Suitability of MongoDB compared to other databases for the storage and querying of CityJSON using GraphQL

Karin Jacquelien Staring
student #4952510

1e supervisor: Stelios Vitalis
2e supervisor: Linda van den Brink

November 13, 2019

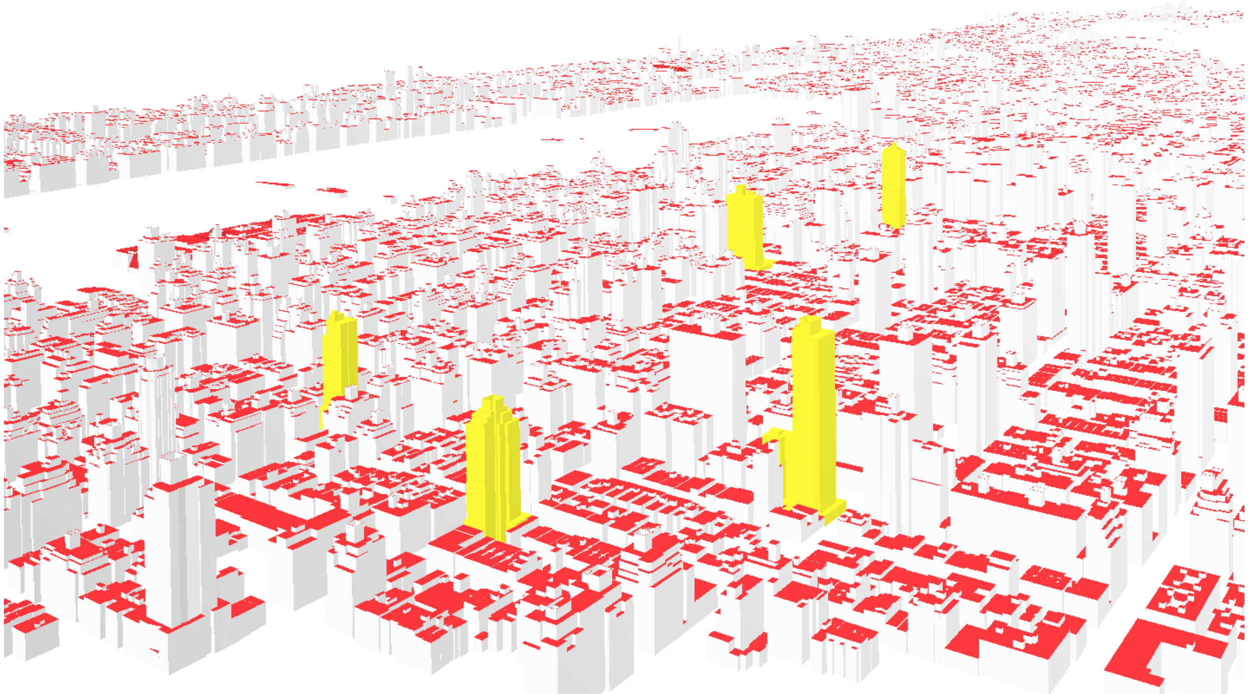

Figure 1: CityJSON dataset of New York from cityjson.org

Glossary

API Application Programming Interface, which is an interface for the communication between applications (?). 7, 8

BSON BSON is the binary representation of JSON. 7, 16

buffer A strategy in which, once data elements are loaded, the disk block is kept in memory to avoid waiting for the rotating disk, which is a slower memory storage device. This is also known as caching. The disk block is then kept in cache (?). 3

cluster Disk blocks of data are stored together as a physical cluster on a disk. Clustering means that objects that are used together are stored together to reduce the number of disk accesses (?). 3

GML Geographic Markup Language. 3

ID Identity. 3

index An index is a special lookup table that the database uses to improve retrieval time. 3

JSON JavaScrip Object Notation. 3, 7, 9, 13, 14, 16

NoSQL not only SQL. 2, 4, 5, 6, 9, 14

REST API REST (REpresentational State Transfer) API (Application Program Interface) is a standardized architecture style to structure and build an API. These APIs are build upon URLs and the HTTP communication protocol (?). 2, 7, 8

serialization It converts an object to bytes. This is done to save the state of an object and to exchange the data across different language and programming environments. 7

sharding The partitioning of large volumes of data across servers and virtual data nodes (?). 5

SQL Structured Query Language. 2, 4, 5, 9

XML Extensible Markup Language. 3

1. Related work with background information

Most subsections are background information about the individual components, which are CityJSON, database requirements, SQL databases, NoSQL databases, MongoDB, REST APIs and GraphQL. There is one related work subsection, namely "CityGML implementations in SQL and NoSQL databases".

1.1. Data format CityJSON

In this section, there are two subsections. The first subsection "CityGML and CityJSON" explains the relation between CityGML and CityJSON. The specifications of CityJSON can be found on cityjson.org under the tab specifications. In the subsection "Design of CityJSON", the reasons behind certain design choices will be explained.

1.1.1. CityGML and CityJSON

CityGML and CityJSON are different encodings of the CityGML data model. CityJSON is built upon JSON, while CityGML is build upon XML and GML. XML and GML are both markup languages, whose structures can not easily be managed when programming. JSON matches better with the data structure of most programming languages and is more compact, because it uses arrays and dictionaries instead of tags (**?**). JSON is therefore mostly used in a more interactive way and for dynamic web applications and XML is mostly used for data management applications (**?**). XML is also more difficult to read for humans than JSON. Besides this, CityJSON enables only one way to represent the semantics and geometries of a feature, which also simplifies the usage. On the other hand, XML schemas to validate XML documents are nowadays better developed than JSON schemas and XML supports namespaces. Schemas are typically used to check if a documents satisfies a certain structure and content. CityJSON is however able to use additional software such as cjio and val3dity to add extra validation tools that cannot be expressed with JSON schemas (**?**).

1.1.2. Design of CityJSON

There are parts inside the CityJSON file that use indices or/and IDs. Vertices, which are (xyz) coordinates of geometric primitives, are for instance defined inside the CityJSON object. The reason for this is that vertices can be shared by repeating the index and not the vertex. In this way, topological relationships can be found more easily and made more robust. Beside that, repeating the index instead of the vertex might be more efficient because vertices are often shared by several surfaces. It is also possible to transform the entire CityJSON object to make the file less prone to rounding (**?**).

1.2. Database requirements

1. We must be able to query the data in the database with a query language. When doing this, the data must be accessed with sufficient retrieval time and with a limited rate of query processing. To achieve this, the following mechanisms can be implemented: indices, clusters and buffers (**?**).
2. Scalability is the capacity of a database system to adapt to the amount of data stored, which is needed to guarantee that the database is always available (**?**).
3. Database users can only execute operations if they have the authority to do these operations. These operations should not be executed by unauthorized users (**?**).
4. Transaction reliability should be implemented to support strongly consistent data (**?**). This will assure that data will be modified in allowed ways (**?**).
5. The content of the database should be easily explorable. Adding metadata and visualization tools could be used to explore the database.
6. For spatial databases, an additional requirement is maintaining topological consistency.

The literature describes more database requirements than the ones listed above, but this research does not take them into consideration.

1.2.1. Transaction reliability further explained

ACID properties ensure the following:

1. 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 (?).
2. Consistency guarantees that transactions never observe or cause inconsistent data (?).
3. 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 concurrent transactions (?).
4. 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 (?).

However, not all database systems rely on these ACID properties. NoSQL databases often rely on BASE properties (?).

BASE properties ensure the following:

1. Basically Available guarantees the availability of data even when the database system contains some failures. This could be done by the replication of data across multiple storage nodes. However, this does not necessary include permanent storage when all servers shut down (?).
2. Soft State means that it is the developer's task to guarantee consistency. This can be achieved by testing if the new or changed data contradicts with the already stored facts in the database. If there are no contradictions present, the database is consistent. The state of the system can be invalid for some time, which means that the database does not maintain consistency in terms of the ACID properties (?).
3. Eventual consistency means that the system with all storage nodes will become consistent over time, when it does not receive input anymore. However, during this time requesting different storage nodes may result in different responses (?).

1.3. Differences between SQL and NoSQL databases

SQL databases are mainly designed for transactions and insurance software. However, transaction reliability often impedes sufficient retrieval time of large amounts of data, because transaction reliability requires processing power. For social web services, sufficient retrieval time is more important than transaction reliability. This is why NoSQL databases were developed (?).

Differences between SQL and NoSQL databases			
Characteristics	SQL	NoSQL	Section or reference
Database schema	predefined schema	no or more flexible schema	1.3.1
Encryption to guarantee transaction reliability	easy	difficult	1.3.1
Support unstructured data	difficult	easy	1.3.1
Scalability	vertically scalable	horizontally scalable	1.3.2
Many simultaneous requests	difficult	easy	1.3.2
Allow joins	yes	no	1.3.2
Properties transaction reliability	ACID	BASE (or other properties)	1.3.2
Use of Standard Query Language	yes	no	(?)
Support for spatial operations	more complex ones	basic ones	(?)

Table 1: Summary of the differences between SQL and NoSQL databases

1.3.1. Database schema and encryption

SQL databases are table structured databases with a database schema at design time, which enables the encryption of data. The encryption of data is important to guarantee transaction reliability. When the database is incorrectly changed afterwards, this could damage and reduce the performance of the SQL database . NoSQL databases are developed for datasets that need more flexibility. These databases are not table structured and it is easier to store data without a predefined schema or to change the schema afterwards (?). This simplifies the schema design. It enables easy evolution of data formats and it facilitates quick integration of data from different sources (?). However, the encryption of data to guarantee transaction reliability is more difficult (?).

1.3.2. Scalability

SQL databases are vertically scalable and NoSQL databases are horizontally scalable. For vertical scalability, users have to increase the CPU, RAM or the SSD of their database server to store datasets that do not fit. One server can be easily overloaded with simultaneous requests. An option is to add manual sharding as an additional feature to the SQL database, but this is complex to use. Sharding is the partitioning of large volumes of data across servers and virtual data nodes (?). However, sharding can be easily done by scaling horizontally. In this case, the load will be shared by adding more servers in a cluster environment when the size of the data increases (?). Sharding is used for NoSQL databases to ensure parallel storage and processing, but it does not allow joins between shards. This means that joins are difficult in NoSQL databases. A downside of maintaining multiple servers is that NoSQL databases ensure less transaction reliability. They rely on the BASE properties instead of the ACID properties (?).

1.4. Different types of NoSQL databases

NoSQL databases are mainly categorized into key-value, wide-column, document and graph databases (?).

1.5. MongoDB

Information about MongoDB can be found on https://docs.mongodb.com/manual/.

1.5.1. Design of MongoDB

MongoDB is a Document Database. The database contains collections with documents as can be seen in figure 2. Every stored document gets a unique key, which is the `_id` field.

Characteristics of the `_id` field:

1. It is a primary key to uniquely identify the documents in the collection.
2. Automatically indexed
3. ObjectID by default or overwritten by the user
4. Once a document exists, the `_id` field is immutable

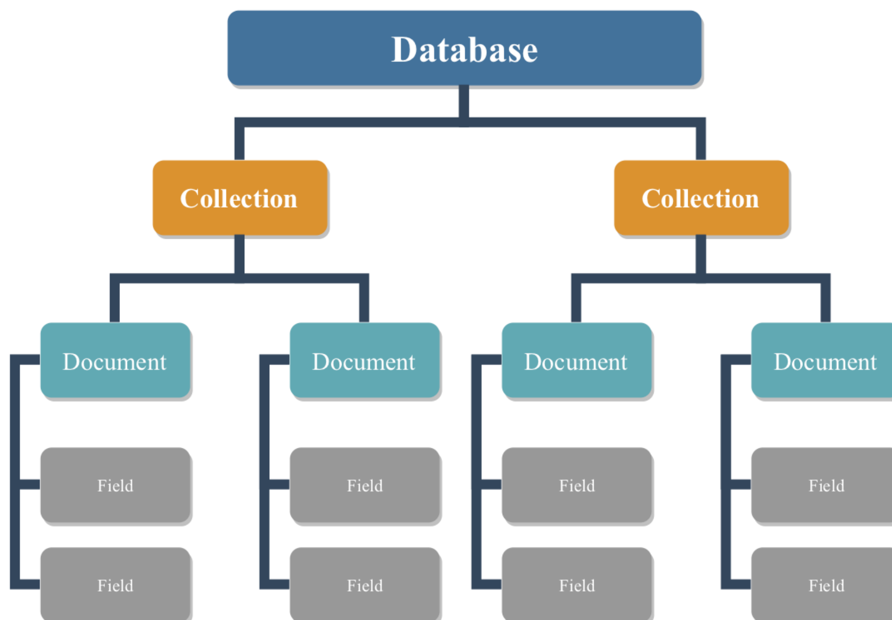

Figure 2: Storage concepts of MongoDB

1.5.2. MongoDB References

The two types of references are manual references and DBRefs. The difference between them is that manual references save the `_id` field from another document as reference. This type allows referencing inside one collection. On the other hand, DBRefs allows referencing between different collections by storing the `_id` field, collection name and, optionally, database name. In both cases, additional queries have to be done to retrieve the referenced documents. Manual references are preferred.

1.5.3. MongoDB and GeoJSON

MongoDB supports geospatial queries on GeoJSON Objects when the coordinates are related to the WGS84 coordinate reference system and supports the following GeoJSON object types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, GeometryCollection in 2D.

1.5.4. MongoDB and JSON schema

The $jsonSchema operator/object can be used as document validator when inserting and updating documents in MongoDB and contains a JSON Schema object according to [1]. However, the following are not supported in MongoDB:

1. Hyper-text and hyper-media definitions allow JSON data to be understood as hyper-text [2]. Hyper-text is text that links text to other texts and hyper-media is hyper-text that could include graphics, videos and sounds [3].
2. The keywords $ref, $schema, default, definitions, format, id and unknown keywords
3. The integer type can not used, but the BSON types must be used with the bsonType keyword. BSON is the serialization format used to store documents in MongoDB and to make remote procedure calls which allow the client to use a remote server.

1.6. GraphQL

This section consists of three subsections. The subsection "REST API" explains the originally used REST API, which is standardized architecture style to structure and build an API. The subsection "GraphQL" explains how GraphQL works in general. The specifications of the GraphQL language can be found on https://graphql.github.io/graphql-spec/. The last subsection, namely "Strengths and weaknesses of GraphQL", outlines the strengths and weaknesses of GraphQL.

1.6.1. REST API

1.6.2. GraphQL

GraphQL is a new query language for APIs invented by Facebook. Because GraphQL is a translator, it does not mandate a specific storage engine or programming language (**?**). GraphQL queries will only be send in the body of a GET and POST request, while REST APIs also use PUT etc.

The GraphQL schema is a multi-graph that starts with a root node that branches into operation types and then into API-specific resources with types and fields. The GraphQL schema also includes resolving the operations with resolver functions in the operation type (**?**). The GraphQL schema is defined on the server-side (**?**).

GraphQL has the following operation types:

1. Query to obtain data
2. Mutation to change data

[1] https://tools.ietf.org/html/draft-zyp-json-schema-04
[2] https://tools.ietf.org/html/draft-luff-json-hyper-schema-00
[3] https://www.w3.org/WhatIs.html

3. Subscription to push data from the server to the client

A GraphQL Query is hierarchical and structured. The type system defines the relationships between objects. Each level of the GraphQL query typically corresponds to a specific type (?). In figure 3, there are two levels requested, namely the city object and a nested attribute object. Types can be nested or even recursive. The meaning of the name GraphQL is that entities and relationships can be regarded as a graph (?).

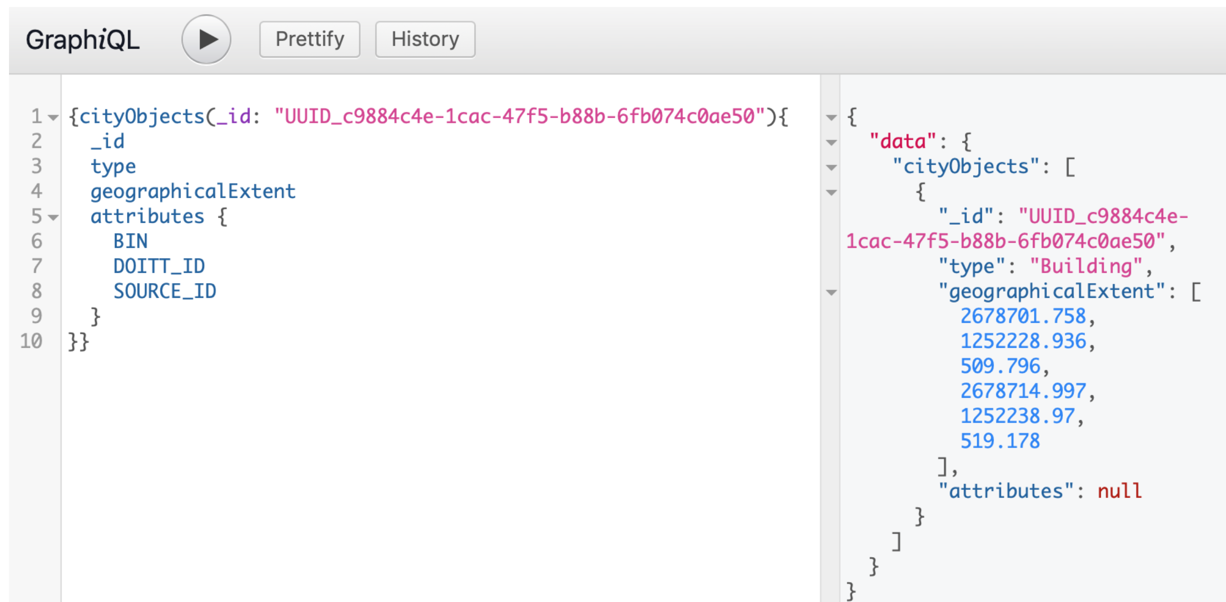

Figure 3: GraphQL request and response in GraphiQL

Every field is null by default. Instead of having a complete failure for the request, null will be returned when the database goes down, an asynchronous action fails or an exception is thrown. In figure 3, it can be seen that there are no attributes found and null is returned. It is possible to use the non-null variant of types when it is necessary to guarantee that the field will never return null to the client (?).

1.6.3. Strengths and weaknesses of GraphQL

Strengths of GraphQL:

1. REST APIs typically use multiple endpoints, while GraphQL typically uses one. Calling multiple endpoints to request the required data increases the client-server calls (?). While GraphQL could use multiple endpoints, this could make it more difficult to use GraphQL with tools like GraphiQL, which is a React component that allows us to write GraphQL queries in our browser (?). Besides that, GraphiQL shows the self-documenting nature of GraphQL, because it auto-completes fields as we type.

2. GraphQL only returns the data from the API endpoint that is specifically requested. In GraphQL, the specification for queries are encoded on the client-side rather than the server-side (?). This is a strength of GraphQL, because the client controls the returned data. New capabilities could be added to GraphQL via new types and fields without creating breaking changes that would require a new API version. REST APIs have limited control over the returned data, which means that every change can be considered a breaking change. A breaking change requires a new API version (?) (?).

Weaknesses of GraphQL:

1. Private fields, which are private variables in a class that can only be modified from inside the class and not outside the class, are not supported in GraphQL (?).
2. GraphQL does not follow the HTTP specification for caching. The implementation of caching is more difficult since every query can be different and it is up to the developer (?). The DataLoader utility might be an useful implementation for this (?).
3. The performance of the server could drop when the GraphQL server has to process complex queries such as queries with deep nesting (?).

1.7. CityGML implementations in SQL and NoSQL databases

SQL databases such as PostgreSQL with a PostGIS extension support spatial operations including 3D operators. But when querying a CityGML file stored in a SQL database, the amount of processing time increases significantly due to the large number of tables and relations to traverse on. Implementing the CityGML schema in a SQL database is also considered to be difficult to implement and can not be easily extended or parellelized (?). 3DCityDB is an implementation that stores CityGML in PostgreSQL (?). Before storing CityGML in the NoSQL database MongoDB, the data has to be converted to JSON (?). When storing data in MongoDB, results suggest that the performance of queries are better than SQL databases by an average factor of 10 which increases exponentially when the data size increases (?). The described weaknesses of storing data in NoSQL databases are the shortcomings of 3D support, ACID properties and a standard query language (?). Nevertheless the results are promising for handling and querying large datasets (?). MongoDB supports for instance Map-Reduce analysis to make use of parallel computation and (spatial) indexing to speed up the database query operations (?). Another interesting implementation is called GeoRocket, which is a cloud-based Geodata store which uses MongoDB on the backend (?).

2. Research scope

The scope of the research is limited to storing a simple CityJSON file in different databases. The meaning of a simple CityJSON file is shown in figure 4. A simple CityJSON file excludes the following:

1. Geometry-templates object, including geometryInstances
2. Extensions object
3. The geometry object of the Address object
4. Appearance object including materials and textures

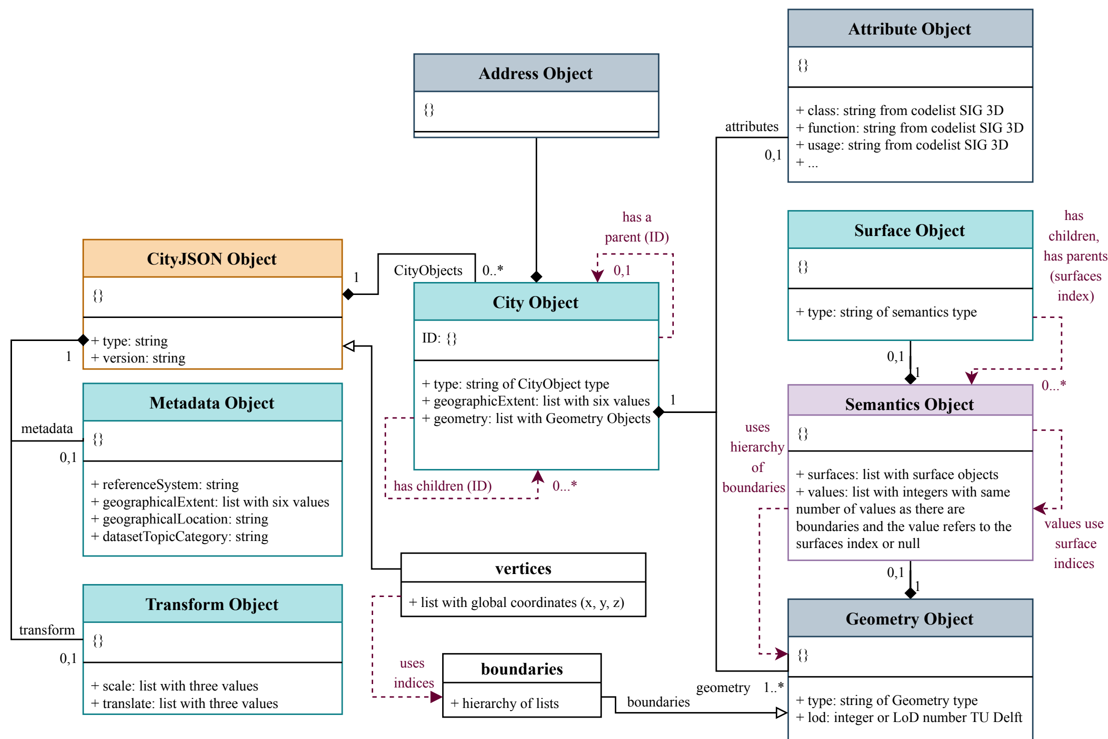

Figure 4: UML diagram of a simple CityJSON file

When investigating the objects that occur in a CityJSON file, the first three objects were uncommon and therefore they are excluded. While the appearance object is common, this object is excluded, because of its complexity. The appearances can be removed with the citygml-tools [4].

3. Methodology

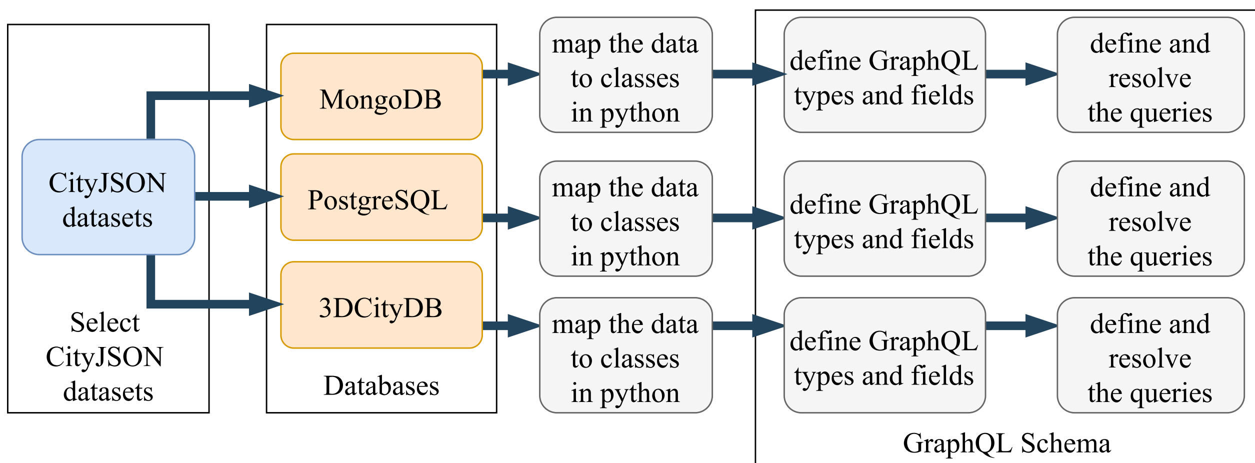

Figure 5: Workflow

[4]https://github.com/citygml4j/citygml-tools

3.1. Research question 1

The datasets will be stored in three databases, namely MongoDB, PostgreSQL(own implementation) and PostgreSQL(3DCityDB). In section **??**, it is decribed how CityJSON files could be loaded in MongoDB and PostgreSQL (3DCityDB).

3.1.1. Overview of the decisions to store a CityJSON file

A lot of decisions have to be made to store a CityJSON file. An overview of these decisions will be given in a list below.

1. How to store the objects: type and version?
2. How to store the object: metadata?
3. How to store the different types of city objects?
4. How to store parents and children?
5. How to store the object: attribute?
6. How to store the object: address?
7. How to store geometry related objects and members (including the boundaries, vertices, reference system, geographical extent and transform object)?
8. How to store the object: semantics?

The relations between the objects are given in figure 4. The first research question can be answered by storing the CityJSON file in the database and converting the data in the database back to a CityJSON file. In this way, it could be seen if all objects are stored and if the extracted CityJSON file is the same as the original.

3.1.2. Storage of CityJSON in MongoDB

The database contains one CityJSON object. Different collections could be used to store the city objects. Every document inside the collection contains a different city object. In this way, manual references inside the collection can be used and DBRefs between collections. Because every city object is stored in a separate document, the size of every individual document is kept small.

According to the list in section 3.1.1, 7 options are considered.

7 options			
type and version	in metadata collection		
metadata	in metadata collection		
city objects	one collection for all city objects	every city object type in a different collection	
parents and children	reference the children and parents with IDs	different parent and child collection	nest the children in the parent object
attribute	part of city object		
address	part of city object		
boundaries and vertices	replace the index with the vertices.	replace the index with the vertices. If there is a transform object, the vertices are transformed before the index is replaced. Possible advantage: faster retrieval time when vector calculations on arrays are slow, possible disadvantage: the database is less robust	
transform	in transform collection		
semantics	store as it is: advantages are that there is no redundancy which also means more reliability	replace the values with the surface: advantage: might have a faster retrieval time, although we expect that obtaining the value from an array with the index is almost as fast	

Table 2: The investigated decisions for the storage of CityJSON in MongoDB

It is no option to link surfaces and values which refer to the boundaries using IDs, because the IDs would apply to the global collection instead of the local document. We expect that obtaining the value from an array with the index is much faster.

However, this is impossible for files with many vertices, because the vertices could not be stored in one document due to the fact that the server of MongoDB only supports BSON documents with a size up to 16793598 bytes. Therefore, every vertex must be stored in a separate document with the index as ID and DBRefs would be needed to join them since the references are in different collections. This is no option, because we expect that joins will be slow.

To decide which options could be used best when using GraphQL as usecase, a Python script is made to store CityJSON in MongoDB according to the different options from table 2 and to convert the data in the database back to a CityJSON file. In this way, all options can be tested based on simplicity meaning that the implementation must be as close as possible to the structure and naming of CityJSON, retrieval time and query possibilities when using

GraphQL.

3.1.3. Validation of CityJSON in MongoDB

MongoDB uses the $jsonSchema operator, which is described in section 1.5.4 and the CityJSON Schema from section 5 will be used as JSON Schema object. Because MongoDB lacks support for specific keywords, the integer type and hyper-text/media definitions, the JSON Schema object has to be changed when validating the documents in MongoDB.

3.2. Research question 3

The data in the databases must be mapped to classes in Python. Three issues are described below.

1. When manual references are used to store the parents and children, the objects must be referenced or embedded inside one class.

2. How to map the data in the database to classes in python in a more flexible way? For example, it would be inconvenient to specify every member(key-value pair) that could occur in a simple CityJSON file in advance.

3. How to store the geometries? Preliminary research shows that some issues are encountered when storing the geometries. For example, MultiSurfaces could be stored by specifying the hierarchy of four lists and a float value on forehand. However, the hierarchy of Solids is different and therefore Solids can not be stored for now.

4. Time planning

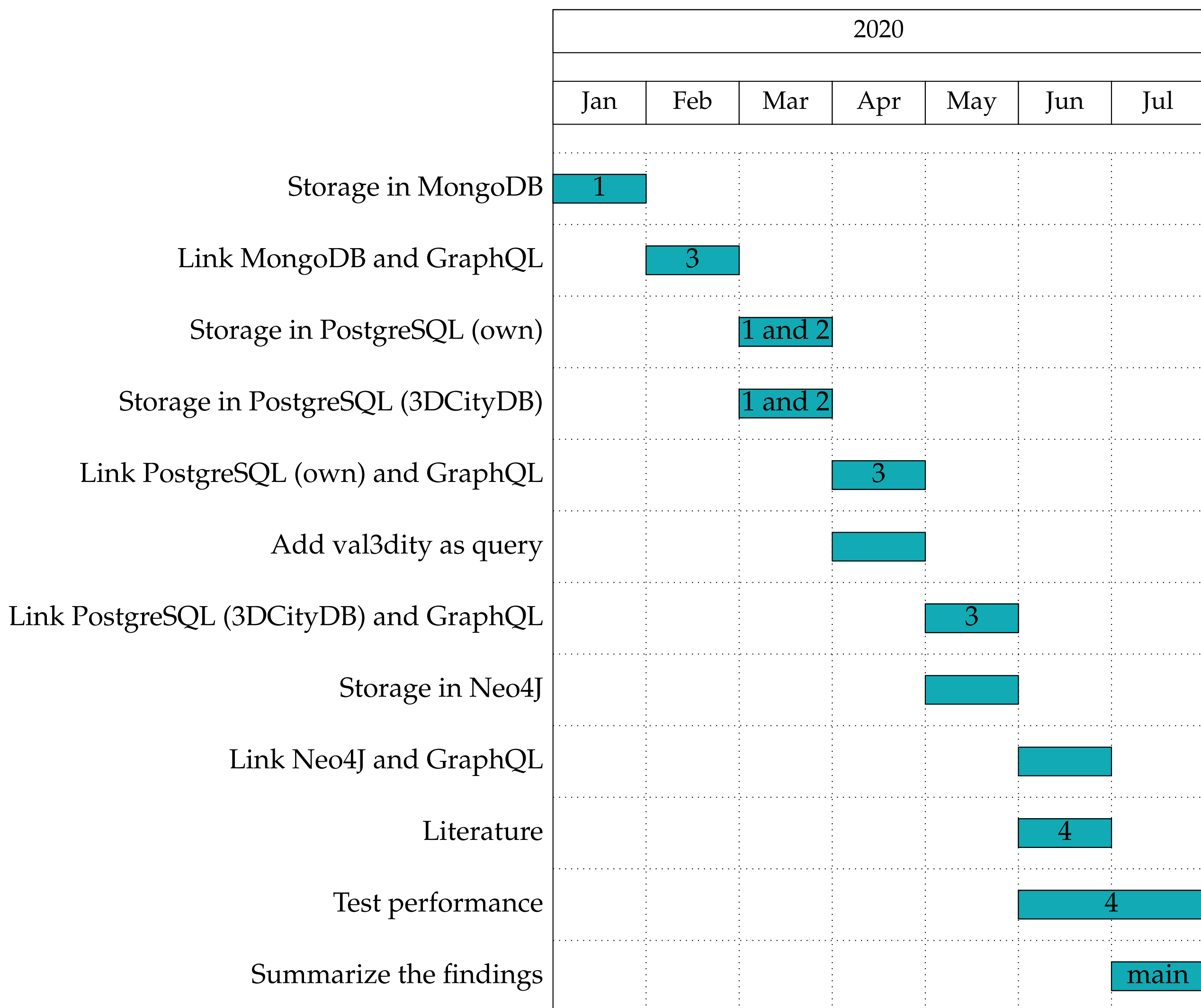

Figure 6: Gantt chart: the numbers in the chart belong to the research questions

If there is time left from april to june, CityJSON will also be stored in the NoSQL database Neo4J and accessed using GraphQL.

5. Datasets

The used datasets are described below. The semantics of newyork are removed, because the file would exceed the maximum (100 MB) of Github otherwise. Table 3 and table 4 describe the datasets. The JSON Schema for CityJSON, namely cityjson.min.schema.json [5], is used for validation, because it does not use references.

1. helsinki: 197 KB (small size) [6]

2. delfshaven: 1.4 MB (small size) from cityjson.org

[5] https://3d.bk.tudelft.nl/schemas/cityjson/1.0.1/
[6] CityGML_BUILDINGS_LOD2_NOTEXTURES_664502x2.gml

3. denhaag: 2.6 MB (small size) from cityjson.org
4. vienna: 6.5 MB (small size) from cityjson.org
5. potsdam: 16.9 MB (medium size) [7]
6. newyork without semantics stored: 62.9 MB (large size) from cityjson.org

Specifications of the files						
Specification	helsinki	delfshaven	denhaag	vienna	potsdam	newyork
metadata	X	X	X	X	X	X
referenceSystem attribute		X	X		X	
geographicalExtent attribute	X	X	X	X	X	X
attributes	X	X	X	X	X	X
address	X				X	
measuredHeight attribute	X			X	X	
roofType attribute	X		X	X	X	
parents and children			X	X		
semantics	X	X	X	X	X	X
appearances	X	X	X		X	
extensions						
geometry-templates						
transform	X	X				X

Table 3: Specifications of the CityJSON files used

Geometry types of the files						
Specification	helsinki	delfshaven	denhaag	vienna	potsdam	newyork
MultiPoint						
MultiLineString						
MultiSurface		X		X	X	X
CompositeSurface			X			
Solid	X		X	X	X	
MultiSolid						
CompositeSolid						
LoDs used	1, 2	2	2	2	1, 2	2
Number of LoDs per cityobject	2	1	1	2	2	1

Table 4: Geometry types of the CityJSON files used

[7] https://de.ftp.opendatasoft.com /potsdam/Gebmodell3D$_{C}ityGML/Potsdam3D_3_6.zip$

6. Research question 1

6.1. Storage of CityJSON in MongoDB

There are a few issues encountered when storing CityJSON in MongoDB. There are namely three reasons why the geometries of CityJSON are problematic to store as GeoJSON Objects.

1. MongoDB only allows us to do geospatial queries on GeoJSON when the coordinates are related to the WGS84 reference system. Because coordinates in CityJSON can also be related to other reference systems, the geometries of CityJSON can not directly be stored in MongoDB.

2. Not all geometry types of CityJSON are supported by GeoJSON. For instance, the geometry type MultiSurface of CityJSON is not supported by GeoJSON. The MultiSurface must be converted to a MultiPolygon to become a GeoJSON Object.

3. GeoJSON is originally in 2D instead of 3D.

The only time when the extracted CityJSON file is different from the original CityJSON file is when the key "presentLoDs" is present in the original CityJSON file. The key could not be stored in MongoDB, because MongoDB does not allow to store a dot as a key as can be seen below.

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

Besides this, there are two other reasons why the extracted CityJSON file might seem different from the original CityJSON file.

1. The original file could store duplicate surface objects, while the extracted file does not.

2. The vertices or surfaces are stored in a different sequence.

6.2. Validation of CityJSON in MongoDB

Validation rules can be added to a collection by adding the $jsonSchema operator. Since every collection uses the $jsonSchema operator to set schema validation rules, the original JSON Schema for CityJSON has to be converted to individual JSON Schema objects. The implemented changes are described in the list below:

1. The JSON Schema object for the metadata collection contains the type, version and metadata objects from the original JSON Schema. The keyword format is deleted and the integer type had to be replaced from (type: integer) to (bsonType: int). Because BSON is used to store documents in MongoDB, the keyword type is always replaced by bsonType.

2. The JSON Schema object for the transform collection contains the transform object from the original JSON Schema. The same replacement as used in the metadata collection is applied to this $jsonSchema.

In figure 7 and figure 8, it is shown how validation in MongoDB works. There is one document inserted that fails the validation and one that passes the validation.

```
{
  $jsonSchema: {
    properties: {
      scale: {
        bsonType: 'array',
        items: {
          bsonType: 'number'
        },
        maxItems: 3,
        minItems: 3
      },
      translate: {
        bsonType: 'array',
        items: {
          bsonType: 'number'
        },
        maxItems: 3,
        minItems: 3
      }
    },
    required: [
      'scale',
      'translate'
    ],
    type: 'object'
  }
}
```

Figure 7: Document that passed the validation rules

```
{
  $jsonSchema: {
    properties: {
      scale: {
        bsonType: 'array',
        items: {
          bsonType: 'number'
        },
        maxItems: 3,
        minItems: 3
      },
      translate: {
        bsonType: 'array',
        items: {
          bsonType: 'number'
        },
        maxItems: 3,
        minItems: 3
      }
    },
    required: [
      'scale',
      'translate'
    ],
    type: 'object'
  }
}
```

Figure 8: Document that failed the validation rules, because the scale object is required, but not present.

A. Preliminary results

A.1. Access MongoDB using GraphQL

1. Type is a built-in method in Python, which could not be provided as argument when querying.
2. `_id` can not be used as argument when querying, but is used in MongoDB as the primary key to uniquely identify the documents in the collection.
3. How to map the data in the database to classes in python in a more flexible way? For example, it would be inconvenient to specify every member(key-value pair) that could occur in a simple CityJSON file in advance.
4. How to store the boundaries of a geometryObject? For now, it is only possible to store MultiSurfaces. MultiSurfaces contain 4 lists and then a float value. This is different for Solids for instance and therefore these can not be stored for now. Hopefully, there is a way to initialize the boundaries of a class in a flexible way or to initialize the boundaries of a class based on the geometry type.
5. How could we query the fields of a nested object/type? This can be for instance the LoD of a cityObject. This can be done by iterating over all cityObjects to check if one of the

geometryObjects of the cityObject has a LoD field. If the LoD field has the value which is specified in the query, the cityObject is appended to results.

6. How could we filter cityObjects based on multiple fields? For instance, how could we filter objectType = "Building" and LoD = 2? This could be done by keeping the cityObjects where objectType = "Building"and to verify if these remaining cityObjects have a geometry where LoD = 2. Only the cityObjects that satisfy both conditions are returned.

B. Remaining questions that have to be investigated

1. XML and JSON schemas are different from each other (inheritance and namespaces)? But does this matter in the case of CityJSON? CityJSON has cjio and val3dity.
2. Is it logical to say that GraphQL could be used/develop as (standard) query language for NoSQL databases?
3. Is it an advantage if the geometry and semantics can only be specified in one way? Why are there many ways to represent the same geometry in CityGML?
4. Are there already 3D data models for the Netherlands in use? If not, why not?
5. papers 3DCityDB: https://www.3dcitydb.org/3dcitydb/publications/
6. Is the data linked with the code?
7. Shall I also include a query based on date?
8. How to deal with rounding of the transform object and vertices?
9. Would it possible to add schema validation in MongoDB and use the JSON validation schema of cityjson.org?
10. OGC EXAMPLE interesting: http://docs.opengeospatial.org/per/18-021.pdf