models requires that the city objects are transformed to one global reference system. This is done and based on the idea that the database has to contain multiple city models with different reference systems.

However, this might not be the fastest and most accurate solution. It might be better to use one city model in one database. No global reference system would be needed anymore. Geometrical coordinates could be used. Since it is not possible to perform 3D operations on geographical coordinates without a transformation, 3D operations are only possible with geometrical coordinates. This means that the queries could also be implemented for one city model in one database, but the difficulty will be to establish the connection with the right database or database schema through GraphQL based on the user's location.

Furthermore, the implementations of CityJSON and CityGML in PostgreSQL use different simplified representations of the geometry. While the implementation with 3DCityDB uses the `envelope` of a city object, the implementation of CityJSON uses the `convexhull`. The usage of the `convexhull` increases the retrieval times, because it represents the surface of the city object more accurately. The design choice is however independent of the used exchange format.

Also, the implementation of CityJSON in MongoDB uses a different representation. The hierarchical structure of arrays is stored in the database, but the simplified representation is created outside the database. The underlying idea was that the geometry definitions of the CityGML data model can not be stored as GeoJSON objects as described in section 4.1.3. However, it would be possible to store the simplified representation as a GeoJSON object, but this has not been done.

Lastly, the indexes are resolved to coordinates and the integer coordinates are transformed to real coordinates for CityJSON in MongoDB and in PostgreSQL. MongoDB is not able to store the indexing mechanism of CityJSON. PostgreSQL is more suitable, because the maximum field size is 1 GB instead of 16 MB. This means that PostgreSQL can store a larger vertices list in one field. Nevertheless, the geometry is stored as individual surfaces and not as an hierarchy of arrays. The indexing mechanism is not used, because the real coordinates are used to create the individual surfaces with supported geometry types in PostgreSQL.

Mapping of relationships

The implementations have influenced the mapping of the relationships in at least one way. The mapping of CityJSON to PostgreSQL regards dependent entities as independent entities with an associative relationship, while they should be mapped as dependent entities like they are mapped in MongoDB. This does not mean that it is not possible to map them as dependent entities in PostgreSQL, but it is not investigated to map them as dependent entities due to the methodology. The implementation of CityGML in PostgreSQL with 3DCityDB maps composite and aggregate relationships.

GraphQL

The implementations have influenced the mapping of the database schemas to GraphQL in two ways. The implementations of CityJSON in MongoDB and in PostgreSQL use different Python libraries. These libraries cause the queries to be differently resolved, because the resolver of the `graphene_sqlalchemy` library queries the objects after the SQL query is returned and resolver of the `graphene_mongo` library queries the objects before they are returned.

Additionally, the implementation of CityJSON in MongoDB only stores `MultiSurfaces`, because GraphQL is not able to handle fields with varying data types. The same would happen to the implementations in PostgreSQL when the geometry would be stored as an hierarchy of arrays in a JSONB column. The problem of fields with varying data types is therefore not related to MongoDB, but to the usage of JSON in combination with GraphQL.

7.4 Recommendations

This section includes general recommendations, recommendations that are only applicable to the defined use case and recommendations that are applicable to other use cases.

General

At first, the SQL/MQL queries should have been implemented as efficient as possible without being connected to GraphQL. This should have been done to understand the performance of the databases and to understand the impact of GraphQL. This has not been done in this research. The retrieval times of this research must therefore be interpreted with suspicion. Additionally, the queries are implemented based on the abilities of GraphQL. As a result, it might be that the limitations of GraphQL did not always emerge.

This did not only influence the retrieval times, but it also influenced over-fetching from the database. Over-fetching from the database might be due to poorly designed resolved functions with inefficient SQL/MQL queries in terms of selecting fields through GraphQL.

It would also be better to test the retrieval times based on more ID-based arguments across the database, because of the indexes and the used query plans. Additionally, the implementations should have been tested with larger datasets. A quick test with larger datasets has been performed in section 6.5.6, which shows that the indexes are not implemented on the right columns and units.

Furthermore, CityJSON can be used in combination with PostgreSQL, MongoDB and GraphQL, but the technologies are not tested on all aspects. The technologies have not been tested on the amount of users, the amount of data, transaction reliability etc. This could for instance describe the suitability of the database to handle large amounts of data.

Applicable to the defined use case

The comparison between CityJSON in MongoDB and CityJSON in PostgreSQL is not optimally investigated according to three aspects.

At first, tables in PostgreSQL have automatically their own IDs, but this is not automatically the case for embedded documents in MongoDB. No IDs are added to the embedded documents. This makes it impossible to select a surface in MongoDB and therefore to compare the queries that select a surface with an ID-based reference to the database.

Secondly, no simplified representations of the city objects are added as GeoJSON objects in MongoDB. This makes it impossible to compare the spatial queries fairly. I would recommend to do this in order to compare spatial queries with geographical coordinates in 2D between MongoDB and PostgreSQL in terms of retrieval times .

Thirdly, the dependent entities have been mapped as independent entities in PostgreSQL. However, they should be mapped as dependent entities that can not exist on their own. MongoDB has mapped dependent entities as embedded documents, which can not exist on their own. In this way, it would be possible to compare the possibilities to implement the relationships fairly.

Applicable to other use cases

The use case did not force to implement more types of queries such as JSON-based filters and aggregate queries.

The differences between querying on a column in PostgreSQL, on an attributes of a JSONB column in PostgreSQL or on an attribute field in MongoDB are not investigated in terms of retrieval times. Although the usage of JSON can result in less tables and therefore in faster retrieval times for ID-based queries, it is unknown whether the usage of JSON influences the efficiency to query the attributes.

Aggregate queries are not implemented, because the queries are implemented in the most straightforward way. GraphQL might not be suitable for aggregate queries, but this is not properly investigated. Further research can therefore investigate how suitable and efficient in terms of retrieval times MongoDB, PostgreSQL and GraphQL are to handle aggregate queries.

The defines use case does not operate on the geometry definitions of the CityGML data model, but on simplified representations of them. As a consequence, the implementation of the geometry definitions of the CityGML data model are not tested. Other use cases could be used to test whether it is possible to visualize them easily or whether it is possible to perform spatial operations on them in the database.

In summary, a more general understanding of the suitability for all use cases could be provided with a framework that tests more types of queries.

A Reproducibility self-assessment

The Python scripts can be downloaded from the GitHub repository at `https://github.com/kjstaring/scripts`. The data is not present in the repository and should be downloaded from the websites of section 5.1. The Python scripts must be adjusted to open the downloaded files and to store the created files. The experiments are performed manually for each query and each implementation with `JMeter`. It would have been better for the reproducibility of this thesis to automate this part, because of the logfiles, the commands in the terminal and the manual editing in Excel. The `JMeter` files with the setting and queries have been stored in the folder results.

B PostgreSQL database schema

The PostgreSQL database schema exists of the tables in figure B.1, B.2, B.3, B.4, B.5, B.6 and B.7.

```
 Column  | Type   | Nullable | Default
---------+--------+----------+---------
 id      | text   | not null |
 object  | jsonb  |          |
Indexes: ”metadata_pkey” PRIMARY KEY, btree (id)
```

Figure B.1: `metadata` table description

```
 Column  | Type   | Nullable | Default
---------+--------+----------+---------
 id      | text   | not null |
 object  | jsonb  |          |
Indexes: ”transform_pkey” PRIMARY KEY, btree (id)
```

Figure B.2: `transform` table description

```
    Column       |         Type          | Nullable | Default
-----------------+-----------------------+----------+---------
 id              | text                  | not null |
 object          | jsonb                 |          |
 attributes      | jsonb                 |          |
 convexhull      | geometry(Polygon)     |          |
 globalconvexhull|                       |          |
 metadata_id     | text                  |          |
Indexes:
”city_object_pkey” PRIMARY KEY, btree (id)
”convexhull_index” gist (convexhull)
”globalconvexhull_index” gist (globalconvexhull)
Foreign-key constraints:
”city_object_metadata_id_fkey” FOREIGN KEY (metadata_id)
REFERENCES metadata(id)
```

Figure B.3: `city_object` table description

```
    Column     | Type | Nullable | Default
---------------+------+----------+---------
 parents_id    | text | not null |
 children_id   | text | not null |
Indexes: ”parents_children_pkey”
PRIMARY KEY, btree (parents_id , children_id)
```

Figure B.4: `parents_children` table description

```
     Column      |  Type    | Nullable |          Default
-----------------+----------+----------+----------------------------------
 id              | integer  | not null | nextval(’geometry_seq ’)
 object          | jsonb    |          |
 city_object_id  | text     |          |
Indexes: ”geometries_pkey” PRIMARY KEY, btree (id)
Foreign-key constraints:
”geometries_city_object_id_fkey” FOREIGN KEY (city_object_id)
REFERENCES city_object(id)
```

Figure B.5: `geometries` table description

```
       Column          |        Type          |Nullable|        Default
-----------------------+----------------------+--------+-------------------------
 id                    |integer               |not null| nextval(’surfaces_seq ’)
 geometry              |geometry(PolygonZ)    |        |
 solid_num             |integer               |        |
 shell_num_void        |integer               |        |
 surface_num           |integer               |        |
 geometries_id         |integer               |        |
 semantic_surface_id   |integer               |        |
 city_object_id        |text                  |        |
Indexes:
”surfaces_pkey” PRIMARY KEY, btree (id)
”geometry_index” gist (geometry)
”semantic_surface_index” btree (semantic_surface_id)
Foreign-key constraints:
”surfaces_city_object_id_fkey” FOREIGN KEY (city_object_id)
REFERENCES city_object(id)
”surfaces_geometries_id_fkey” FOREIGN KEY (geometries_id)
REFERENCES geometries(id)
”surfaces_semantic_surface_id_fkey” FOREIGN KEY (semantic_surface_id)
REFERENCES semantic_surface(id)
```

Figure B.6: `surfaces` table description

```
     Column      |  Type   | Nullable |               Default
-----------------+---------+----------+--------------------------------------
 id              | integer | not null | nextval('semantic_surface_seq')
 object          | jsonb   |          |
 city_object_id  | text    |          |
 geometries_id   | integer |          |
Indexes:
"semantic_surface_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
"semantic_surface_city_object_id_fkey" FOREIGN KEY (city_object_id)
REFERENCES cityjsondb.city_object(id)
"semantic_surface_geometries_id_fkey" FOREIGN KEY (geometries_id)
REFERENCES cityjsondb.geometries(id)
```

Figure B.7: semantic_surface description table

C 3DCityDB database schema

The 3DCityDB database schema exists of the tables described in section ??.

```
 Column                 |          Type            | Nullable |  Default
------------------------+--------------------------+----------+------------------
 id                     | integer                  | not null | sequence
 objectclass_id         | integer                  | not null |
 gmlid                  | character varying(256)   |          |
 gmlid_codespace        | character varying(1000)  |          |
 name                   | character varying(1000)  |          |
 name_codespace         | character varying(4000)  |          |
 description            | character varying(4000)  |          |
 envelope               | geometry(PolygonZ,7415)  |          |
 creation_date          | timestamp with time zone |          |
 termination_date       | timestamp with time zone |          |
 relative_to_terrain    | character varying(256)   |          |
 relative_to_water      | character varying(256)   |          |
 last_modification_date | timestamp with time zone |          |
 updating_person        | character varying(256)   |          |
 reason_for_update      | character varying(4000)  |          |
 lineage                | character varying(256)   |          |
 xml_source             | text                     |          |
    Indexes:
"cityobject_pk" PRIMARY KEY, btree (id) WITH (fillfactor='100')
"cityobject_envelope_spx" gist (envelope)
"cityobject_inx" btree (gmlid, gmlid_codespace) WITH (fillfactor='90')
"cityobject_lineage_inx" btree (lineage)
"cityobject_objectclass_fkx" btree (objectclass_id) WITH (fillfactor='90')
    Foreign-key constraints:
"cityobject_objectclass_fk" FOREIGN KEY (objectclass_id)
REFERENCES objectclass(id) MATCH FULL ON UPDATE CASCADE
```

Figure C.1: `cityobject` table description

```
      Column        |                  Type                    | Nullable | Default
--------------------+------------------------------------------+----------+-----------
 id                 | integer                                  | not null | nextval()
 gmlid              | character varying(256)                   |          |
 gmlid_codespace    | character varying(1000)                  |          |
 parent_id          | integer                                  |          |
 root_id            | integer                                  |          |
 is_solid           | numeric                                  |          |
 is_composite       | numeric                                  |          |
 is_triangulated    | numeric                                  |          |
 is_xlink           | numeric                                  |          |
 is_reverse         | numeric                                  |          |
 solid_geometry     | geometry(PolyhedralSurfaceZ,7415)        |          |
 geometry           | geometry(PolygonZ,7415)                  |          |
 implicit_geometry  | geometry(PolygonZ)                       |          |
 cityobject_id      | integer                                  |          |
    Indexes:
”surface_geometry_pk” PRIMARY KEY, btree (id) WITH (fillfactor='100')
”surface_geom_cityobj_fkx” btree (cityobject_id) WITH (fillfactor='90')
”surface_geom_inx” btree (gmlid, gmlid_codespace) WITH (fillfactor='90')
”surface_geom_parent_fkx” btree (parent_id) WITH (fillfactor='90')
”surface_geom_root_fkx” btree (root_id) WITH (fillfactor='90')
”surface_geom_solid_spx” gist (solid_geometry)
”surface_geom_spx” gist (geometry)
    Foreign−key constraints:
”surface_geom_cityobj_fk” FOREIGN KEY (cityobject_id)
REFERENCES cityobject(id) MATCH FULL ON UPDATE CASCADE
”surface_geom_parent_fk” FOREIGN KEY (parent_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE
”surface_geom_root_fk” FOREIGN KEY (root_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE
```

Figure C.2: `surface_geometry` table description

```
          Column            |  Type   | Nullable | Default
----------------------------+---------+----------+---------
 id                         | integer | not null |
 objectclass_id             | integer | not null |
 building_id                | integer |          |
 room_id                    | integer |          |
 building_installation_id   | integer |          |
 lod2_multi_surface_id      | integer |          |
 lod3_multi_surface_id      | integer |          |
 lod4_multi_surface_id      | integer |          |
    Indexes:
"thematic_surface_pk" PRIMARY KEY, btree (id) WITH (fillfactor='100')
"them_surface_bldg_inst_fkx" btree (building_installation_id) WITH (fillfactor='90')
"them_surface_building_fkx" btree (building_id) WITH (fillfactor='90')
"them_surface_lod2msrf_fkx" btree (lod2_multi_surface_id) WITH (fillfactor='90')
"them_surface_lod3msrf_fkx" btree (lod3_multi_surface_id) WITH (fillfactor='90')
"them_surface_lod4msrf_fkx" btree (lod4_multi_surface_id) WITH (fillfactor='90')
"them_surface_objclass_fkx" btree (objectclass_id) WITH (fillfactor='90')
"them_surface_room_fkx" btree (room_id) WITH (fillfactor='90')
    Foreign-key constraints:
"them_surface_bldg_inst_fk" FOREIGN KEY (building_installation_id)
REFERENCES building_installation(id) MATCH FULL ON UPDATE CASCADE
"them_surface_building_fk" FOREIGN KEY (building_id)
REFERENCES building(id) MATCH FULL ON UPDATE CASCADE
"them_surface_cityobject_fk" FOREIGN KEY (id)
REFERENCES cityobject(id) MATCH FULL ON UPDATE CASCADE
"them_surface_lod2msrf_fk" FOREIGN KEY (lod2_multi_surface_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE
"them_surface_lod3msrf_fk" FOREIGN KEY (lod3_multi_surface_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE
"them_surface_lod4msrf_fk" FOREIGN KEY (lod4_multi_surface_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE
"them_surface_objclass_fk" FOREIGN KEY (objectclass_id)
REFERENCES objectclass(id) MATCH FULL ON UPDATE CASCADE
"them_surface_room_fk" FOREIGN KEY (room_id)
REFERENCES room(id) MATCH FULL ON UPDATE CASCADE
```

Figure C.3: `thematic_surface` table description

```
    Column    |          Type           | Nullable | Default
--------------+-------------------------+----------+---------
 srid         | integer                 | not null |
 gml_srs_name | character varying(1000) |          |
    Indexes:
"database_srs_pk" PRIMARY KEY, btree (srid) WITH (fillfactor='100')
```

Figure C.4: `database_srs` table description

```
           Column            |                  Type                  | Nullable
-----------------------------+----------------------------------------+----------
 id                          | integer                                | not null
 objectclass_id              | integer                                | not null
 building_parent_id          | integer                                |
 building_root_id            | integer                                |
 class                       | character varying(256)                 |
 class_codespace             | character varying(4000)                |
 function                    | character varying(1000)                |
 function_codespace          | character varying(4000)                |
 usage                       | character varying(1000)                |
 usage_codespace             | character varying(4000)                |
 year_of_construction        | date                                   |
 year_of_demolition          | date                                   |
 roof_type                   | character varying(256)                 |
 roof_type_codespace         | character varying(4000)                |
 measured_height             | double precision                       |
 measured_height_unit        | character varying(4000)                |
 storeys_above_ground        | numeric(8,0)                           |
 storeys_below_ground        | numeric(8,0)                           |
 storey_heights_above_ground | character varying(4000)                |
 storey_heights_ag_unit      | character varying(4000)                |
 storey_heights_below_ground | character varying(4000)                |
 storey_heights_bg_unit      | character varying(4000)                |
 lod1_terrain_intersection   | geometry(MultiLineStringZ,7415)        |
 lod2_terrain_intersection   | geometry(MultiLineStringZ,7415)        |
 lod3_terrain_intersection   | geometry(MultiLineStringZ,7415)        |
 lod4_terrain_intersection   | geometry(MultiLineStringZ,7415)        |
 lod2_multi_curve            | geometry(MultiLineStringZ,7415)        |
 lod3_multi_curve            | geometry(MultiLineStringZ,7415)        |
 lod4_multi_curve            | geometry(MultiLineStringZ,7415)        |
 lod0_footprint_id           | integer                                |
 lod0_roofprint_id           | integer                                |
 lod1_multi_surface_id       | integer                                |
 lod2_multi_surface_id       | integer                                |
 lod3_multi_surface_id       | integer                                |
 lod4_multi_surface_id       | integer                                |
 lod1_solid_id               | integer                                |
 lod2_solid_id               | integer                                |
 lod3_solid_id               | integer                                |
 lod4_solid_id               | integer                                |
```

Figure C.5: `building` table description

```
    Indexes:
”building_pk” PRIMARY KEY, btree (id) WITH (fillfactor=’100’)
”building_lod0footprint_fkx” btree (lod0_footprint_id) WITH (fillfactor=’90’)
”building_lod0roofprint_fkx” btree (lod0_roofprint_id) WITH (fillfactor=’90’)
”building_lod1msrf_fkx” btree (lod1_multi_surface_id) WITH (fillfactor=’90’)
”building_lod1solid_fkx” btree (lod1_solid_id) WITH (fillfactor=’90’)
”building_lod1terr_spx” gist (lod1_terrain_intersection)
”building_lod2curve_spx” gist (lod2_multi_curve)
”building_lod2msrf_fkx” btree (lod2_multi_surface_id) WITH (fillfactor=’90’)
”building_lod2solid_fkx” btree (lod2_solid_id) WITH (fillfactor=’90’)
”building_lod2terr_spx” gist (lod2_terrain_intersection)
”building_lod3curve_spx” gist (lod3_multi_curve)
”building_lod3msrf_fkx” btree (lod3_multi_surface_id) WITH (fillfactor=’90’)
”building_lod3solid_fkx” btree (lod3_solid_id) WITH (fillfactor=’90’)
”building_lod3terr_spx” gist (lod3_terrain_intersection)
”building_lod4curve_spx” gist (lod4_multi_curve)
”building_lod4msrf_fkx” btree (lod4_multi_surface_id) WITH (fillfactor=’90’)
”building_lod4solid_fkx” btree (lod4_solid_id) WITH (fillfactor=’90’)
”building_lod4terr_spx” gist (lod4_terrain_intersection)
”building_objectclass_fkx” btree (objectclass_id) WITH (fillfactor=’90’)
”building_parent_fkx” btree (building_parent_id) WITH (fillfactor=’90’)
”building_root_fkx” btree (building_root_id) WITH (fillfactor=’90’)
    Foreign−key constraints:
”building_cityobject_fk” FOREIGN KEY (id)
REFERENCES cityobject(id) MATCH FULL ON UPDATE CASCADE
”building_lod0footprint_fk” FOREIGN KEY (lod0_footprint_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE
”building_lod0roofprint_fk” FOREIGN KEY (lod0_roofprint_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE ADE
”building_lod1solid_fk” FOREIGN KEY (lod1_solid_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE ADE
”building_lod2solid_fk” FOREIGN KEY (lod2_solid_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE ADE
”building_lod3solid_fk” FOREIGN KEY (lod3_solid_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE ADE
”building_lod4solid_fk” FOREIGN KEY (lod4_solid_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE
”building_objectclass_fk” FOREIGN KEY (objectclass_id)
REFERENCES objectclass(id) MATCH FULL ON UPDATE CASCADE
”building_parent_fk” FOREIGN KEY (building_parent_id)
REFERENCES building(id) MATCH FULL ON UPDATE CASCADE
”building_root_fk” FOREIGN KEY (building_root_id)
REFERENCES building(id) MATCH FULL ON UPDATE CASCADE
```

Figure C.6: `building` table indexes and foreign-key constraints

```
     Column      |          Type          | Nullable | Default
-----------------+------------------------+----------+---------
 id              | integer                | not null |
 is_ade_class    | numeric                |          |
 is_toplevel     | numeric                |          |
 classname       | character varying(256) |          |
 tablename       | character varying(30)  |          |
 superclass_id   | integer                |          |
 baseclass_id    | integer                |          |
 ade_id          | integer                |          |
    Indexes:
"objectclass_pk" PRIMARY KEY, btree (id) WITH (fillfactor='100')
"objectclass_baseclass_fkx" btree (baseclass_id) WITH (fillfactor='90')
"objectclass_superclass_fkx" btree (superclass_id) WITH (fillfactor='90')
    Foreign-key constraints:
"objectclass_ade_fk" FOREIGN KEY (ade_id)
REFERENCES ade(id) MATCH FULL ON UPDATE CASCADE ON DELETE CASCADE
```

Figure C.7: `objectclass` table description

```
         Column           |           Type             | Nullable | Default
--------------------------+----------------------------+----------+------------
 id                       | integer                    | not null | nextval()
 parent_genattrib_id      | integer                    |          |
 root_genattrib_id        | integer                    |          |
 attrname                 | character varying(256)     | not null |
 datatype                 | integer                    |          |
 strval                   | character varying(4000)    |          |
 intval                   | integer                    |          |
 realval                  | double precision           |          |
 urival                   | character varying(4000)    |          |
 dateval                  | timestamp with time zone   |          |
 unit                     | character varying(4000)    |          |
 genattribset_codespace   | character varying(4000)    |          |
 blobval                  | bytea                      |          |
 geomval                  | geometry(GeometryZ,7415)   |          |
 surface_geometry_id      | integer                    |          |
 cityobject_id            | integer                    |          |
Indexes:
”cityobj_genericattrib_pk” PRIMARY KEY, btree (id) WITH (fillfactor=’100’)
”genericattrib_cityobj_fkx” btree (cityobject_id) WITH (fillfactor=’90’)
”genericattrib_geom_fkx” btree (surface_geometry_id) WITH (fillfactor=’90’)
”genericattrib_parent_fkx” btree (parent_genattrib_id) WITH (fillfactor=’90’)
”genericattrib_root_fkx” btree (root_genattrib_id) WITH (fillfactor=’90’)
Foreign-key constraints:
”genericattrib_cityobj_fk” FOREIGN KEY (cityobject_id)
REFERENCES cityobject(id) MATCH FULL ON UPDATE CASCADE
”genericattrib_geom_fk” FOREIGN KEY (surface_geometry_id)
REFERENCES surface_geometry(id) MATCH FULL ON UPDATE CASCADE
”genericattrib_parent_fk” FOREIGN KEY (parent_genattrib_id)
REFERENCES cityobject_genericattrib(id) MATCH FULL ON UPDATE CASCADE
”genericattrib_root_fk” FOREIGN KEY (root_genattrib_id)
REFERENCES cityobject_genericattrib(id) MATCH FULL ON UPDATE CASCADE
```

Figure C.8: `cityobject_genericattrib` table description

```
         Column             |         Type            | Nullable | Default
----------------------------+-------------------------+----------+---------
 child_id                   | integer                 | not null |
 parent_id                  | integer                 | not null |
 join_table_or_column_name  | character varying(30)   | not null |
 min_occurs                 | integer                 |          |
 max_occurs                 | integer                 |          |
 is_composite               | numeric                 |          |
Indexes:
”aggregation_info_pk” PRIMARY KEY, btree
(child_id, parent_id, join_table_or_column_name)
Foreign-key constraints:
”aggregation_info_fk1” FOREIGN KEY (child_id)
REFERENCES objectclass(id) MATCH FULL ON UPDATE CASCADE ON DELETE CASCADE
”aggregation_info_fk2” FOREIGN KEY (parent_id)
REFERENCES objectclass(id) MATCH FULL ON UPDATE CASCADE ON DELETE CASCADE
```

Figure C.9: `aggregation_info` table description

D Queries postgresql

```
pg_ctl –D /usr/local/var/postgres stop
pg_ctl –D /usr/local/var/postgres start
open /usr/local/bin/jmeter
https://jdbc.postgresql.org/download.html
The logfiles are stored in
/Usr/local/var/postgres/log
```

Query 1

GraphQL request:

```
{location(position: {lat: 4.450846, long: 51.906183, alt: 0}){
latitude
longitude
altitude}}
```

GraphQL response:

```
{”data”: {”location”: {
      ”latitude”: 4.450846,
      ”longitude”: 51.906183,
      ”altitude”: 4.450846
}}}
```

Query 2

This query applies to the `delfshaven` dataset.

GraphQL request:

```
{citymodel(position: {lat: 4.450846, long: 51.906183, alt: 0}){
id}}
```

GraphQL response:

```
{”data”: {
    ”citymodel”: [
      {”id”: ”metadata_delfshaven”}
    ]
}}
```

SQL

SQL 1 (218 bytes):

```
SELECT metadata.id AS metadata_id,
metadata.object AS metadata_object
FROM cityjsondb.metadata
WHERE ST_Intersects(ST_SetSRID(ST_MakePoint(4.450846, 51.906183, 0.0), 4979),
ST_Transform(ST_MakeEnvelope(
CAST((metadata.object -> 'geographicalExtent') -> 0 AS FLOAT),
CAST((metadata.object -> 'geographicalExtent') -> 1 AS FLOAT),
CAST((metadata.object -> 'geographicalExtent') -> 3 AS FLOAT),
CAST((metadata.object -> 'geographicalExtent') -> 4 AS FLOAT),
CAST(substr(metadata.object ->> 'referenceSystem', 23, 10) AS INTEGER)), 4979))
```

Query 3

This query applies to the `delfshaven` dataset.

GraphQl request:

```
{radius100(position: {lat: 4.450846, long: 51.906183, alt: 0}){
id}}
```

GraphQL response with 56 IDs:

```
{"data": {
    "radius100":
    [
      {"id": "{8FBED2F2-731E-4259-98E0-78A3447E6F68}"},
.....
      {"id": "{62C18FA2-1E09-484B-9769-F38C03C424BE}"}
    ]
}}
```

SQL

SQL 1 for radius100 (43600 bytes):

```
SELECT city_object.id AS city_object_id, city_object.object AS city_object_object,
city_object.attributes AS city_object_attributes,
ST_AsEWKB(city_object.convexhull) AS city_object_convexhull,
ST_AsEWKB(city_object.globalconvexhull) AS city_object_globalconvexhull,
city_object.metadata_id AS city_object_metadata_id
FROM cityjsondb.city_object
WHERE ST_DWithin(ST_SetSRID(ST_MakePoint(4.450846, 51.906183, 0.0), 4979),
ST_Transform(city_object.convexhull, 4979), 100, false) AND
(city_object.object ->> 'type') = 'Building'
```

SQL 1 for radius100index (43600 bytes):

```
SELECT city_object.id AS city_object_id,
city_object.object AS city_object_object,
city_object.attributes AS city_object_attributes,
ST_AsEWKB(city_object.convexhull) AS city_object_convexhull,
ST_AsEWKB(city_object.globalconvexhull) AS city_object_globalconvexhull,
city_object.metadata_id AS city_object_metadata_id
FROM cityjsondb.city_object
WHERE ST_DWithin(ST_SetSRID(ST_MakePoint(4.450846, 51.906183, 0.0), 4979)
, city_object.globalconvexhull, 100, false)
AND (city_object.object ->> 'type') = 'Building'
```

Query 4

This query applies to the delfshaven dataset.

GraphQL request:

```
{inside(position: {lat: 4.450846, long: 51.906183, alt: 0}){
id}}
```

GraphQL response:

```
{”data”: {
    ”inside”: [
      {”id”: ”{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}”}
    ]
}}
```

SQL

SQL 1 for inside (800 bytes):

```
SELECT city_object.id AS city_object_id,
city_object.object AS city_object_object,
city_object.attributes AS city_object_attributes,
ST_AsEWKB(city_object.convexhull) AS city_object_convexhull,
ST_AsEWKB(city_object.globalconvexhull) AS city_object_globalconvexhull,
city_object.metadata_id AS city_object_metadata_id
FROM cityjsondb.city_object
WHERE ST_Intersects(ST_SetSRID(ST_MakePoint(4.450846, 51.906183, 0.0), 4979),
ST_Transform(city_object.convexhull, 4979))
AND (city_object.object ->> ’type’) = ’Building’
```

SQL 1 for insideindex (800 bytes):

```
SELECT city_object.id AS city_object_id,
city_object.object AS city_object_object,
city_object.attributes AS city_object_attributes,
ST_AsEWKB(city_object.convexhull) AS city_object_convexhull,
ST_AsEWKB(city_object.globalconvexhull) AS city_object_globalconvexhull,
city_object.metadata_id AS city_object_metadata_id
FROM cityjsondb.city_object
WHERE ST_Intersects(ST_SetSRID(ST_MakePoint(4.450846, 51.906183, 0.0), 4979),
city_object.globalconvexhull)
AND (city_object.object ->> ’type’) = ’Building’
```

Query 5

This query applies to the potsdam dataset. It uses "UUID_402a38ac-27d6-4a35-b725-ee1f9b1d725e". The geometry has 9 surfaces.

GraphQL request:

```
{cityobjects(id: ”UUID_402a38ac-27d6-4a35-b725-ee1f9b1d725e”){
  maxlod{
    id
}}}
```

GraphQL response:

```
{"data": {
    "cityobjects": [
      {"maxlod": {
          "id": "2846"}}
    ]
}}
```

SQL

SQL 1 (1208 bytes):

```
SELECT city_object.id AS city_object_id,
 city_object.object AS city_object_object,
 city_object.attributes AS city_object_attributes,
ST_AsEWKB(city_object.convexhull) AS city_object_convexhull,
ST_AsEWKB(city_object.globalconvexhull) AS city_object_globalconvexhull,
 city_object.metadata_id AS city_object_metadata_id
FROM cityjsondb.city_object
WHERE city_object.id = 'UUID_402a38ac-27d6-4a35-b725-ee1f9b1d725e'
```

SQL 2 (222 bytes):

```
SELECT geometries.id AS geometries_id,
 geometries.object AS geometries_object,
 geometries.city_object_id AS geometries_city_object_id
FROM cityjsondb.geometries
WHERE 'UUID_402a38ac-27d6-4a35-b725-ee1f9b1d725e' = geometries.city_object_id
```

Query 6a

This query applies to the `delfshaven` dataset. It uses

```
{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}
```

GraphQL request:

```
{cityobjects(id: "{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}"){
id}}
```

GraphQL response:

```
{"data": {
    "cityobjects": [
      {"id": "{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}"}
    ]
}}
```

SQL

SQL 1 (800 bytes):

```
SELECT city_object.id AS city_object_id,
city_object.object AS city_object_object,
city_object.attributes AS city_object_attributes,
ST_AsEWKB(city_object.convexhull) AS city_object_convexhull,
ST_AsEWKB(city_object.globalconvexhull) AS city_object_globalconvexhull,
city_object.metadata_id AS city_object_metadata_id
FROM cityjsondb.city_object
WHERE city_object.id = ’{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}’
```

Query 6b

This query applies to the `delfshaven` dataset. It uses

```
{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}
```

GraphQL request:

```
{cityobjects(id: ”{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}”){
id
attributes}}
```

GraphQL response:

```
{”data”: {
    ”cityobjects”: [
      {”id”: ”{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}”,
        ”attributes”: ”{
        \”status\”: \”1\”,
        \”bron_geo\”: \”Lidar 15-30 punten - nov. 2008\”,
        \”bron_tex\”: \”UltraCAM-X 10cm juni 2008\”,
        \”voll_tex\”: \”complete\”, \”TerrainHeight\”: 1.69}”}
    ]
}}
```

SQL

SQL 1 (800 bytes):

```
SELECT city_object.id AS city_object_id,
city_object.object AS city_object_object,
city_object.attributes AS city_object_attributes,
ST_AsEWKB(city_object.convexhull) AS city_object_convexhull,
ST_AsEWKB(city_object.globalconvexhull) AS city_object_globalconvexhull,
city_object.metadata_id AS city_object_metadata_id
FROM cityjsondb.city_object
WHERE city_object.id = ’{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}’
```

Query 6c

This query applies to the `delfshaven` dataset. It uses

```
{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}
```

GraphQL request:

```
{cityobjects(id: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}”){
id
geometries{
  id
  object
  semantics{
  id
  object
  surfaces{
    id
        geometry
        solidNum
        shellNumVoid
        surfaceNum
  }
}}}}
```

GraphQL response with 1 `RoofSurface`, 1 `GroundSurface` and 9 `WallSurface`:

```
{”data”: {
    ”cityobjects”: [{
        ”id”: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}”,
        ”geometries”: [{
            ”id”: ”508”,
            ”object”: ”{\”lod\”: 2, \”type\”: \”MultiSurface\”}”,
            ”semantics”: [
              {
                ”id”: ”1522”,
                ”object”: ”{\”type\”: \”RoofSurface\”}”,
                ”surfaces”: [
                  {
                    ”id”: ”8696”,
                    ”geometry”: ”\”POLYGON Z ((...))\”” ,
                    ”solidNum”: null ,
                    ”shellNumVoid”: 0,
                    ”surfaceNum”: 0
                  }
                ]
              },
              {
                ”id”: ”1523”,
                ”object”: ”{\”type\”: \”GroundSurface\”}”,
                ”surfaces”: [
                  {
                    ”id”: ”8697”,
                    ”geometry”: ”\”POLYGON Z ((...))\”” ,
                    ”solidNum”: null ,
                    ”shellNumVoid”: 0,
                    ”surfaceNum”: 1
                  }
                ]
              },
              {
                ”id”: ”1524”,
                ”object”: ”{\”type\”: \”WallSurface\”}”,
                ”surfaces”: [
                  {
                    ”id”: ”8698”,
                    ”geometry”: ”\”POLYGON Z ((...))\””,
                    ”solidNum”: null,
                    ”shellNumVoid”: 0,
                    ”surfaceNum”: 2
                  },
...
                  {
                    ”id”: ”8706”,
                    ”geometry”: ”\”POLYGON Z ((...))\””,
                    ”solidNum”: null,
                    ”shellNumVoid”: 0,
                    ”surfaceNum”: 10
                  }
                ]
}]}]}]}}
```

SQL

SQL 1 (800 bytes):

```
SELECT city_object.id AS city_object_id,
 city_object.object AS city_object_object,
 city_object.attributes AS city_object_attributes,
ST_AsEWKB(city_object.convexhull) AS city_object_convexhull,
ST_AsEWKB(city_object.globalconvexhull) AS city_object_globalconvexhull,
 city_object.metadata_id AS city_object_metadata_id
FROM cityjsondb.city_object
WHERE city_object.id = '{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}'
```

SQL 2 (136 bytes):

```
SELECT geometries.id AS geometries_id,
 geometries.object AS geometries_object,
 geometries.city_object_id AS geometries_city_object_id
FROM cityjsondb.geometries
WHERE '{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}' = geometries.city_object_id
```

SQL 3 (325 bytes):

```
SELECT semantic_surface.id AS semantic_surface_id,
 semantic_surface.object AS semantic_surface_object,
 semantic_surface.city_object_id AS semantic_surface_city_object_id,
 semantic_surface.geometries_id AS semantic_surface_geometries_id
FROM cityjsondb.semantic_surface
WHERE 508 = semantic_surface.geometries_id
```

SQL 4 (642 bytes):

```
SELECT surfaces.id AS surfaces_id,
ST_AsEWKB(surfaces.geometry) AS surfaces_geometry,
 surfaces.solid_num AS surfaces_solid_num,
 surfaces.shell_num_void AS surfaces_shell_num_void,
 surfaces.surface_num AS surfaces_surface_num,
 surfaces.geometries_id AS surfaces_geometries_id,
 surfaces.semantic_surface_id AS surfaces_semantic_surface_id,
 surfaces.city_object_id AS surfaces_city_object_id
FROM cityjsondb.surfaces
WHERE 1522 = surfaces.semantic_surface_id
```

SQL 5 (229 bytes):

```
SELECT ST_AsText(ST_GeomFromEWKB('\x01030000a040710000010000000a000000
 a01a2fdd821bf6407f6abcf4c5971a4162105839b4e83340105839b4521cf6401f85eb
 51c4971a4162105839b4e833401b2fdd24581cf64048e17a14e7971a4162105839b4e8
 33406de7fba9eb1cf6401f85eb51e6971a4162105839b4e833406de7fba9ef1cf640a4
 703d0a11981a4162105839b4e83340448b6ce7591cf6402db29d6f12981a4162105839b
 4e833405839b4c8581cf640b29def2711981a4162105839b4e83340105839b47e1bf640
 cdcccccc12981a4162105839b4e833406de7fba9771bf640a4703d0ac6971a416210583
 9b4e83340a01a2fdd821bf6407f6abcf4c5971a4162105839b4e83340'::bytea))
AS "ST_AsText_1"
```

SQL 6 (602 bytes):

```
SELECT surfaces.id AS surfaces_id,
ST_AsEWKB(surfaces.geometry) AS surfaces_geometry,
 surfaces.solid_num AS surfaces_solid_num,
 surfaces.shell_num_void AS surfaces_shell_num_void,
 surfaces.surface_num AS surfaces_surface_num,
 surfaces.geometries_id AS surfaces_geometries_id,
 surfaces.semantic_surface_id AS surfaces_semantic_surface_id,
 surfaces.city_object_id AS surfaces_city_object_id
FROM cityjsondb.surfaces
WHERE 1523 = surfaces.semantic_surface_id
```

SQL 7 (249 bytes):

```
SELECT ST_AsText(ST_GeomFromEWKB('...'::bytea)) AS "ST_AsText_1"
```

SQL 8 (2607 bytes):

```
SELECT surfaces.id AS surfaces_id,
ST_AsEWKB(surfaces.geometry) AS surfaces_geometry,
 surfaces.solid_num AS surfaces_solid_num,
 surfaces.shell_num_void AS surfaces_shell_num_void,
 surfaces.surface_num AS surfaces_surface_num,
 surfaces.geometries_id AS surfaces_geometries_id,
 surfaces.semantic_surface_id AS surfaces_semantic_surface_id,
 surfaces.city_object_id AS surfaces_city_object_id
FROM cityjsondb.surfaces
WHERE 1524 = surfaces.semantic_surface_id
```

SQL 9 (149 bytes), 10 (150 bytes), 11 (156 bytes), 12 (154 bytes), 13 (151 bytes), 14 (151 bytes), 15 (151 bytes), 16 (153 bytes), 17 (154 bytes):

```
SELECT ST_AsText(ST_GeomFromEWKB('...'::bytea)) AS "ST_AsText_1"
```

Query 7

This query applies to the denhaag dataset. It uses the city object "GUID_273C3ED5-F33F-442D-ADD3-898E307B4516_1" with surface ”16477”.

GraphQL request:

```
{surfaces(id: ”16477”){
id
geometry
solidNum
shellNumVoid
surfaceNum
semantics{
  id
  object
}}}
```

GraphQL response:

```
{”data”: {
    ”surfaces”: [
      {
        ”id”: ”16477”,
        ”geometry”: ”\”POLYGON Z ((...))\”” ,
        ”solidNum”: 0,
        ”shellNumVoid”: 0,
        ”surfaceNum”: 6,
        ”semantics”: {
          ”id”: ”3554”,
          ”object”: ”{\”type\”: \”RoofSurface\”,
          \”Slope\”: 46.434,
          \”Direction\”: 188.116}”}
          }
    ]
}}
```

SQL

SQL 1 (469 bytes):

```
SELECT surfaces.id AS surfaces_id ,
ST_AsEWKB(surfaces.geometry) AS surfaces_geometry ,
surfaces.solid_num AS surfaces_solid_num ,
surfaces.shell_num_void AS surfaces_shell_num_void ,
surfaces.surface_num AS surfaces_surface_num ,
surfaces.geometries_id AS surfaces_geometries_id ,
surfaces.semantic_surface_id AS surfaces_semantic_surface_id ,
surfaces.city_object_id AS surfaces_city_object_id
FROM cityjsondb.surfaces
WHERE surfaces.id = ’16477’
```

SQL 2 (166 bytes):

```
SELECT ST_AsText(ST_GeomFromEWKB('...'::bytea)) AS ”ST_AsText_1”
```

SQL 3 (223 bytes):

```
SELECT semantic_surface.id AS semantic_surface_id,
 semantic_surface.object AS semantic_surface_object,
 semantic_surface.city_object_id AS semantic_surface_city_object_id,
 semantic_surface.geometries_id AS semantic_surface_geometries_id
FROM cityjsondb.semantic_surface
WHERE semantic_surface.id = 3554
```

E Queries 3DCityDB

Query 1

GraphQL request:

```
{location(position: {lat: 4.450846, long: 51.906183, alt: 0}){
latitude
longitude
altitude}}
```

GraphQL response:

```
{"data": {"location": {
      "latitude": 4.450846,
      "longitude": 51.906183,
      "altitude": 4.450846
}}}
```

Query 3

This query applies to the `delfshaven` dataset.

GraphQL request:

```
{radius100(position: {lat: 4.450846, long: 51.906183, alt: 0}){
id}}
```

GraphQL response with 59 IDs:

```
{"data": {"radius100": [
      {"id": "177"},
      .....
      {"id": "3337"}
]}}
```

SQL

SQL 1 (34841 bytes):

```
SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
building.id AS building_id,
cityobject.id AS cityobject_id,
cityobject.gmlid AS cityobject_gmlid,
ST_AsEWKB(cityobject.envelope) AS cityobject_envelope,
...
building.lod1_solid_id AS building_lod1_solid_id,
building.lod2_solid_id AS building_lod2_solid_id
FROM cityobject JOIN building ON building.id = cityobject.id
WHERE ST_DWithin(ST_SetSRID(ST_MakePoint(4.450846, 51.906183, 0.0), 4979),
ST_Transform(cityobject.envelope, 4979), 100, false)
```

Query 4

This query applies to the `delfshaven` dataset. 1973 in 3DCityDB corresponds to

```
{A3DF9B7C−9349−4703−88F4−C971EDB9D0A6}
```

.

GraphQL request:

```
{inside(position: {lat: 4.450846, long: 51.906183, alt: 0}){
id}}
```

GraphQL response:

```
{”data”: {
    ”inside”: [{
        ”id”: ”1973”
}]}}
```

SQL

SQL 1 (2063 bytes):

```
SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
building.id AS building_id,
cityobject.id AS cityobject_id,
cityobject.gmlid AS cityobject_gmlid,
ST_AsEWKB(cityobject.envelope) AS cityobject_envelope,
...
building.lod1_multi_surface_id AS building_lod1_multi_surface_id,
building.lod2_multi_surface_id AS building_lod2_multi_surface_id,
FROM cityobject JOIN building ON building.id = cityobject.id
WHERE ST_Intersects(ST_SetSRID(ST_MakePoint(4.450846, 51.906183, 0.0), 4979),
ST_Transform(cityobject.envelope, 4979))
```

Query 5

This query applies to the `potsdam` dataset. It uses `"UUID_402a38ac-27d6-4a35-b725-ee1f9b1d725e"`. This ID corresponds to 14502 in 3DCityDB. The geometry has 9 surfaces.

GraphQL request:

```
{buildings(id: ”14502”){
  maxlod{
    id}
}}
```

GraphQL response with 9 `surface_geometries`:

```
{”data”: {
    ”buildings”: [{
        ”maxlod”: [
            {”id”: ”37938”},
            {”id”: ”37942”},
            {”id”: ”37948”},
            {”id”: ”37952”},
            {”id”: ”37956”},
            {”id”: ”37960”},
            {”id”: ”37964”},
            {”id”: ”37969”},
            {”id”: ”37972”}
]}]}}
```

SQL

SQL 1 (2045 bytes):

```
SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
building.id AS building_id,
cityobject.id AS cityobject_id,
cityobject.gmlid AS cityobject_gmlid,
ST_AsEWKB(cityobject.envelope) AS cityobject_envelope,
...
building.lod1_multi_surface_id AS building_lod1_multi_surface_id
FROM cityobject JOIN building ON building.id = cityobject.id
WHERE building.id = '14502'
```

SQL 2 (4553 bytes):

```
statement: SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
thematic_surface.id AS thematic_surface_id,
cityobject.id AS cityobject_id
....
FROM cityobject JOIN thematic_surface ON thematic_surface.id = cityobject.id
WHERE 14502 = thematic_surface.building_id
```

SQL 3 (490 bytes), 4 (490 bytes), 5 (490 bytes), 6 (490 bytes), 7 (490 bytes), 8 (490 bytes), 9 (490 bytes), 10 (490 bytes), 11 (490 bytes):

```
SELECT surface_geometry.id AS surface_geometry_id
...
FROM surface_geometry
WHERE surface_geometry.id = 37942
```

Query 6a

This query applies to the `delfshaven` dataset. It uses 1973.

GraphQL request:

```
{cityobjects(id: ”1973”){
id}}
```

GraphQL response:

```
{"data": {
    "cityobjects": [{
        "id": "1973"
}]}}
```

SQL

SQL1 (807 bytes):

```
SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
cityobject.id AS cityobject_id, cityobject.gmlid AS cityobject_gmlid,
ST_AsEWKB(cityobject.envelope) AS cityobject_envelope,
...
FROM cityobject
WHERE cityobject.id = '1973'
```

Query 6b

This query applies to the `delfshaven` dataset. It uses 1973.

GraphQL request:

```
{cityobjects(id: "1973"){
id
genericattrib{
  attrname
  value
}}}
```

GraphQL response:

```
{"data": {
    "cityobjects": [
      {"id": "1973",
        "genericattrib": [
          {"attrname": "TerrainHeight",
            "value": "1.6900000000"},
          {"attrname": "bron_tex",
            "value": "UltraCAM-X 10cm juni 2008"},
          {"attrname": "voll_tex",
            "value": "complete"},
          {"attrname": "bron_geo",
            "value": "Lidar 15-30 punten - nov. 2008"},
          {"attrname": "status",
            "value": "1"}
]}]}}
```

SQL

SQL 1 (807 bytes):

```
SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
 cityobject.id AS cityobject_id, cityobject.gmlid AS cityobject_gmlid,
ST_AsEWKB(cityobject.envelope) AS cityobject_envelope,
 ...
FROM cityobject
WHERE cityobject.id = '1973'
```

SQL 2 (1030 bytes):

```
SELECT cityobject_genericattrib.id AS cityobject_genericattrib_id,
 cityobject_genericattrib.attrname AS cityobject_genericattrib_attrname,
 cityobject_genericattrib.datatype AS cityobject_genericattrib_datatype,
 ...
FROM cityobject_genericattrib
WHERE 1973 = cityobject_genericattrib.cityobject_id
```

Query 6c

This query applies to the `delfshaven` dataset. It uses 1973.

GraphQL request:

```
{cityobjects(id: "1973"){
  id
  building{
    thematicSurfaces{
      id
      objectclass{
        classname}
      surfaces{
        id
        children{
          id}
        isSolid
        isComposite
        solidGeometry
        geometry
}}}}}
```

GraphQL response:

```
{"data": {
    "cityobjects": [{
        "id": "1973",
        "building": [{
            "thematicSurfaces": [
              {
              "id": "1975",
                "objectclass": {
                  "classname": "BuildingRoofSurface"},
                "surfaces": [
                  ... 2 surface_geometryTypes of which one with a geometry ...]
               },
              {
              ”id”: ”1978”,
                ”objectclass”: {
                  ”classname”: ”BuildingGroundSurface”},
                ”surfaces”: [
                  ... 2 surface_geometryTypes of which one with a geometry ...]
               },
              {
              ”id”: ”1980”,
                ”objectclass”: {
                  ”classname”: ”BuildingWallSurface”},
                ”surfaces”: [
                  {”id”: ”9984”,
                    ”children”: [
                      ... 10 IDs of surface_geometryTypes ...
                    ],
                    ”isSolid”: 0,
                    ”isComposite”: 0,
                    ”solidGeometry”: null,
                    ”geometry”: null
                  },
                  {... 9 other surface_geometryTypes with geometries ... }]
               }
]}]}]}}
```

SQL

SQL 1 (807 bytes):

```
SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
 cityobject.id AS cityobject_id,
 ...
FROM cityobject
WHERE cityobject.id = ’1973’
```

SQL 2 (2063 bytes):

```
SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
 building.id AS building_id,
 ...
 building.lod4_solid_id AS building_lod4_solid_id
FROM cityobject JOIN building ON building.id = cityobject.id
WHERE 1973 = building.id
```

SQL 3 (1997 bytes):

```
SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
 thematic_surface.id AS thematic_surface_id,
 ...
 thematic_surface.lod2_multi_surface_id AS thematic_surface_lod2_multi_surface_id
 ...
```

```
FROM cityobject JOIN thematic_surface ON thematic_surface.id = cityobject.id
WHERE 1973 = thematic_surface.building_id
```

SQL 4 (232 bytes), 9 (234 bytes), 14 (232 bytes):

```
SELECT objectclass.id AS objectclass_id ,
 objectclass.classname AS objectclass_classname
 ...
FROM objectclass
WHERE objectclass.id = 33
```

SQL 5 (984 bytes), 6 (984 bytes), 7 (395 bytes), 10 (944 bytes), 11 (944 bytes), 12 (395 bytes), 15 (3148 bytes), 16 (3148 bytes), 17 (395 bytes), 19 (395 bytes), 21 (395 bytes), 23 (395 bytes), 25 (395 bytes), 27 (395 bytes), 29 (395 bytes), 31 (395 bytes), 33 (395 bytes):

```
SELECT surface_geometry.id AS surface_geometry_id
 ...
FROM surface_geometry
WHERE 1975 = surface_geometry.cityobject_id
```

SQL 8 (299 bytes), 13 (249 bytes), 18 (149 bytes), 20 (150 bytes), 22 (156 bytes), 24 (156 bytes), 26 (151 bytes), 28 (151 bytes), 30 (151 bytes), 32 (153 bytes), 34 (154 bytes):

```
SELECT ST_AsText(ST_GeomFromEWKB('...'::bytea)) AS ”ST_AsText_1”
```

Query 7

This query applies to the `denhaag` dataset. The city object `"GUID_273C3ED5-F33F-442D-ADD3-898E307B4516_1"` corresponds to 13024 in 3DCityDB and the surface to 42242.

GraphQL request:

```
{surfaceGeometry(id: ”42242”) {
   id
   children{
      id
      isSolid
      isComposite
      solidGeometry
      geometry
      }
   thematicSurfaceMsrf2{
            id
      genericattrib{
            attrname
            value}
      objectclass {
            classname}
      }
}}
```

GraphQL response:

```
{”data”: {
      ”surfaceGeometry”: [
         {”id”: ”42242”,
           ”children”: [
          {”id”: ”42242”,
            ”isSolid”: 0,
            ”isComposite”: 0,
            ”solidGeometry”: null,
            ”geometry”: null},
          {”id”: ”42243”,
            ”isSolid”: 0,
            ”isComposite”: 0,
            ”solidGeometry”: null,
            ”geometry”: ”\”POLYGON Z ((...))\””}
        ],
        ”thematicSurfaceMsrf2”: [
          {”id”: ”13031”,
            ”genericattrib”: [
              {”attrname”: ”Direction”,
                ”value”: ”188.1160000000”},
              {”attrname”: ”Slope”,
                ”value”: ”46.4340000000”}
            ],
            ”objectclass”: {
              ”classname”: ”BuildingRoofSurface”}
          }
]}]}}
```

SQL

SQL 1 (490 bytes):

```
SELECT surface_geometry.id AS surface_geometry_id
 ...
FROM surface_geometry
WHERE surface_geometry.id = ’42242’
```

SQL 2 (815 bytes):

```
SELECT surface_geometry.id AS surface_geometry_id
 ...
FROM surface_geometry
WHERE 42242 = surface_geometry.root_id
```

SQL 3 (166 bytes):

```
SELECT ST_AsText(ST_GeomFromEWKB(’...’::bytea)) AS ”ST_AsText_1”
```

SQL 4 (1121 bytes):

```
SELECT (SELECT objectclass.tablename FROM objectclass
WHERE objectclass.id = cityobject.objectclass_id) AS anon_1,
 thematic_surface.id AS thematic_surface_id,
 ...
FROM cityobject JOIN thematic_surface ON thematic_surface.id = cityobject.id
WHERE 42242 = thematic_surface.lod2_multi_surface_id
```

SQL 5 (741 bytes):

```
SELECT cityobject_genericattrib.id AS cityobject_genericattrib_id,
...
FROM cityobject_genericattrib
WHERE 13031 = cityobject_genericattrib.cityobject_id
```

SQL 6 (232 bytes):

```
SELECT objectclass.id AS objectclass_id
...
FROM objectclass
WHERE objectclass.id = 33
```

F Queries MongoDB

Query 1

GraphQL request:

```
{location(position: {lat: 4.450846, long: 51.906183, alt: 0}){
latitude
longitude
altitude}}
```

GraphQL response:

```
{"data": {
    "location": {
      "latitude": 4.450846,
      "longitude": 51.906183,
      "altitude": 0
}}}
```

Query 2

This query applies to the delfshaven dataset.

GraphQL request:

```
{citymodel(position: {lat: 4.450846, long: 51.906183, alt: 0}){
_id}}
```

GraphQL response:

```
{"data": {
    "citymodel": [
      {
        "_id": "metadata_delfshaven"
}]}}
```

MQL

MQL 1:

```
ns: CityJSON.metadata
query: {}
sort: {}
projection: {},
planSummary: COLLSCAN
```

Size in Mongo Shell (623 bytes):

```
var cursor = db.metadata.find();
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

Query 3

This query applies to the `delfshaven` dataset.

GraphQL request:

```
{radius100(position: {lat: 4.450846, long: 51.906183, alt: 0}){
_id}}
```

GraphQL response with 56 IDs:

```
{"data": {
    "radius100":
    [
      {"id": "{8FBED2F2-731E-4259-98E0-78A3447E6F68}"},
.....
      {"id": "{62C18FA2-1E09-484B-9769-F38C03C424BE}"}
    ]
}}
```

MQL

MQL 1:

```
ns: CityJSON.metadata
query: {}
sort: {}
projection: {},
planSummary: COLLSCAN
```

Size in Mongo Shell (623 bytes):

```
var cursor = db.metadata.find({});
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

MQL 2:

```
ns: CityJSON.CityObjects
query: { metadata_id: ”metadata_delfshaven”, type: ”Building” }
sort: {}
projection: {},
planSummary: COLLSCAN
```

Size in Mongo Shell (3646573 bytes):

```
var cursor = db.CityObjects.find({
metadata_id: ”metadata_delfshaven”, type: ”Building” });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

MQL 3:

```
ns: CityJSON.CityObjects
query: { metadata_id: ”metadata_denhaag”, type: ”Building” }
sort: {}
projection: {},
planSummary: COLLSCAN
```

Size in Mongo Shell (246757 bytes):

```
var cursor = db.CityObjects.find({
metadata_id: ”metadata_denhaag”, type: ”Building” });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

MQL 4:

```
ns: CityJSON.CityObjects
query: { metadata_id: ”metadata_potsdam”, type: ”Building” }
sort: {}
projection: {},
planSummary: COLLSCAN
```

Size in Mongo Shell (28023661 bytes):

```
var cursor = db.CityObjects.find({
metadata_id: ”metadata_potsdam”, type: ”Building” });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

Query 4

This query applies to the `delfshaven` dataset.

GraphQL request:

```
{inside(position: {lat: 4.450846, long: 51.906183, alt: 0}){
_id}}
```

GraphQL response:

```
{"data": {
    "inside": [
      {"id": "{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}"}
    ]
}}
```

SQL

MQL 1:

```
ns: CityJSON.metadata
query: {}
sort: {}
projection: {},
planSummary: COLLSCAN
```

Size in Mongo Shell (623 bytes):

```
var cursor = db.metadata.find({});
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

MQL 2:

```
ns: CityJSON.CityObjects
query: { metadata_id: "metadata_delfshaven", type: "Building" }
sort: {}
projection: {},
planSummary: COLLSCAN
```

Size in Mongo Shell (3646573 bytes):

```
var cursor = db.CityObjects.find({
metadata_id: "metadata_delfshaven", type: "Building" });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

MQL 3:

```
ns: CityJSON.CityObjects
query: { metadata_id: ”metadata_denhaag”, type: ”Building” }
sort: {}
projection: {},
planSummary: COLLSCAN
```

Size in Mongo Shell (246757 bytes):

```
var cursor = db.CityObjects.find({
metadata_id: ”metadata_denhaag”, type: ”Building” });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

MQL 4:

```
ns: CityJSON.CityObjects
query: { metadata_id: ”metadata_potsdam”, type: ”Building” }
sort: {}
projection: {},
planSummary: COLLSCAN
```

Size in Mongo Shell (28023661 bytes):

```
var cursor = db.CityObjects.find({
metadata_id: ”metadata_potsdam”, type: ”Building” });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

Query 5

This query applies to the `potsdam` dataset. It uses `"UUID_402a38ac-27d6-4a35-b725-ee1f9b1d725e"`. The geometry has 9 surfaces.

GraphQL request:

```
{maxlod(Id: ”UUID_402a38ac−27d6−4a35−b725−ee1f9b1d725e”){_id}}
```

GraphQL response (the ID changes every time as explained in section 5.5.7:

```
{”data”: {
    ”maxlod”: {
      ”_id”: ”5f5cefb1f01ad9e941ced4d8”
}}}
```

MQL

MQL 1:

```
ns: CityJSON.CityObjects
query: { _id: "UUID_402a38ac-27d6-4a35-b725-ee1f9b1d725e" }
sort: {}
projection: {}
```

Size in Mongo Shell (5023 bytes):

```
var cursor = db.CityObjects.find({
_id: "UUID_402a38ac-27d6-4a35-b725-ee1f9b1d725e" });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

Query 6a

This query applies to the `delfshaven` dataset. It uses

```
{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}
```

GraphQL request:

```
{cityobjects(Id: "{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}"){
_id}}
```

GraphQL response:

```
{"data": {
    "cityobjects": [
      {"_id": "{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}"}
]}}
```

MQL

MQL 1:

```
ns: CityJSON.CityObjects
query: { _id: "{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}" }
sort: {}
projection: {}
```

Size in Mongo Shell (2948 bytes):

```
var cursor = db.CityObjects.find({
_id: "{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}" });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

Query 6b

This query applies to the `delfshaven` dataset. It uses

{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}

GraphQL request:

```
{cityobjects(Id: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}”){
_id
attributes {
  TerrainHeight
  bron_tex
  voll_tex
  bron_geo
  status
}}}
```

GraphQL response:

```
{”data”: {
    ”cityobjects”: [
      {
        ”_id”: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}”,
        ”attributes”: {
          ”TerrainHeight”: 1.69,
          ”bron_tex”: ”UltraCAM–X 10cm juni 2008”,
          ”voll_tex”: ”complete”,
          ”bron_geo”: ”Lidar 15−30 punten − nov. 2008”,
          ”status”: ”1”
        }
}]}}
```

MQL

MQL 1:

```
ns: CityJSON.CityObjects
query: { _id: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}” }
sort: {}
projection: {}
```

Size in Mongo Shell (2948 bytes):

```
var cursor = db.CityObjects.find({ _id: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}” });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

Query 6c

This query applies to the `delfshaven` dataset. It uses

```
{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}
```

GraphQL request:

```
{cityobjects(Id: ”{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}”){
_id
geometry{
  type
  lod
  semantics{
    values
    surfaces{
      type}
  }
  boundaries
}}}
```

GraphQL response:

```
{”data”: {
    ”cityobjects”: [
      {
        ”_id”: ”{A3DF9B7C-9349-4703-88F4-C971EDB9D0A6}”,
        ”geometry”: [
          {
            ”type”: ”MultiSurface”,
            ”lod”: 2,
            ”semantics”: {
              ”values”: [
                0,1,2,2,2,2,2,2,2,2,2
              ],
              ”surfaces”: [
                {”type”: ”RoofSurface”
                },
                {”type”: ”GroundSurface”
                },
                {”type”: ”WallSurface”
                }
              ]
            },
            ”boundaries”: [
            [[
        [90552.179,435697.489,19.909],
        [90565.169,435697.08,19.909],
        [90565.509,435705.77,19.909],
        [90574.729,435705.58,19.909],
        [90574.979,435716.26,19.909],
        [90565.619,435716.609,19.909],
        [90565.549,435716.289,19.909],
        [90551.919,435716.7,19.909],
        [90551.479,435697.51,19.909]
                ]],
...
```

```
              [[
        [90552.179,435697.489,19.909],
        [90551.479,435697.51,19.909],
        [90551.479,435697.51,0],
        [90552.179,435697.489,0]
                ]]
]}]}]}}
```

MQL

MQL 1:

```
ns: CityJSON.CityObjects
query: { _id: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}” }
sort: {}
projection: {}
```

Size in Mongo Shell (2948 bytes):

```
var cursor = db.CityObjects.find({
_id: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}” });
var size = 0;
cursor.forEach(
    function(doc){
        size += Object.bsonsize(doc)
    }
);
print(size);
```

Query 7

This query is not implemented, but it is possible to query the surface and the semantic surface object together.

```
{cityobjects(Id: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}”){
_id
geometry{
  type
  lod
  boundarysurfaces{
    surface
    semanticsobject{
      type
    }
}}}}
```

```
{
  ”data”: {
    ”cityobjects”: [
      {
        ”_id”: ”{A3DF9B7C−9349−4703−88F4–C971EDB9D0A6}”,
        ”geometry”: [{
            ”type”: ”MultiSurface”,
            ”lod”: 2,
            ”boundarysurfaces”: {
              ”surface”: [[
        [90552.179,435697.489,19.909],
        [90565.169,435697.08,19.909],
        [90565.509,435705.77,19.909],
        [90574.729,435705.58,19.909],
        [90574.979,435716.26,19.909],
        [90565.619,435716.609,19.909],
        [90565.549,435716.289,19.909],
        [90551.919,435716.7,19.909],
        [90551.479,435697.51,19.909]
                ]],
              ”semanticsobject”: {
                ”type”: ”RoofSurface”
              }
}}]}]}}
```

G Query plan postgresql

The database makes a query plan in order to execute the query efficiently and to optimize the retrieval times.

Query 3A

```
Seq Scan on city_object
(cost=0.00..172904.02 rows=1 width=602)
(actual time=174.887..1280.344 rows=56 loops=1)
Filter: (((object –>> ’type’::text) = ’Building’::text) AND
st_dwithin(’...’::geography,
(st_transform(convexhull, 4979))::geography, ’100’::double precision, false))
Rows Removed by Filter: 6202
Planning Time: 0.589 ms
Execution Time: 1286.851 ms
```

Query 3B

```
Seq Scan on city_object
(cost=0.00..157259.02 rows=1 width=602)
(actual time=2.248..55.857 rows=56 loops=1)
Filter: (((object –>> ’type’::text) = ’Building’::text) AND
st_dwithin(’...’::geography,
(globalconvexhull)::geography, ’100’::double precision, false))
Rows Removed by Filter: 6202
Planning Time: 0.529 ms
Execution Time: 55.957 ms
```

Query 4A

```
Seq Scan on city_object
(cost=0.00..172888.37 rows=1 width=602)
(actual time=192.300..1292.854 rows=1 loops=1)
Filter: (((object –>> ’type’::text) = ’Building’::text) AND
st_intersects(’...’::geometry, st_transform(convexhull, 4979)))
Rows Removed by Filter: 6257
Planning Time: 0.073 ms
Execution Time: 1299.088 ms
```

Query 4B

The query is using the index on the `globalconvexhull`.

```
Index Scan using globalconvexhull_index on city_object
(cost=0.15..33.67 rows=1 width=602)
(actual time=0.273..0.273 rows=1 loops=1)
Index Cond: (globalconvexhull && ’...’::geometry)”
Filter: (((object –>> ’type’::text) = ’Building’::text) AND
st_intersects(’...’::geometry, globalconvexhull))
Planning Time: 0.406 ms
Execution Time: 0.344 ms
```

Query 6C

SQL query 4, 6 and 8 are using the index on the `semantic_surface_id`.

```
Index Scan using semantic_surface_index on surfaces
(cost=0.42..8.95 rows=2 width=96)
(actual time=0.031..0.031 rows=1 loops=1)
Index Cond: (semantic_surface_id = 1522)
Planning Time: 0.422 ms
Execution Time: 0.045 ms
```

Bibliography

Berg, K. L., Seymour, T., and Goel, R. (2012). History of databases. *International Journal of Management & Information Systems (IJMIS)*, 17(1):29–36.

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., and Coltekin, A. (2015). Applications of 3d city models: State of the art review. *ISPRS International Journal of Geo-Information*, 4:2842–2889.

Blut, C. and Blankenbach, J. (2020). Three-dimensional CityGML building models in mobile augmented reality: a smartphone-based pose tracking system. *International Journal of Digital Earth*, pages 1–20.

Blut, C., Blut, T., and Blankenbach, J. (2019). CityGML goes mobile: application of large 3d CityGML models on smartphones. *International Journal of Digital Earth*, 12(1):25–42.

Chen, P. P.-S. (1976). The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36.

Davoudian, A., Chen, L., and Liu, M. (2018). A survey on NoSQL stores. *ACM Computing Surveys*, 51(2):1–43.

Foundation, T. G. (2020). Graphql current working draft.

Frank, A. U. (1988). Requirement for a database management system for a gis. Technical report.

Fritchey, G. (2018). *SQL Server 2017 Query Performance Tuning*. Apress.

Gleison Brito, Thais Momback, M. T. V. (2019). Migrating to graphql: a practical assessment. *IEEE*.

Guo, Y., Deng, F., and Yang, X. (2018). Design and implementation of real-time management system architecture based on GraphQL. *IOP Conference Series: Materials Science and Engineering*, 466:012015.

Kolbe, T. H. (2019). 3d city database for citygml documentation version 4.2. Technical report, Technische universitat Munchen.

Ledoux, H., Ohori, K. A., Kumar, K., Dukai, B., Labetski, A., and Vitalis, S. (2019). Cityjson: a compact and easy-to-use encoding of the citygml data model. *Open Geospatial Data, Software and Standards*, 4(4).

Meier, A. and Kaufmann, M. (2019). *SQL & NoSQL Databases*. Springer Fachmedien Wiesbaden.

MongoDB (2020). The mongodb 4.4 manual.

Open Geospatial Consortium (2012). Ogc city geography markup language (citygml) encoding standard, version: 2.0.0.

Oussous, A., Benjelloun, F.-Z., Lahcen, A. A., and Belfkih, S. (2015). Comparison and classification of nosql databases for big data.

Petkovic, D. (2017). Json integration in relational database systems. *International Journal of Computer Applications*, 168(5).

Schellevis, L., Zuidweg, H., Penninga, F., Snoei, M., Roes, J., and Haasnoot, P. (2019). Api strategie voor de nederlandse overheid. pages 1–20.

Stadler, A., Nagel, C., König, G., and Kolbe, T. H. (2009). Making interoperability persistent: A 3d geo database based on citygml.

Taskula, T. (2019). Advanced data fetching with graphql: Case bakery service.

Višnjevac, N., Mihajlović, R., Šoškic
, M., Cvijetinović, Ž., and Bajat, B. (2017). Using NoSQL databases in the 3d cadastre domain. *Geodetski vestnik*, 61(03):412–426.

Zhu, Q., Hu, M., Zhang, Y., and Du, Z. (2009). Research and practice in three-dimensional city modeling. *Geo-spatial Information Science*, 12(1):18–24.