cjdb

A Simple, Fast, and Lean Database Solution for the CityGML Data Model

Powałka, Leon; Poon, Chris; Xia, Yitong; Meines, Siebren; Yan, Lan; Cai, Yuduan; Stavropoulou, Gina; Dukai, Balázs; Ledoux, Hugo

DOI
10.1007/978-3-031-43699-4_47

Publication date
2024

Document Version
Final published version

Published in
Recent Advances in 3D Geoinformation Science

Citation (APA)
Powałka, L., Poon, C., Xia, Y., Meines, S., Yan, L., Cai, Y., Stavropoulou, G., Dukai, B., & Ledoux, H. (2024). cjdb: A Simple, Fast, and Lean Database Solution for the CityGML Data Model. In T. H. Kolbe, A. Donaubauer, & C. Beil (Eds.), *Recent Advances in 3D Geoinformation Science: Proceedings of the 18th 3D GeoInfo Conference* (pp. 781-796). (Lecture Notes in Geoinformation and Cartography). Springer. https://doi.org/10.1007/978-3-031-43699-4_47

cjdb: A Simple, Fast, and Lean Database Solution for the CityGML Data Model

Leon Powałka, Chris Poon, Yitong Xia, Siebren Meines, Lan Yan, Yuduan Cai, Gina Stavropoulou, Balázs Dukai, and Hugo Ledoux

Abstract When it comes to storing 3D city models in a database, the implementation of the CityGML data model can be quite demanding and often results in complicated schemas. As an example, 3DCityDB, a widely used solution, depends on a schema having 66 tables, mapping closely the CityGML architecture. In this paper, we propose an alternative (called 'cjdb') for storing CityGML models efficiently in PostgreSQL with a much simpler table structure and data model design (only 3 tables are necessary). This is achieved by storing the attributes and geometries of the objects directly in JSON. In the case of the geometries we thus adopt the *Simple Feature* paradigm and we use the structure of CityJSON. We compare our solution against 3DCityDB with large real-world 3D city models, and we find that cjdb has significantly lower demands in storage space (around a factor of 10), allows for faster import/export of data, and has a comparable data retrieval speed with some queries being faster and some slower. The accompanying software (importer and exporter) is available at https://github.com/cityjson/cjdb/ under a permissive open-source license.

Keywords CityGML · 3DCityDB · 3D modelling · DBMS · CityJSON

1 Introduction

The international standard *City Geography Markup Language* (CityGML) is a data model designed for the storage of digital 3D models representing urban areas and landscapes (Kutzner et al. 2020; Gröger and Plümer 2012; OGC 2021b). It allows us to define and store the majority of commonplace 3D objects within cities, such as

This article was selected based on the results of a double-blind review of the full paper.

L. Powałka · C. Poon · Y. Xia · S. Meines · L. Yan · Y. Cai · G. Stavropoulou · H. Ledoux (✉)
Delft University of Technology, Delft, The Netherlands
e-mail: h.ledoux@tudelft.nl

B. Dukai
3DGI, Delft, the Netherlands

T. H. Kolbe et al. (eds.), *Recent Advances in 3D Geoinformation Science*, Lecture Notes in Geoinformation and Cartography, https://doi.org/10.1007/978-3-031-43699-4_47

buildings, roads, rivers, bridges, vegetation, and city furniture. Additionally, it supports various levels of detail (LoDs) for the 3D objects, which enables and facilitates complex applications and use-cases (Biljecki et al. 2015).

The CityGML data model, currently at version 3.0, has three known encodings (more details in Sect. 2):

XML/GML encoding: The XML/GML encoding (built upon GML (OGC 2007)) was initially the only standardised encoding for CityGML, which explains the—rather confusing—name choice for the data model. The latest official release of the XML/GML encoding supports CityGML version 2.0 (OGC 2012); however a release planned for 2023 will also support CityGML v3.0.

CityJSON: Its version 1.0 is standardised by the OGC (OGC 2021a), and its version 1.1[1] implements a subset of the CityGML v3.0 data model. Its flat hierarchy and simple structure make it around 6 times more compact than the XML/GML encoding, thus allowing for easier manipulation and exchange on the web (Ledoux et al. 2019).

3DCityDB: The *3D City Database* is a "geo-database solution" (schema and accompanying software) supporting three different relational DBMSs (database management systems). It implements a mapping of the CityGML data model (currently for v1.0 and v2.0 only) to the database schema to allow for a fast implementation (Yao et al. 2018). It is not standardised by the OGC.

DBMSs can greatly simplify the management of large 3D city models: they are arguably the best tool to store and manage very large datasets (of any kind), are already part of the ecosystem of many organisations, and offer several advantages over file-based systems, eg security, versioning, scalability, etc. (Ramakrishnan and Gehrke 2001). This makes 3DCityDB a popular solution, especially for handling country-level data and for offering access to multiple users. In most cases, the data owners store the data with 3DCityDB on a remote server and allow the users to access the data through a website, filter it by objects/LoDs/areas and obtain a subset of the 3D city model, in various different formats, for instance KML, COLLADA, and glTF.

However, while the 3DCityDB is widely used, Pantelios (2022) argues that its use can be somewhat complex and difficult for end-users. The main culprit is the fact that datasets are split over 66 tables and the *Simple Feature* paradigm OGC (2006) is not used (geometries are stored across different tables, not in a column of an object), which translates to very complex queries that necessitate several joins. Pantelios (2022) solution is to create extra views for attributes and geometries, and to offer simplified access to them through a graphical interface (built upon QGIS). However, this comes at the cost of increasing the size of the database.

We present in this paper an alternative to 3DCityDB, which we name 'cjdb'. It is composed of a database schema (containing only 3 tables) and accompanying software for import and export of CityJSON v1.1 files (thus the CityGML v3.0 core model is supported). As further explained in Sect. 3, our data model is inspired

[1] https://cityjson.org.

by the *Simple Feature* paradigm (each row has the geometries of the object stored in one column), but instead of using PostGIS geometry types, we exploit the fact that PosgresSQL can store JSON objects directly in binary format with the `jsonb` type. The reason for this choice is that PostGIS geometry types (notice that to use 3D types the SFCGAL extension (Borne et al. 2023) would be required), would not allow the storage of appearances (textures and/or materials) and of semantic information on the surfaces. Therefore, the geometries of a given city object (eg a building, a tree, a lamppost) are stored together with the object in JSON format, as defined by CityJSON. Our simple structure allows us to compress by an order of around 10 the typical size of a database as stored with 3DCityDB (taking into account data and (spatial) indexes), and, as shown in Sect. 4, this size reduction does not come with a penalty for the speed of the data retrieval. Our data model is at the moment only for PostgreSQL, but because it is so simple (only 3 tables are necessary), it could surely be ported to other databases.

2 Related Work

2.1 CityGML Data Model

To represent a region in 3D, CityGML recursively decomposes it into semantic objects (Gröger and Plümer 2012). It defines the classes most commonly found in an urban or a regional context, and the hierarchical relationships between them (eg a building is composed of parts, which are formed of walls, which have windows). Also, the CityGML semantic classes are structured into several modules, eg Building, Land Use, Water Bodies, and Transportation.

The geometry of the objects is realised with a subset of the geometry definitions in ISO19107 (ISO 2003) (only linear and planar primitives are however allowed), which also allows aggregations of geometries: a single building can for instance be modelled with a `CompositeSolid`. Furthermore, it is possible to attach textures, materials, and semantics to each of the surfaces of a 3D geometry. The geometry types of most geo-DBMSs do not allow us to represent such complex 3D geometries.

One of the main characteristics of CityGML is that it supports different levels of detail (LoDs) for each of the classes, which means that in theory for a single building, or a single tree, several geometries could be stored.

2.2 CityJSON + CityJSONFeature

CityJSON, with its latest version 1.1, is a JSON-based exchange format for the CityGML data model. It implements all the core modules of CityGML v3.0, and some

other modules are supported.[2] As explained in Ledoux et al. (2019), it was designed to improve the weaknesses of the XML-encoding of CityGML: large filesize, complex structure to manipulate, several ways to store a given characteristic, unfit for the web, etc.

Its geometry structure is similar to that of computer graphics formats (eg OBJ and STL), and allows us to compress by a factor of around 6 XML-encoded CityGML files. A thorough comparison shows that it is nearly as compact as formats that do not allow semantics, complex attributes, and coordinate reference systems (Praschl and Pointner 2022).

While the original files for CityJSON v1.0 were compact, one weakness was that files for large areas were not suitable for streaming. That is, to be able to process one object in a file, the client had to have in memory the whole file. Version 1.1 solved this issue by introducing a new type: `CityJSONFeature`, which represents *independently* one city object in a CityJSON file (eg a 'Building' or a 'Bridge'). The idea is to decompose a region into its many features, create several JSON objects of type `CityJSONFeature`, and stream them sequentially or store them in a JSON text sequence (IETF 2015).[3] This is conceptually the same as the *GeoJSON Text Sequences*[4] used for processing and exchanging large 2D GIS datasets.

We exploit this type to store independently each feature in one row of the database, although, as explained below, we modify the JSON structure slightly, split it over a few columns, and use indexes to accelerate performance.

2.3 NoSQL Databases

Nys and Billen (2021) developed a non-relational data model for CityJSON, and implement it in a JSON document database (MongoDB). They tested with one dataset (containing around 3500 city objects), and managed to reduce the size by a factor of 40% when compared to 3DCityDB. Only one query was benchmarked (retrieval of a random building with its attributes and single geometry), and their solution performed better than 3DCityDB (again around 40% faster). There is no information about how their solution performs with other queries that practitioners expect from a DBMS solution (eg the ones from Section 4.3).

2.4 3DCityDB

While the data model of CityGML could have been automatically mapped to relational tables, Yao et al. (2018) mentions that for 3DCityDB a semi-automatic method

[2] see details at: https://www.cityjson.org/citygml/v30/.

[3] *JSON Lines text file* is one possibility: https://jsonlines.org/.

[4] https://datatracker.ietf.org/doc/html/rfc7946#appendix-C.

was used to reduced the number of tables and the number of joins to perform queries (which will also typically reduce query times). The result is nonetheless, for v4.4, a total of 66 tables, many of which remain empty if, for instance, only buildings without appearances are stored.

As Pantelios (2022) mentions, the attributes for a given type are stored in different tables, depending on whether an attribute is prescribed by the CityGML data model or not. This complicates greatly the retrieval of information with SQL queries.

Interestingly, the 3D geometries are decomposed into their (semantic) surfaces, and each surface is stored in a separately row in a single table. Different flags are used to indicate whether a 3D geometry is a solid/watertight or a surface/not. While this approach allows to compactly store 3D volumetric geometries, in practice several joins are necessary to retrieve all the surfaces of a given feature. The volumetric 3D types available in PostGIS-SFCGAL (Borne et al. 2023) are also used (thus duplication of data).

Interestingly, PostGIS geometries for the footprint are not stored, only the 2D bounding box. This is in our opinion an odd choice because many queries on 3D city models are 2D queries (all buildings inside a given area, within a given distance, etc.).

3 Data Model, Software, and Engineering Decisions

3.1 Data Model

As shown in Fig. 1,

the cjdb data model is simple and akin to using the Simple Feature paradigm (OGC 2006), as PostGIS does. Each row in the table `city_object` stores a CityJSON city object (for instance a 'Building', a 'BuildingPart', a 'SolitaryVegetationObject', etc.) and the 'geometry' column stores a JSON array of geometries (Fig. 2 shows an example).

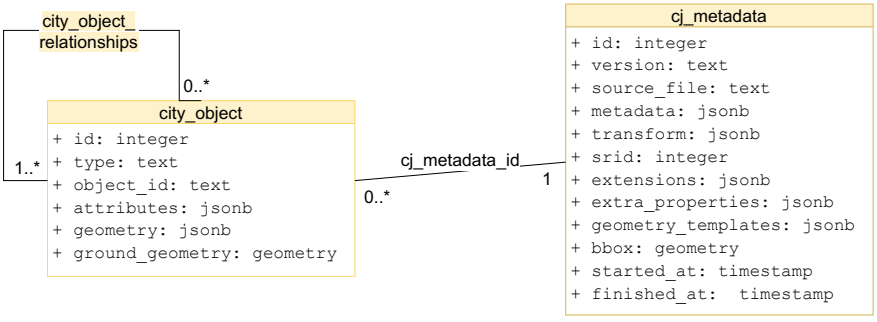

Fig. 1 UML diagram of cjdb

```
[
  {
     "type": "Solid",
     "lod": "2.2",
     "boundaries": [
        [ [[ [11.1, 22.6, 9.9], [16.21, 42.8, 19.9], ... ]
     ],
     "semantics": {
       "surfaces" : [
         { "type": "RoofSurface" },
         { "type": "WallSurface" },
         ...
       ],
       "values": [ [0, 1, ...] ]
     }
   }
 ]

```

Fig. 2 Example snippet stored in the 'geometry' column: an array of CityJSON geometries

The array is necessary because a given feature can have more than one geometry (eg the most obvious case is when several LoDs are stored), and it should be noticed that the JSON stored in the database is different from CityJSON: vertices are not stored separately and we replace the vertex identifiers in the "`boundaries`" property by their real-world coordinates.

For each row, we also store the identifier of the object, its type, the attributes (stored as `jsonb`), and we also store the ground geometry as a PostGIS 2D type (to be able to spatially index them in 2D, see below). Notice that the children objects are stored in separate rows (and not with their parent object), ie a 'Building' does not have its potential children in the same row, and each of them (eg a 'BuildingPart') is stored in a separate row.

The implemented version adds one more table: a table named `city_object_relationships` to store the relations between 'parents' and 'children' city objects (CityJSON has a flat structure and stores for instance a 'Building' at the same level as its 'BuildingPart' or 'BuildingInstallation' and links them). This table is added to improve query speed since we often need to process a parent with its children.

Finally, the table `cj_metadata` has one entry for each CityJSON file imported to the database and stores the following properties of CityJSON:

- the coordinate reference system (CRS);
- the precision used for storing the coordinates, used mostly when exporting data;
- the geometry templates (used for trees or bus stops);
- the CityJSON Extensions (to extend the base data model of CityJSON with application-specific types/attributes/semantics);
- the bounding box of the file.

3.2 Importer

To facilitate the usage of the cjdb data model we have implemented an importer and released it open-source under the MIT license: https://github.com/cityjson/cjdb.[5] The tool is developed in Python and has a command-line interface. Observe that because the data model is feature-centric, the importer will read JSONL files (JSON Sequences) and not CityJSON files. A CityJSON file can however be automatically converted into a list of features with the accompanying software *cjio*.[6]

The importer creates the 3 necessary tables, and populates them by parsing and modifying the CityJSON features according to the cjdb data model, as explained above.

Ground geometry extraction: As many queries on a 3D city model are typically performed in 2D, such as retrieving all objects within a certain area or selecting the object clicked upon in a 2D view, we have chosen to store the ground surface of each object as a 2D PostGIS geometry. This is achieved by iterating over all of the object's surfaces and selecting the horizontal ones with the lowest elevation. If multiple levels of detail (LoDs) are available, we select the lowest LoD. This addition enables us to perform rapid 2D spatial queries on the data without any joins (see Sect. 4). In comparison, performing the same spatial queries in 3DcityDB requires multiple joins. Alternatively, the enveloping bounding box, which in 3DCityDB is stored together with the object, can be used instead of the actual object geometry in order to perform spatial queries when the accuracy is not important.

Indexes: The data in the `city_object` table is expected to be retrieved mostly through spatial queries. Therefore, we decided to add a GiST index on the 'ground_ geometry' column and cluster the table based on that index. In order to improve query performance on the JSON columns we added a Generalized Inverted index (GIN), which is specialised for items with composite values. Additional full or partial BTree indexes can be applied during import, on specific attributes of the city objects, if the user expects that the table will be queried based on those attributes.

3.3 Exporter

The Python implementation of cjdb also offers an exporter. As is the case for 3DCityDB, a SQL query is used to filter which objects in the database should be exported (the identifiers in the table `city_object`). The output is a CityJSONL file (a sequence of `CityJSONFeatures`), which can automatically be converted to a CityJSON file with cjio.

[5] The version described in this article is v1.3.

[6] https://github.com/cityjson/cjio.

Table 1 The 3 datasets used for the benchmark

	# `Building`	# `BuildingPart`	LoDs present	# attributes
3DBAG	112673	110387	0/1.2/1.3/2.2	30
NYC	23777	0	2	3
Vienna	307	1015	2	7

Table 2 Import and export times, from/to CityJSONL

	3DCityDB		cjdb	
	Import	Export	Import	Export
3DBAG	6780	721	1260	412
NYC	273	161	23	25
Vienna	12	7	2	2.5

All times in seconds

4 Benchmark

To compare the performance of the cjdb data model against that of 3DCityDB, we created a benchmark dataset using data from 3 different countries; the Netherlands (3DBAG), Austria (Vienna), and USA (NYC). The 3DBAG dataset is composed of 100 tiles from the 3DBAG[7] (Peters et al. 2022); we randomly chose tiles from 3 cities in the Netherlands (Delft, Amersfoort, and Zwolle). The Vienna dataset covers the Austrian city whereas the NYC dataset covers a small part of central New York City; both datasets can be downloaded in CityJSON format from https://www.cityjson.org/datasets/. As can be seen in Table 1, all the datasets are building-centric, as is often the case with 3D city models, but here they are modelled differently and have different LoDs/sizes.

We imported each dataset in two different databases, one created with 3DCityDB and one cjdb, and below we compare them in terms of import/export time, data size, and data retrieval. Since cjdb is only available for PostgreSQL, we did not perform any tests on Oracle or PolarDB.

4.1 Import and Export Times

We compared the import time for all 3 datasets and we found that cjdb is considerably faster than 3DCityDB. As an example, the 100 tiles of the 3DBAG were imported in 21 min in cjdb whereas it took 113 min with 3DCityDB; see Table 2 for all details. This is expected, since the storage in the cjdb database is very close to the data model

[7] www.3dbag.nl.

of CityJSONL, while for 3DCityDB the geometries need to be processed and each surface to be stored separately. However, it is worth noting that in 3DCityBD the creation of the necessary tables is performed separately before the import, whereas cjdb creates the necessary tables at the time of import.

For the export, we exported each dataset to a CityJSON file (for 3DCityDB) and to a CityJSONL file (for cjdb). As shown in Table 2, cjdb is also generally faster.

Observe that those values are somewhat unfair to compare because the output is to a different format (and some time would be needed to convert from a CityJSONL to a CityJSON), because different languages are used for the importer/exporter (Java for 3DCityDB, Python for cjdb), and because 3DCityDB exporter is multi-threaded.

4.2 Database Size

One significant difference between 3DCityDB and cjdb is the database size they occupy. After importing the datasets, we measured the total size of each database, including the indexes and the TOAST tables (The Oversized Attribute Storage Technique), as shown in Table 3. We notice that cjdb occupies significantly less space that what 3DCityDB demands for the same data.

The main reason for this size difference lies in the different approaches to storing the features. In 3DCityDB, the different semantic surfaces (wall/roof/ground/etc) are considered separate city objects, contributing to the total number of rows in the `cityobject` table. In cjdb, the walls, roof and surfaces are considered intrinsic properties of each city object and thus remain in the `jsonb` format stored at the geometry column of each object. This decision has several advantages: reduction of the size of the `city_objects` table and faster and simpler queries where only city objects are concerned (eg Q1 and Q4 in Table 4). But there is also a major drawback: we cannot perform spatial queries on the geometries of specific semantic surfaces.

Another reason for the different database sizes is the different amount of indexes. We notice that cjdb implements significantly less indexes and as a result requires way less storage for them. In 3DCityDB, indexes account for almost half of the database's size.

One more noticeable difference is that, since cjdb stores attributes and geometries in `jsonb` format, it requires more space for TOAST tables. TOAST (*The Oversize Attribute Storage Technique*) is a PostgreSQL mechanism which controls the size of the data stored in a field. If the data exceeds the maximum allowed limit, TOAST breaks the too-wide field values down into smaller pieces, and stores them out-of-line in a TOAST table. Columns of type `jsonb` tend to carry quite wide values and often utilise the TOAST tables. Thus when measuring the cjdb database size we need to take the extra toast tables into account. However, even with the extra TOAST tables, the total database size remains around ten times smaller than that of 3DCityDB.

Finally, it should be noticed that the 3DCityDB size should in reality be larger than the number we obtain: since its data model allows only but one geometry per LoD and that the refined LoDs by Biljecki et al. (2016) are not supported, the 3DCityDB

Table 3 Database size comparison for the 3 datasets, all values in MB

	3DCityDB				cjdb			
	Tables	Indexes	TOAST	Total	Tables	Indexes	TOAST	Total
3DBAG	5463	4322	112	9898	257	57	755	1070
NYC	590	735	0.5	1326	26	4	25	54
Vienna	30	42	0.5	73	1.5	0.5	4	6

Table 4 The 8 queries we used for the benchmark

Q1	Retrieve the ids of all buildings based on one attribute (roof height higher than 20 m)
Q2	Retrieve all buildings within a 2D bounding box
Q3	Retrieve building intersecting with a 2D point
Q4	Retrieve the number of parts for each building
Q5	Retrieve all buildings having a specific LoD geometry
Q6	Add new ‘footprint_area’ attribute
Q7	Update ‘footprint_area’ attribute by adding 10 m
Q8	Delete ‘footprint_area’ attribute

importer selects either the LoD1.2 or LoD1.3 from the inputs, and thus one LoD is missing. In cjdb, all available LoDs are stored together in the ‘geometry’ column.

4.3 Data Retrieval

The data retrieval comparison was performed based on the execution time of SQL queries which aim to retrieve the same data from both databases. PostgreSQL heavily relies on caching, therefore the queries below were run several times to ensure the cache was warm.

We performed 8 queries that we believe are representative of what a typical practitioner (or server hosting data to be downloaded) would need.

Those are listed in Table 4. The exact SQL queries we used for the 3DBAG dataset are listed in Appendix 6; similar queries were used for the other 2 datasets.

Q1. Query based on attributes: 3DCityDB offers a list of predefined building attributes within the `building` table, which include ‘year_of_construction’ and ‘roof_type’—attributes that are not in this list are stored in the table `cityobject_genericattrib`. Cjdb on the other hand offers more flexibility since all the attributes remain in JSON format in the attributes column, regardless of the attribute name.

Since none of our datasets have attributes from the 3DCityDB’s predefined list, we decided to compare the attribute-based data retrieval for both databases based on non-listed attributes. In this specific example, we queried all the buildings with

roof height ('h_dak_max' for BAG 'HoeheDach' for Vienna) higher than 20 m. The New York dataset was not taken into account for this query, since there is no specific attribute about the roof height.

For cjdb no join is necessary since the attributes are stored together with the city object but the equivalent in 3DCityDB requires a join between the `city_object` and the `cityobject_genericattrib` tables. As shown in Table 5, cjdb it is faster than 3DcityDB for Vienna but performs almost the same as 3DcityDB for the 3DBAG dataset. This is related to both the size of the dataset and the size of the attributes column in cjdb; the bigger the `jsonb` column, the slower the query, since the information stored in the TOAST tables will need to be retrieved and decompressed.

Q2 & Q3. 2D Spatial queries: The spatial queries in 3DCityDB tend to be complicated since the geometries of the objects are stored in other tables and require joins to retrieve them. As an example, in order to find all buildings within a certain 2D bounding box, their ground surfaces had to be retrieved from the `surface_geometry` table. An alternative but less accurate solution would be to use the bounding box geometry of the object, which is stored in the city objects' table. Cjdb does not require any join to retrieve the same data, since the `city_objects` table contains the ground geometries of the objects. As a result the cjdb query is significantly faster than the equivalent of 3DCityDB, as shown in Table 5. We observe similar speed differences with other spatial queries, such as retrieving the building intersecting with a given point in 2D (Q3).

Q4. Number of parts query: We compared how the two databases perform with the retrieval of the number of building parts per building. For simplicity we considered only first level children for each building and we also require the buildings without any parts to be part of the result. Both queries require a single join and their execution times varies depending on the size of the dataset. When the 3DBAG dataset is examined, cjdb seems to be slower than 3DCityBD but for the other datasets, cjdb is faster. However it is worth mentioning that for cjdb the join with the city object table could be skipped and the number of parts could be retrieved with a single aggregation query on the `city_object_relationships` table, but only for the objects which have parts.

Q5. LoD-based query: While one strength of CityGML is that many LoDs of one city object can be stored with the object, exporting a given one to perform an analysis is useful in practice. We therefore tested a query to obtain all building ids with LoD1 (in 3DCityDB) and LoD1.2 (in cjdb) for the 3DBAG dataset and LoD2 for the other 2 datasets. Cjdb performs slower for all datasets; this is probably due to the large size of the 'geometry' column which is in `jsonb` format and therefore requires joining to the TOAST tables.

It should be noticed that if the geometry of the building needs also to be retrieved, the equivalent query for 3DCityBD requires joins which significantly lower the speed.

Q6, Q7 & Q8. INSERT/UPDATE/DELETE attribute queries: We also compared how the databases perform when adding, modifying or deleting an attribute of a building. When it comes to adding new attributes to the database (Q6), cjdb performs considerably slower than 3DcityDB. This can be attributed to the different structure

Table 5 Average execution times for the benchmark queries (all times in ms)

	3DBAG		NYC		Vienna	
	3DCityDB	cjdb	3DCityDB	cjdb	3DCityDB	cjdb
Q1	290	285	–	–	22	2
Q2	172	1.4	59	0.3	14	0.4
Q3	1.0	0.2	0.4	0.1	0.2	0.1
Q4	418	478	240	46	15	2
Q5	343	1877	217	392	15	34
Q6	3981	11 660	1135	1425	15	10
Q7	10 796	10 903	2333	1161	15	9
Q8	4393	10 984	1040	682	59	8

of the databases: new attributes in 3DCityDB can simply be inserted as rows in the relevant table, whereas in cjdb the `jsonb` must be modified. However, when it comes to updating existing attributes (Q7), the speed of cjdb is similar for datasets having many attributes, and faster for datasets having few attibutes. Deleting attributes (Q8) follows a similar behaviour. Generally, it can be noticed that the number of attributes and the size of the dataset can significantly affect queries on the `jsonb` columns.

5 Conclusions and Future Work

The cjdb project started with the goal of creating a simpler and leaner alternative to 3DCityDB for web servers, allowing users to efficiently store and retrieve 3D city models. Our data model follows the Simple Feature paradigm and has only 3 tables (instead of 66 for 3DCityDB). This is achieved by maintaining the structure of CityJSON and storing JSON directly in the database, using the PostgreSQL type `jsonb`. Because the structure of the data model is close to that of CityJSON files, we can significantly improve the import/export times to/from a database. Furthermore, as we have shown, our simple model is around ten times more compact and it offers retrieval speed comparable to those of 3DCityDB. More specifically we notice that the cjdb performs better when it comes to 2D spatial queries but it performs slower when the `jsonb` columns need to be altered (up to 3 times slower). We also notice that the query speed on the `jsonb` columns is significantly affected by the size of the dataset; the more the attributes in the `jsonb` columns get, more time is required to parse them.

Figure 3 shows the 8 queries, each time have been normalised (we divided the time of cjdb by that of 3DCityDB).

It should be mentioned that the data model of 3DCityDB is more generic and can be implemented with 3 different DBMSs (and not only PostgreSQL). It also allows us to query semantic surfaces, something that is currently not possible with the cjdb model

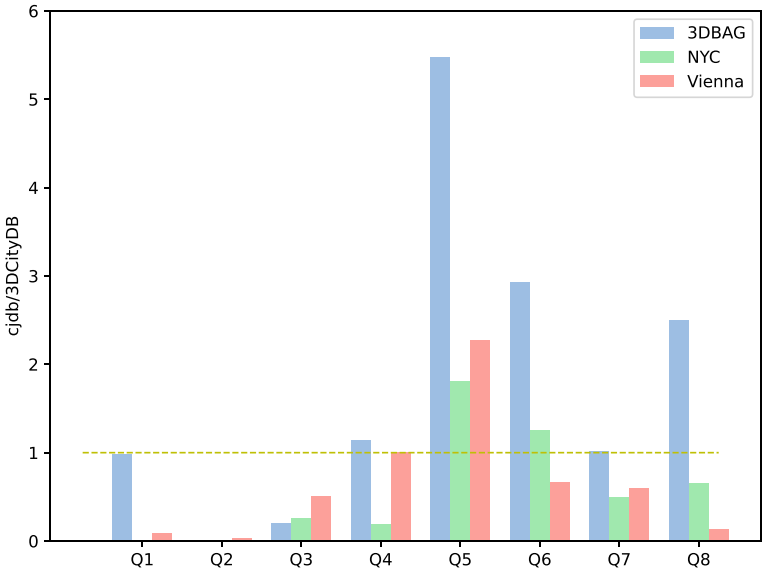

Fig. 3 Data retrieval comparison between cjdb and 3DCityBD for the 8 queries using the 3DBAG dataset. The y-axis corresponds to (cjdb/3DCityDB); a bar lower than the yellow line (located at 1.0) means that cjdb has a faster query time, a bar higher than 1.0 means a slower query time

because it stores the semantic surfaces in `jsonb` format in the 'geometry' column of each city object. However, we plan to remedy to the situation by implementing database functions to extract the geometries from semantic surfaces. Furthermore, at the moment, retrieving data from cjdb requires the user to parse the `jsonb` and extract the necessary information, something that is far from being optimal. We plan in the near future to implement some helper plugins in mainstream open-source products (eg QGIS) to be able to view and query cjdb, similar to what is currently being built for 3DCityDB.[8]

While this was not previously discussed, it should be mentioned that CityJSON Extensions are supported by the cjdb data model. This means that extra functions to dynamically extend the data model (as required for 3DCityDB, see Yao and Kolbe (2017)) are not necessary. This is because CityJSON Extensions, unlike CityGML Application Domain Extensions (ADEs), are constrained to follow the structure and rules of other city objects (see Ledoux et al. (2019) for more details).

We also plan to add support for textures and material, at the moment the information is simply stored in the JSON of each geometry, but as is the case for semantic surfaces, we will add database functions to allow users to query and update those.

[8] https://github.com/tudelft3d/3DCityDB-Tools-for-QGIS.

6 The 8 SQL Queries Used for the Benchmark for the 3DBAG Dataset

Q1

3DCityDB

```
SELECT
    gmlid
FROM
    3dcitydb.cityobject co
    JOIN 3dcitydb.cityobject_genericattrib coga
    ON co.id = coga.cityobject_id
WHERE
    coga.attrname = 'h_dak_max'
    AND coga.realval > 20;
```

cjdb

```
SELECT
    object_id
FROM
    cjdb.city_object
WHERE
    (attributes -> 'h_dak_max') :: float > 20;
```

Q2

3DCityDB

```
SELECT
    c.gmlid,
    sg.geometry
FROM
    3dcitydb.thematic_surface AS ts
    LEFT JOIN 3dcitydb.surface_geometry AS sg
    ON ts.lod2_multi_surface_id = sg.parent_id
    LEFT JOIN 3dcitydb.cityobject AS c
    ON ts.building_id = c.id
WHERE
    ST_Contains(
        ST_MakeEnvelope(
            85400,
            446900,
            85600,
            447100,
            7415
        ),
        sg.geometry
    )
    AND ts.objectclass_id = 35;
```

cjdb

```
SELECT
    object_id,
    ground_geometry
FROM
    cjdb.city_object
WHERE
    TYPE = 'Building'
    AND ST_Contains(
        ST_MakeEnvelope(
            85400,
            446900,
            85600,
            447100,
            7415
        ),
        ground_geometry
    );
```

Q3

3DCityDB

```
SELECT
    c.gmlid,
    sg.geometry
FROM
    3dcitydb.thematic_surface AS ts
    LEFT JOIN 3dcitydb.surface_geometry AS sg
    ON ts.lod2_multi_surface_id = sg.parent_id
    LEFT JOIN 3dcitydb.cityobject AS c
    ON ts.building_id = c.id
WHERE
    sg.geometry && st_setsrid(ST_MakePoint(85200,
    446900), 7415) AND ts.objectclass_id = 35;
```

cjdb

```
SELECT
    object_id,
    ground_geometry
FROM
    cjdb.city_object
WHERE
    "type" = 'Building'
    AND ground_geometry
    && st_setsrid(ST_MakePoint(85200, 446900),
    7415);
```

Q4

3DCityDB

```
SELECT
    cj.gmlid AS building_id,
    count(b.id) AS number_of_parts
FROM
    3dcitydb.cityobject cj
    LEFT JOIN 3dcitydb.building b
    ON cj.id = b.building_parent_id
WHERE
    cj.objectclass_id = 26
GROUP BY
    cj.id;
```

cjdb

```
SELECT
    co.object_id AS building_id,
    COUNT(cor.child_id) AS number_of_parts
FROM
    cjdb.city_object co
    LEFT JOIN cjdb.city_object_relationships cor
    ON co.object_id = cor.parent_id
WHERE
    co.type = 'Building'
GROUP BY
    co.object_id;
```

Q5

```
SELECT
    co.gmlid
FROM
    3dcitydb.cityobject co
    JOIN 3dcitydb.building b
    ON co.id = b.id
WHERE
    b.lod1_solid_id IS NOT NULL;
```

```
SELECT
    object_id
FROM
    cjdb.city_object
WHERE
    geometry::jsonb @> '[{"lod": 1.2}]'::jsonb;
```

Q6

```
INSERT INTO
    3dcitydb.cityobject_genericattrib
(attrname, datatype, realval, cityobject_id)
SELECT
    'footprint_area',
    3,
    ST_Area(cityobject.envelope),
    cityobject.id
FROM
    3dcitydb.cityobject
WHERE
    cityobject.objectclass_id = 26;
```

```
UPDATE
    cjdb.city_object
SET
    attributes = jsonb_set(
        attributes :: jsonb,
        '{footprint_area}',
        to_jsonb(ST_Area(ground_geometry))
    ) :: json
WHERE
    TYPE = 'Building';
```

Q7

```
UPDATE
    3dcitydb.cityobject_genericattrib
SET
    realval = realval + 10
WHERE
    attrname = 'footprint_area';
```

```
UPDATE
    cjdb.city_object
SET
    attributes = jsonb_set(
        attributes :: jsonb,
        '{footprint_area}',
        to_jsonb(
            CAST (
                jsonb_path_query_first(
                    attributes :: jsonb,
                    '$.footprint_area'
                ) AS real
            ) + 10.0
        )
    ) :: json
WHERE
    TYPE = 'Building';
```

Q8

```
DELETE FROM
    3dcitydb.cityobject_genericattrib
WHERE
    attrname = 'footprint_area';
```

```
UPDATE
    cjdb.city_object
SET
    attributes = jsonb_set_lax(
        attributes :: jsonb,
        '{footprint_area}',
        NULL,
        TRUE,
        'delete_key'
    ) :: json
WHERE
    TYPE = 'Building';
```

References

Biljecki F, Stoter J, Ledoux H, Zlatanova S, Çöltekin A (2015) Applications of 3D city models: state of the art review. ISPRS Int J Geo-Inf 4(4):2220–9964. https://doi.org/10.3390/ijgi4042842

Biljecki F, Ledoux H, Stoter J (2016) An improved LOD specification for 3D building models. In: Computers, environment and Urban systems, pp 25–37. https://doi.org/10.1016/j.compenvurbsys.2016.04.005

Borne M, Mercier H, Mora V, Courtin O (2023) SFCGAL. https://oslandia.gitlab.io/SFCGAL/, last visited: 01 May 2023

Gröger G, Plümer L (2012) CityGML—interoperable semantic 3D city models. ISPRS J Photogram Rem Sens 71:12–33

IETF (2015) JavaScript Object notation (JSON) text sequences. https://datatracker.ietf.org/doc/html/rfc7464

ISO (2003) ISO 19107:2003: geographic information—spatial schema. International Organization for Standardization

Kutzner T, Chaturvedi K, Kolbe TH (2020) CityGML 3.0: new functions open up new applications. PFG J Photogram Rem Sens Geoinf Sci 88(1):43–61

Ledoux H, Ohori KA, Kumar K, Dukai B, Labetski A, Vitalis S (2019) CityJSON: a compact and easy-to-use encoding of the CityGML data model. Open Geospatial Data Softw Stand 4(4). https://doi.org/10.1186/s40965-019-0064-0

Nys GA, Billen R (2021) From consistency to flexibility: a simplified database schema for the management of CityJSON 3D city models. Trans GIS. https://doi.org/10.1111/tgis.12807

OGC (2006) OpenGIS implementation specification for geographic information—simple feature access. Open Geospatial Consortium inc., document 06-103r3

OGC (2007) Geography markup language (GML) encoding standard. Open Geospatial Consortium inc., document 07-036, version 3.2.1

OGC (2012) OGC city geography markup language (CityGML) encoding standard. Open Geospatial Consortium inc., document 12-019, version 2.0.0

OGC (2021a) CityJSON community standard 1.0. Open Geospatial Consortium inc., document 20-072r2, version 1.0

OGC (2021b) OGC city geography markup language (CityGML) part 1: conceptual model standard. Open Geospatial Consortium inc., document 20-010, version 3.0.0, available at https://docs.ogc.org/is/20-010/20-010.html

Pantelios K (2022) Development of a QGIS plugin for the CityGML 3D city database. Master's thesis, MSc thesis in Geomatics, Delft University of Technology

Peters R, Dukai B, Vitalis S, van Liempt J, Stoter J (2022) Automated 3D reconstruction of LoD2 and LoD1 models for all 10 million buildings of the Netherlands. Photogram Eng Rem Sens 88(3):165–170

Praschl C, Pointner A (2022) Evaluation of file formats in the context of geo-referenced 3d geometries. GitHub repository at https://github.com/FHOOEAIST/geofiles, https://doi.org/10.5281/zenodo.5376368

Ramakrishnan R, Gehrke J (2001) Database management systems. McGraw-Hill Science/Engineering/Math

Yao Z, Kolbe TH (2017) Dynamically extending spatial databases to support CityGML application domain extensions using graph transformations. In: Kersten TP (ed) Kulturelles Erbe erfassen und bewahren—Von der Dokumentation zum virtuellen Rundgang, 37, vol 26. Wissenschaftlich-Technische Jahrestagung der DGPF, DGPF, Würzburg, Germany, pp 316–331

Yao Z, Nagel C, Kunde F, Hudra G, Willkomm P, Donaubauer A, Adolphi T, Kolbe TH (2018) 3DCityDB—a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. Open Geospatial Data Softw Stand 3(2)