CJDB
A simple and efficient data model to store CityJSON in a database

GEO1011 Synthesis Project

Leon Powałka (5605822)
Yuduan Cai (5625483)
Siebren Meines (4880412)
Chris Poon (4355938)
Yitong Xia (5445825)
Lan Yan (5471648)

1st supervisor: Hugo Ledoux
external supervisor: Balázs Dukai

November 17, 2022

Contents

1 Overview

A 3D city model uses three-dimensional geometries to represent and model urban environments, in which the building model is the key feature [1]. With the development of computer and data collection technologies, 3D city models are gaining growing capacities regarding storing rich information. This makes 3D city models more potentially useful than ever in the urban application domain [1].

When implementing 3D city models, the CityGML model is currently the most frequently used standard. It is being used by cities all over the world. CityJSON is an encoding for a subset of the OGC CityGML data model [3]. It is a JSON-based data exchange format for digital 3D models of cities and landscapes, and it is easy for various operations. Due to the uniqueness of its structure, it is necessary to design corresponding data models to store CityJSON files in the database, to make querying and updating data within the database easy and convenient.

There are already some open-source solutions for storing CityJSON files in the database, 3DCityDB is one of them [7]. 3DCityDB can store, manage and visualize data well, and it is open source. But the database design is very complex: for a tile of 3D BAG data, 3DcityDB uses a total of 66 tables to store data. The structure of the data model results in difficulties for database users to understand the imported data, and potentially leads to non-optimal operation performance when retrieving data for urban applications. Based on the drawbacks of the existing DBMS when dealing with the CityJSON data format, this project aims to develop a Postgres data model that can store CityJSON files simply and efficiently. The developed Postgres data model (CJDB) has a simpler table structure and data model design, a CityJSON data importer, and a interactive API user interface.

After going through this document (data model, importer and API section), the potential users will have the ability to:

- Import CityJSON files into a Postgres database,
- Perform queries on imported data,
- Perform operations using CJDB API.

In addition, by reading the benchmarking section, the users can gain an overview of the CJDB's performance over the 3DcityDB's.

The CJDB project is open-sourced, available on GitHub page.

The CJDB project will be potentially further developed by 3D geoinformation group of TU Delft and 3DGI.

2 Data Model

The CJDB data model is designed to store CityJSONL files in a Postgres database. After reading this section, the user will:

1) Understand the table structure of CJDB model.

2) Understand the indexing possibilities within the database, and their effects on operations.

2.1 Table Structure

The conceptual data model contains two main tables. The import_meta table for storing imported files' information, e.g. name or metadata of the source file. The cj_object table for storing city objects.

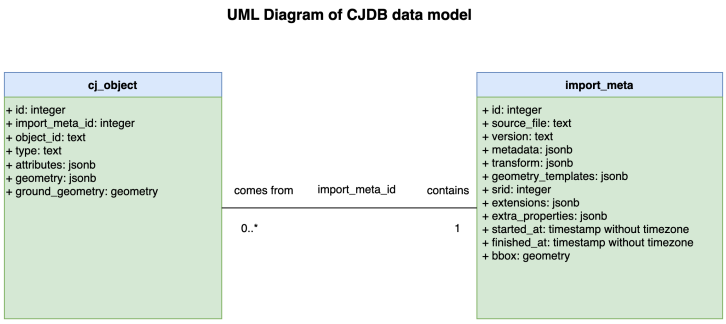


Figure 1: UML Diagram of CJDB data model

The physical data model adds one more table on the conceptual data model: the so called family table to store relations between city objects, e.g the parent-children relationship. This table is added to achieve higher querying speed when selecting objects by their parent/child relationship. Example of this would be: "give me all the objects which are children of X".

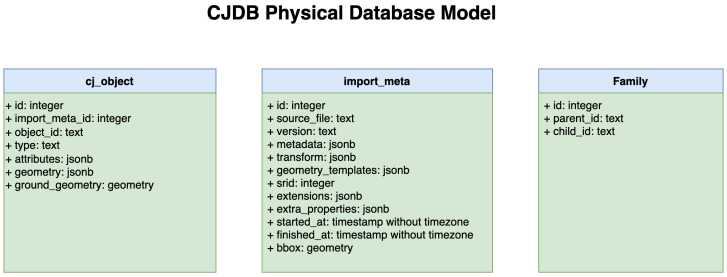


Figure 2: CJDB Physical Database Model

2.1.1 import_meta

The import_meta table stores information from imported files, e.g. name or metadata of the source file. The following section describes its attributes.

id: import_meta record's index within the table.

source_file: name of the source file.

version: CityJSON version used.

metadata: CityJSON metadata object, a JSON object describing the creator, dataset extent or coordinate reference system used, etc.

transform: CityJSON transform object, a JSON object describing how to decompress the integer coordinates of the geometries to obtain real-world coordinates.

geometry_templates: CityJSON geometry-templates object, a JSON object containing the templates that can be reused by different City Objects (usually for trees).

srid: Coordinate reference system (CRS) of the imported city objects in the database. If not specified when importing, the CRS will be the same with the source file's CRS. If specified when importing, the CRS will be the specified CRS.

extensions: CityJSON Extensions, a JSON file that documents how the core data model of CityJSON is extended.

extra_properties: extraRootProperties, a JSON object with added new properties at the root of the imported document.

started_at: importing start time.

finished_at: importing finish time. 'null' if not finished.

Bounding box: bounding box is taken from the geographicExtent object from the metadata section. If the required information is not present, the value is set to null.

2.1.2 cj_object

The cj_object model stores individual city objects, for instance buildings, roads, or bridges. Its attributes are described below. The attributes and geometry are separated into two JSONB column for query optimization purpose.

id: city object's index within the table.

import_meta_id: the source file id of the city object, foreign key to the id column of import_meta table.

object_id: the identification string of the city object (e.g. NL.IMBAG.Pand.0503100000000033-0).

type: type of the city object (e.g. Building, BuildingParts, etc.).

attributes: CityJSON attributes, a JSON object that describes attributes of the city object (e.g. roof type, area, etc.).

geometry: CityJSON geometry, a JSON object that describes the geometry of the city object.

ground_geometry: ground geometry of the city object, in geometry type.

2.1.3 Family

The family model stores the relations between city objects, to optimize queries on the database.

id: family object's index within the database.

parent_id: the identification id of the parent object.

child_id: the identification id of the child object.

2.2 Indexing

Indexing certain columns or expressions speeds up querying on them.

The GIN index is indexed on LoD by default, to increase the query speed on geometries, considering the fact that geometry can be a very large column. The GIN index is automatically created when importing data into the database by the importer.
The users can use B-tree partial index on attributes. It will slightly increase the query speed, but use more space. The users can specify attributes they want to create when using the importer.

3 Importer: cj2pgsql

cj2pgsql is a Python based importer of CityJSONL files to a PostgreSQL database.

3.1 Command Line Interface (CLI) Usage

3.1.1 CLI Instructions

The user can visit the CLI instruction web page for the CLI instructions.

3.1.2 Quickstart Example

Sample CityJSON data can be downloaded from 3D BAG download service.

Then, having the CityJSON file, a combination of cjio and cj2pgsql is needed to import it to a specified schema in a database.

1. Convert CityJSON to CityJSONL

```
cjio --suppress_msg tile_901.json export jsonl stdout > tile_901.jsonl
```

2. Import CityJSONL to the database

```
PGPASSWORD=postgres cj2pgsql -H localhost -U postgres
-d postgres -s CJDB -o tile_901.jsonl
```

Alternatively steps 1 and 2 in a single command:

```
cjio --suppress_msg tile_901.json export jsonl stdout | cj2pgsql
-H localhost -U postgres -d postgres -s CJDB -o
```

The metadata and the objects can then be found in the tables in the specified schema ('CJDB' in this example).

Password can be specified in the 'PGPASSWORD' environment variable. If not specified, the app will prompt for the password.

3.2 cj2pgsql explanations

3.2.1 Model assumptions

The cj2pgsql importer loads the data in accordance with a specific data model, which is also shared with the CJDB_api. More documentation can be found on data model section.

3.2.2 Indexes

Some indexes are created by default, refer to data model section.

Additionally, the user can specify which CityObject attributes are to be indexed with the -x/–attr-index or -px/–partial-attr-index flag. The second option uses a partial index with a "not null" condition on the attribute. This saves disk space when indexing an attribute that is not present among all the imported CityObjects. This is often the case with CityJSON, because in a single dataset there can be different object types, with different attributes.

3.2.3 What is a City Model? How to organize CityJSON data from various sources?

The definition and scope of the City Model are for the users to decide. It is recommended to group together semantically coherent objects, by importing them to the same database schema.

While the static table structure (columns don't change) does support loading any type of CityJSON objects together, the data becomes harder to manage for the users. Example of this would be having different attributes for the same CityObject type (which should be consistent for data coming from the same source). It is advised to put city objects that have very different attributes into different schema or different database.

3.2.4 Types of input

The importer works only on CityJSONL files. Instructions on how to obtain such a file from a CityJSON file can be found on corresponding cijo documentation.

The importer supports three kinds of input:

- a single CityJSONL file
- a directory of CityJSONL files (all files with `jsonl` extensions are located and imported)
- STDIN using the pipe operator: cat file.jsonl | cj2pgsql ...

3.2.5 Coordinate Reference Systems

The cj2pgsql importer does not allow inconsistent CRS (coordinate reference systems) within the same database schema. For storing data in separate CRS using multiple schemas are required.

The data needs to be either harmonized beforehand, or the '-I/–srid' flag can be used upon import, to reproject all the geometries to the one specified CRS. Specifying a 2D CRS (instead of a 3D one) will cause the Z-coordinates to remain unchanged. But reprojections slow down the import significantly. Source data with missing "metadata"/"referenceSystem" cannot be reprojected due to unknown source reference system.

3.2.6 3D reprojections

Pyproj is used for CRS reprojections. It supports 3D CRS transformations between different systems. However, sometimes downloading additional grids is required. The importer will attempt to download the grids needed for the reprojection, with the following message:

Attempting to download additional grids required for CRS transformation. This can also be done manually, and the files should be put in this folder:

pyproj_directory

If that fails, the user will have to download the required grids and put them in the printed pyproj_directory themselves.

3.2.7 CityJSON Extensions

If CityJSON Extensions were present in the imported file, they can be found listed in the extensions column in the import_meta table.

The CityJSON specifications mention 3 different extendable features, and the cj2pgsql importer deals with them as follows:

1. Complex attributes:

   No action is taken. These attributes end up in the 'attributes' JSONB column. Querying by complex attributes values is not supported in the CJDB_api as of v0.1.

2. Additional root properties:

   Additional root properties are placed in the 'extra properties' JSONB column in the 'import_meta' table.

3. Additional CityObject type:

   Additional CityObject types are appended to the list of allowed CityJSON objects. cat file.jsonl | cj2pgsql ...

3.2.8 CityJSON GeometryTemplate

Geometry templates are resolved for each object geometry, so that the object in the table ends up with its real-world coordinates (instead of vertex references or relative template coordinates).

3.2.9 Data validation

The importer does not validate the structure of the file. It is assumed that the input file is schema-valid (CityJSON validator). It sends out warnings when:

1. there appear CityObject types defined neither in the main CityJSON specification nor any of the supplied extensions.
2. the specified target CRS does not have the Z-axis defined.
3. the source dataset does not have a CRS defined at all.

3.3 Running with local code

Create pipenv environment in repository root:

```
pipenv install
```

Run the importer:

```
PYTHONPATH=$PWD pipenv run python cj2pgsql/main.py –help
```

3.4 Running tests

Test cases for Pytest are generated based on the CityJSONL files in:

```
- cj2pgsql/test/files
```

And the argument sets defined in the file:

```
- cj2pgsql/test/inputs/arguments
```

Where each line is a separate argument set.

The tests are run for each combination of a file and argument set. To run them locally, the cj2pgsql/test/inputs/arguments file has to be modified.

Install pytest first using the following command:

```
pip3 install pytest
```

Then, in repository root:

```
pytest cj2pgsql -v
```

or, to see the importer output:

```
pytest cj2pgsql -s
```

4 API

In order to demonstrate a use case, and for potential users to test CJDB, the CJDB API was made. The API aims to show that our data model is user-friendly and efficient. Future users of CJDB, that wish to use the functionality provided by our API, can install it with pip. Users that wish to adapt/ customize the API could clone the CJDB API code from github, and modify it to their needs, to use CJDB.

4.1 API documentation

4.1.1 Usage

This API can be used to work with CJDB in a browser.

4.1.2 Run

- Using CLI

```
CJDB_api -C


postgresql://postgres:postgres@localhost:5432/CJDB -s CJDB2 -p 5000
```

- Using the repository code

```
pipenv run python CJDB_api/run.py -C


postgresql://postgres:postgres@localhost:5432/CJDBv5 -s CJDB2 -p 5000
```

- API will be launched at localhost:

```
http://localhost:5000/api
```

- Arguments
  - 'C' Data base connection string, in format of:

    "postgresql://<user>:<password>@<host>:<port>/<database>"

  - 'p' Port to connect API to. Default = 8080
  - 's' Database schema, where the CityObjects are kept
  - 'd' To run in debug mode.
- Run run.py to launch the API in your localhost

```
http://localhost:5000/api
```

4.1.3 Functionality

- Simple Queries
  - Show
    - Show the first x amount of entries in the database. For instance:
      http://localhost:5000/api/show/5
  - QueryByAttribute
    - Look for specific values of attributes, For instance:
      http://localhost:5000/api/select/id/5
      http://localhost:5000/api/select/type/Building
  - GetInfo
    - Get the attribute info of an object, given the object id. For instance to get the geometry of NL.IMBAG.Pand.0175100000014322:

      http://localhost:5000/api/info/geometry/NL.IMBAG.Pand.0503100000000021
  - FilterAttributes
    - `Filter in the attributes with operators >, < , =. For Instance:`

      `http://localhost:5000/api/filter/data_area/bigger/40`
  - GetChildren
    - Get the children of an object. For instance:
      http://localhost:5000/api/children/NL.IMBAG.Pand.0503100000000021
  - GetParent
    - Get the parent of an object. For instance:
      http://localhost:5000/api/parent/NL.IMBAG.Pand.0503100000000021
- Complex Queries
  - QueryByGroundGeometry
    - Returns all objects within a given bounding box. For instance:

      `http://localhost:5000/api/ground_geometry/(81400,451400,81600,451600)`
  - QueryByPoint
    - Returns the object located at a given point. For instance:

      `http://localhost:5000/api/point/(81402.6705,451405.4224)`
  - `CQL_query`
    - Allows the usage of common query language (CQL) to chain queries. For instance:

      `localhost:5000/api/cql?CQL_FILTER=type="Building"ANDdata_area>30`
- Add/Update

  - AddAttribute
    * Add an attribute to an object, or to all, by replacing the

      `object_id`

      with "all". For instance:

      `http://localhost:5000/api/add/NL.IMBAG.Pand.0503100000000021/attrib/10`

  - UpdateAttrib
    * update the value of an attribute from an object, or to all objects, by replacing the

      `object_id`

      with "all". For instance:

      `http://localhost:5000/api/update/all/attrib/12`

- Deletion
  - AddAttribute
    * Delete an attribute from an object, or from all, by replacing the

      `object_id`

      with "all". For instance:

      `http://localhost:5000/api/del/NL.IMBAG.Pand.0503100000000021/attrib`

  - DelObject
    * Delete an object from the data. If the object has children, these are also deleted.

      `http://localhost:5000/api/delObject/NL.IMBAG.Pand.0503100000000021`

5 Benchmarking

To evaluate and improve the performance of our data model CJDB, we will have to benchmark it. Our implementation of benchmarking consists of a collection of SQL queries that we defined and the data we load into the databases. The performance of CJDB is measured by comparing the execution time of the SQL queries in CJDB and in 3DCityDB. The benchmarking data set consisted of several subsets of 3D BAG.

First, we used one tile of the 3D BAG dataset to create the queries we used for the benchmarking. One tile was used to ensure the queries would run fast, in this stage we paid less attention to the speed comparison, but we made sure that the queries returned the same results or performed similar tasks. The following tasks and criteria were defined:

- Import time
- File size

  - A comparison in storage (GB) for the data models.

- Altering objects in the city model:
  - Adding attributes:
    * Add an attribute roof_area for all buildings.
  - Updating attributes:
    * Add 10 metres to the roof_area attribute for all buildings.
  - Removing attributes:
    * Remove the roof_area attribute for all buildings.

- Queries based on attributes:
  - query building with ID "id".

- Query on date/-time of a city object:
  - Show all buildings made after the year 2000.

- Query based on location:
  - Query all buildings within a certain bounding box.
  - Query all buildings contained at a certain point

- Query based on parent-child relation:
  - Query number of children
  - Query buildings with certain children
  - Query buildings with certain parents

- Query based on LoD
  - Query geometry by LoD
  - Query certain LoD of certain building

The SQL queries for 3DCityDB and CJDB from these tasks and critera can be found at Section 5.1.1 and Section 5.1.2 respectively.

Second, we used a subset of 3D BAG that consisted of 21 tiles which covers Delft. With this 'Delft' dataset and the collection of queries we tested different versions of CJDB, see Section 5.2.1. And other design choices, see Section 5.2.2. And also, compared the query performance between CJDB and 3DCityDB. see Section 5.3.

Lastly, we wanted to test CJDB and compare it to 3DCityDB on our server using the whole dataset of 3D BAG which consisted of 8000+ tiles. This proofed to be too big for our server to handle, so we first cut the input data set down for both CJDB and 3DCityDB to 4000 3D BAG tiles, which was still too big. We then cut down the input data to be 2000 tiles, the database size of CJDB and 3DCityDB with the imported 2000 tiles filled our server to the point it was 100% full. Which did not leave any space for us to run queries, since the database sizes expand with running queries. So we finally decided to use 500 tiles for the tests and comparison between CJDB and 3DCityDB on the server, see Section 5.3.

5.1 Benchmarking queries

5.1.1 3DCityDB

1. The import time is timed inside the Importer itself for 3DCityDB.
2. To determine the size of a database (replace *dname* with the name of the database):

```
SELECT pg_size_pretty( pg_database_size('dbname') );
```

3. Add an attribute footprint_area for all buildings:

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

4. Add 10 metres to the footprint_area attribute for all buildings:

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

5. Remove the footprint_area attribute for all buildings:

```
DELETE FROM cityobject_genericattrib
WHERE attrname = 'footprint_area';
```

6. Query building with ID 'NL.IMBAG.Pand.0629100000020136':

```
SELECT * FROM citydb.cityobject
WHERE gmlid = 'NL.IMBAG.Pand.0629100000020136'
ORDER BY id ASC
```

7. Show all buildings made after the year 2000:

```
SELECT * FROM citydb.cityobject
WHERE cityobject.id IN (
    SELECT cityobject_genericattrib.cityobject_id
    FROM cityobject_genericattrib
    WHERE attrname = 'oorspronkelijkbouwjaar' AND intval > 2000)
ORDER BY id ASC;
```

8. Query all buildings within a certain bounding box:

```
SELECT * FROM citydb.cityobject
WHERE objectclass_id = 26
    AND ST_Contains(
        ST_MakeEnvelope(188000, 590000, 188600, 600000, 7415),
            envelope)
ORDER BY id ASC;
```

9. Query building on point

```
SELECT * FROM citydb.cityobject
WHERE envelope && ST_MakePoint(188566.63, 598936.35)
ORDER BY id ASC;
```

10. Query number of children

```
SELECT a.gmlid, COUNT(a.gmlid)
FROM citydb.cityobject a, citydb.cityobject b, citydb.building
WHERE a.id = building.building_parent_id AND b.id = building.id
GROUP by a.gmlid
```

11. Query building with certain children

```
SELECT citydb.cityobject.*
FROM citydb.cityobject a, citydb.cityobject b,
    citydb.building, citydb.cityobject
WHERE a.id = building.building_parent_id AND b.id = building.id
AND a.gmlid = cityobject.gmlid
AND b.gmlid = 'NL.IMBAG.Pand.0629100000020136-0'
```

12. Query buildings with certain parents

```
SELECT citydb.cityobject.*
FROM citydb.cityobject a, citydb.cityobject b,
    citydb.building, citydb.cityobject
WHERE a.id = building.building_parent_id AND b.id = building.id
AND b.gmlid = cityobject.gmlid
AND a.gmlid = 'NL.IMBAG.Pand.0629100000020136'
```

13. Query geometry by LoD

```
SELECT citydb.cityobject.*
FROM citydb.cityobject, citydb.building, citydb.surface_geometry
WHERE building.lod1_solid_id IS NOT NULL
    AND cityobject.id = building.id
    AND building.lod1_solid_id = surface_geometry.id
ORDER BY id ASC
```

14. Query certain LoD of certain building

```
SELECT citydb.surface_geometry.*
FROM citydb.cityobject, citydb.building, citydb.surface_geometry
WHERE building.lod1_solid_id IS NOT NULL
AND cityobject.id = building.id
AND building.lod1_solid_id = surface_geometry.id
AND cityobject.gmlid = 'NL.IMBAG.Pand.0503100000000018-0'
ORDER BY id ASC
```

5.1.2 CJDB

1. The import time will be timed inside the Importer itself.
2. To determine the size of a database (replace *dname* with the name of the database):

```
SELECT pg_size_pretty( pg_database_size('dbname') );
```

3. Add an attribute footprint_area for all buildings:

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

4. Add 10 metres to the footprint_area attribute for all buildings:

```
UPDATE cjdb.cj_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';
```

5. Remove the footprint_area attribute for all buildings:

```
UPDATE cjdb.cj_object
SET attributes = jsonb_set_lax(attributes::jsonb, '{footprint_area}',
    null, true, 'delete_key')::json
WHERE type = 'Building';
```

6. Query building with ID 'NL.IMBAG.Pand.0629100000020136':

```
SELECT * FROM cjdb.cj_object
WHERE object_id = 'NL.IMBAG.Pand.0629100000020136'
ORDER BY id ASC
```

7. Show all buildings made after the year 2000:

```
SELECT * FROM cjdb2.cj_object
WHERE (attributes->'oorspronkelijkbouwjaar')::int > 2000
ORDER BY id ASC;
```

8. Query all buildings within a certain bounding box:

```
SELECT * FROM cjdb.cj_object
WHERE type = 'Building'
    AND ST_Contains(ST_MakeEnvelope(188000, 590000, 188600,
    600000,7415),ground_geometry)
ORDER BY id ASC;
```

9. Query building on point (return the building which the point locates on)

```
SELECT * FROM cjdb.cj_object
WHERE ground_geometry && ST_MakePoint(188566.63, 598936.35)
ORDER BY object_id ASC;
```

10. Query number of children

```
SELECT COUNT(*), parent_id
FROM cjdb.family
GROUP BY parent_id
```

11. Query building with certain children

```
SELECT o.* FROM cjdb.FAMILY f
INNER JOIN cjdb.cj_object o ON o.object_id = f.parent_id
WHERE f.child_id = 'NL.IMBAG.Pand.0629100000020136-0'
```

12. Query buildings with certain parents

```
SELECT o.*  FROM cjdb.FAMILY f
INNER JOIN cjdb.cj_object o ON o.object_id = f.child_id
WHERE f.parent_id = 'NL.IMBAG.Pand.0629100000020136'
```

13. Query geometry by LoD

```
SELECT * FROM cjdb.cj_object
WHERE geometry::jsonb @> '[{"lod": "1.2"}]'::jsonb
ORDER BY id ASC
```

14. Query certain LoD of certain building

```
SELECT * FROM cjdb.cj_object
WHERE geometry::jsonb @> '[{"lod": "1.2"}]'::jsonb
    AND object_id = 'NL.IMBAG.Pand.0629100000020136'
ORDER BY id ASC
```

5.2 CJDB internal benchmarking

5.2.1 Different version of CJDB

In order to produce a better data model, we've designed 4 versions of the CJDB data model. We validated that the current data model was optimal by designing different structural versions to aid in the validation. The main difference between them is the way they organise the attributes. In the first model (v1), we have geometry (jsonb) and attributes (jsonb) in two columns; in the second model (v2), we have geometry and its value as one of the key-value pairs in attributes; in the third model v3, we have children, parents, and geometry all as one of the key-value pairs in attributes. In the last model (v4), we designed mandatory indexes for columns commonly used by users, including id, object_id, ground geometry and geometry, the detail information is in Table 1. In addition, we have included a separate table for the parents-children relationship between Building and BuildingPart, implementing the many-to-many mapping relationship, in order to test whether we can speed up the related queries.

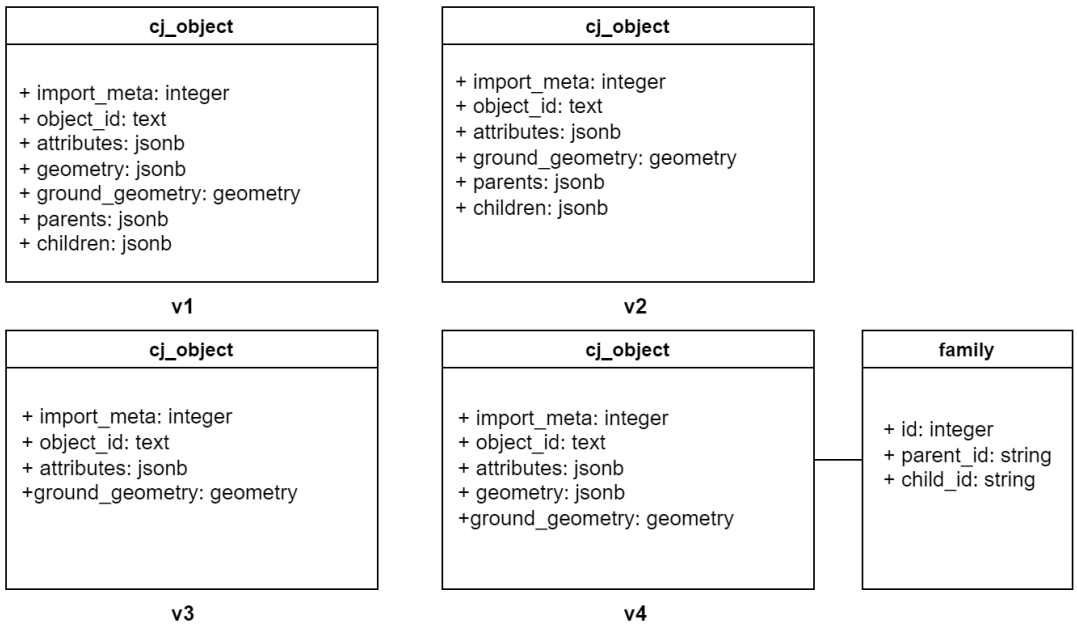
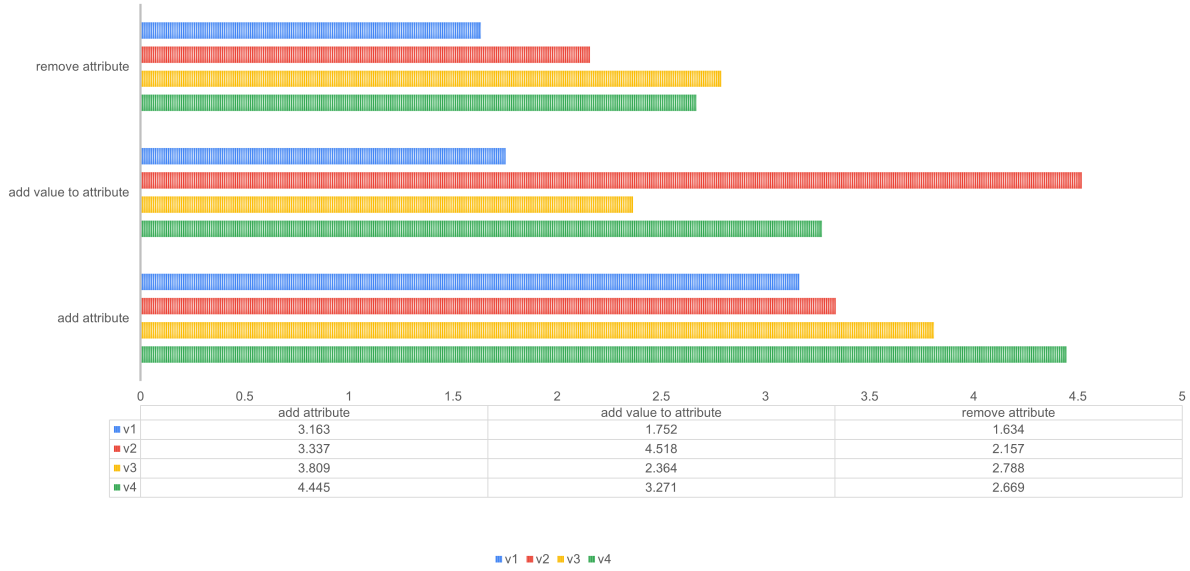


Figure 3: UML diagram of the cj_object table of different versions of CJDB

We divided the queries in 6.2 into two categories, update query and select query, and ran them separately in different versions of the data model, choosing the optimal model by comparing the run times. The queries were all run 10 times to ensure that the database environment was all hot cached.

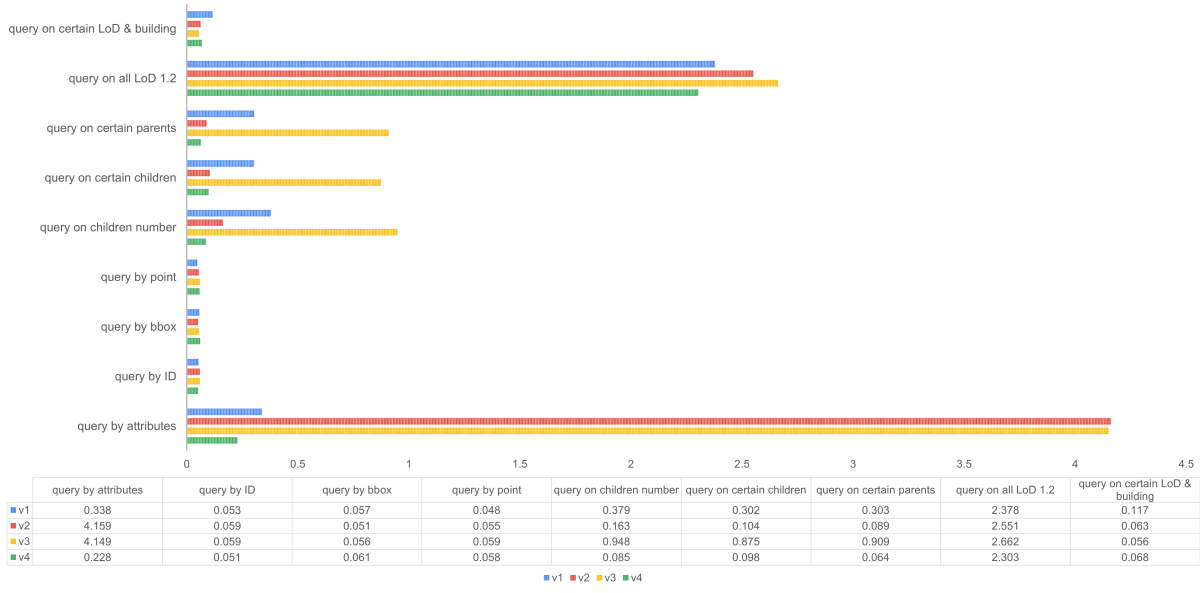
Table 1: indexing in V4

column	indexing method
id	BTREE
object_id	BTREE
ground_geometry	GIST
geometry - LoD	GIN



	add attribute	add value to attribute	remove attribute
v1	3.163	1.752	1.634
v2	3.337	4.518	2.157
v3	3.809	2.364	2.788
v4	4.445	3.271	2.669

Figure 4: performance of different version on updating data



	query by attributes	query by ID	query by bbox	query by point	query on children number	query on certain children	query on certain parents	query on all LoD 1.2	query on certain LoD & building
v1	0.338	0.053	0.057	0.048	0.379	0.302	0.303	2.378	0.117
v2	4.159	0.059	0.051	0.055	0.163	0.104	0.089	2.551	0.063
v3	4.149	0.059	0.056	0.059	0.948	0.875	0.909	2.662	0.056
v4	0.228	0.051	0.061	0.058	0.085	0.098	0.064	2.303	0.068

Figure 5: performance of different version on queries

As can be seen from Figures 4 and 5, v1 and v4 are the two models that perform best in queries.

Because v2 and v3 run significantly longer in queries, we directly leave these two models out of the consideration. Of the two remaining models, v1 performs better in update queries, while v4 is better in queries. We analyse that this is because we have added indexes to v4 and therefore need to update the indexes while updating the data. However, we consider that CJDB users will generally be more likely to use attribute queries more frequently than update queries, and we need to ensure that the queries are running as efficiently as possible.The new family table in v4, for example, has significantly improved the speed of queries on parents/children. In attribute queries, v4's indexes for attributes also improved efficiency. Given the excellent performance of v4 across a range of queries, we chose v4 as the final data model for CJDB.

5.2.2 Choose between bounding box and ground geometry

We have devised two ways to record the bounding box of a building, one by describing the bounding box by min_X, min_Y, max_X, max_Y, and one by a more refined building footprint. As both the bbox and ground_geometry columns have similar functions, they can be used for a variety of spatial queries, such as querying buildings within a certain range and querying whether a point falls within a certain building. We have therefore set up a series of queries to compare the performance of the two.

First, we use query 9 to test bbox and ground_geometry separately, and EXPLAIN ANALYSIS to test the efficiency of the runs, as shown in table 2. The results are shown in table 2. There is no significant difference in efficiency between the two, and both running time and planning times are very similar.

We then consider whether bbox and ground_geometry affect the results and precision of other queries. The first was Query on point (query 10). After testing, it was shown that the choice of bbox and geometry does affect the query results, a clear demonstration of this is shown in Figure 6.

Figure 6 shows the bbox of the building on the left and the ground_geometry on the right. if we use the bbox when the point we want to query falls on the smaller, right-hand side of the building, then the larger building will inevitably be selected. If we use ground_geometry, on the other hand, this does not happen. Only our target building will be selected.We therefore believe that ground_geometry performs better in CJDB.

Table 2: comparison between bbox and ground geometry

column	planning time	execution time	run time	indexing
bbox	0.184 msec	0.156 msec	54 msec	bbox
ground_geometry	0.183 msec	0.134 msec	51 msec	ground_geometry

5.2.3 Indexing test on LoD and partial index on attributes

As geometry is a very large column, we tried to speed up the index by creating index. We used the following query to create the index and tested the performance with and without the index. It took 14:14.101 to create the index.

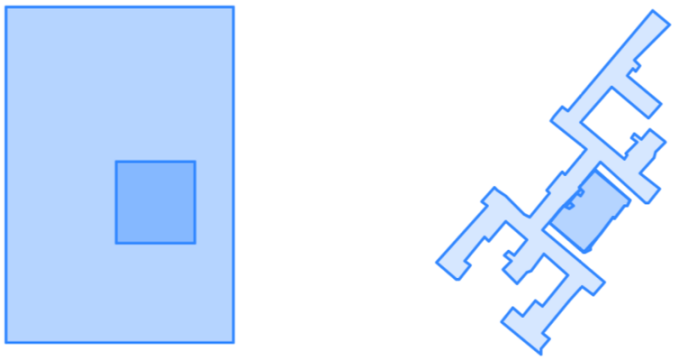


Figure 6: bounding box & ground geometry

```
CREATE INDEX lod ON CJDB.cj_object USING GIN (geometry);
```

As can be seen from the comparison in Table 5, the difference in operational efficiency between with and without indexes is relatively clear: the indexing can improve the operational speed. Considering the sheer volume of data (over 10,000 rows), it is necessary to create an index for geometry. We have added this index creation to the importer.

Table 3: LoD Indexing

column	query on LoD	query on LoD&buildings	indexing size(MB)
indexing	10:37.872	00:0.104	2488
no indexing	12:22.967	00:0.110	0

We also tried creating a partial index on the attributes column, and we tested the run time of query 8 and updating attributes values with indexes, with a partial index, and without indexes, using the example of "oorspronkelijkbouwjaar" in the attributes. The result is in table 6. Among three situations, the partial index runs perform better than the index and is smaller. However, we don't make it mandatory to include a partial index in the importer, we prefer to let the user decide. This is because although there is a real increase in operational efficiency, the improvement is not very significant. User can add the partial index to the data import as needed.

Table 4: Partial index on attributes (unit: s)

index type	planning time	execution time	run time	indexing size(MB)
index	0.586	10.170	26.422	43
partial index	0.355	9.974	26.983	28
no index	0.624	13.061	28.170	0

5.2.4 Analysis of CJDB storage

We imported a total of 500 tiles of 3D BAG on the server, and the size of the CJDB was 5722MB when we imported it without running any queries. After running the above query, the size of CJDB has changed to 8150MB. This means that the query generates a cache, which makes the

database larger, but the exact value of that larger depends on user usage. In order to analyse the size of each table in CJDB separately, the following code was used to check this.

```
  WITH RECURSIVE pg_inherit(inhrelid, inhparent) AS
  (select inhrelid, inhparent
  FROM pg_inherits
  UNION
  SELECT child.inhrelid, parent.inhparent
  FROM pg_inherit child, pg_inherits parent
  WHERE child.inhparent = parent.inhrelid),
  pg_inherit_short AS (SELECT * FROM pg_inherit
  WHERE inhparent NOT IN (SELECT inhrelid FROM pg_inherit))
  SELECT table_schema
  , TABLE_NAME
  , row_estimate
  , pg_size_pretty(total_bytes) AS total
  , pg_size_pretty(index_bytes) AS INDEX
  , pg_size_pretty(toast_bytes) AS toast
  , pg_size_pretty(table_bytes) AS TABLE
  , total_bytes::float8 / sum(total_bytes) OVER ()
        AS total_size_share
FROM (
  SELECT *, total_bytes-index_bytes-COALESCE(toast_bytes,0)
  AS table_bytes
  FROM (
       SELECT c.oid
            , nspname AS table_schema
            , relname AS TABLE_NAME
            , SUM(c.reltuples) OVER (partition BY parent)
                AS row_estimate
            , SUM(pg_total_relation_size(c.oid)) OVER (partition BY parent)
                AS total_bytes
            , SUM(pg_indexes_size(c.oid)) OVER (partition BY parent)
                AS index_bytes
            , SUM(pg_total_relation_size(reltoastrelid))
                OVER (partition BY parent) AS toast_bytes
            , parent
       FROM (
             SELECT pg_class.oid
                 , reltuples
                 , relname
                 , relnamespace
                 , pg_class.reltoastrelid
                 , COALESCE(inhparent, pg_class.oid) parent
             FROM pg_class
                 LEFT JOIN pg_inherit_short ON inhrelid = oid
             WHERE relkind IN ('r', 'p')
          ) c
          LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
) a
```

```
  WHERE oid = parent
) a
ORDER BY total_bytes DESC;
```

Table 5: CJDB storage before running queries (unit: MB)

table name	index	TOAST	table	total
cj_object	258	3629	1682	5569
family	86	0	67	153
import_meta	0.064	0	1	1.072

Table 6: CJDB storage after running queries (unit: MB)

table name	index	TOAST	table	total
cj_object	316	3630	4046	7991
family	86	0	67	153
import_meta	0.064	0	1.144	1.216

TOAST is a mechanism PostgreSQL uses to keep physical data rows from exceeding the size of a data block (typically 8KB). Postgres does not support physical rows that cross block boundaries, so the block size is a hard upper limit on row size. To allow user tables to have rows wider than this, the TOAST mechanism breaks up wide field values into smaller pieces, which are stored "out of line" in a TOAST table associated with the user table.[6]

5.2.5 Analysis of 3DCityDB storage

In section 5.2.4 we analyzed the storage of 500 tiles of 3D BAG imported to CJDB on the server. We did a similar analysis on the storage of 500 tiles of 3D BAG imported to 3DCityDB on the server using the same query from section 5.2.4. See the tables below for the size of each table in 3DCityDB.

Table 7: 3DCityDB storage before running queries (unit: MB)

table name	index	TOAST	table	total
building	663	0.008192	103	766
cityobject	838	0.008192	1036	1874
cityobject_genericattrib	1801	0.008192	2106	3907
surface_geometry	16000	233	23000	39000
thematic_surface	261	NULL	140	401

Table 8: 3DCityDB storage after running queries (unit: MB)

table name	index	TOAST	table	total
building	663	0.008192	103	766
cityobject	838	0.008192	1036	1874
cityobject_genericattrib	2002	0.008192	2279	4281
surface_geometry	16000	233	23000	39000
thematic_surface	261	NULL	140	401

5.3 Benchmarking: CJDB & 3DCityDB

5.3.1 Indexes of 3DCityDB

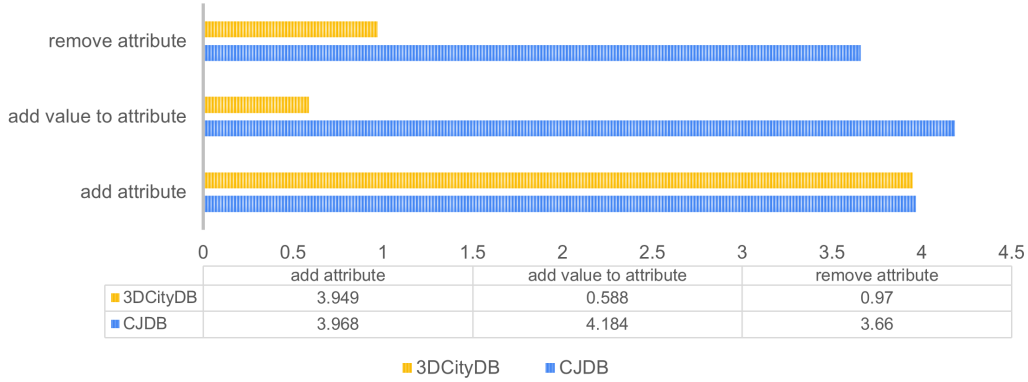
Since in the previous sections the indexing of CJDB has been mentioned, we find it also important to report what indexes are used in 3DCityDB before we make the comparison with CJDB. In 3DCityDB the user can manually activate or deactivate indexes on predefined tables of the 3DCityDB schema and check their status via the 3DCityDB Importer/Exporter. See figure 7 for the index type of the affected columns and from which table they are. In the benchmarking tests for the comparison with CJDB these indexes are activated.

Index type	Column(s)	Table
Spatial	ENVELOPE	CITYOBJECT
Spatial	GEOMETRY	SURFACE_GEOMETRY
Spatial	SOLID_GEOMETRY	SURFACE_GEOMETRY
Normal	GMLID, GMLID_CODESPACE	CITYOBJECT
Normal	LINEAGE	CITYOBJECT
Normal	CREATION_DATE	CITYOBJECT
Normal	TERMINATION_DATE	CITYOBJECT
Normal	LAST_MODIFICATION_DATE	CITYOBJECT
Normal	GMLID, GMLID_CODESPACE	SURFACE_GEOMETRY
Normal	GMLID, GMLID_CODESPACE	APPEARANCE
Normal	THEME	APPEARANCE
Normal	GMLID, GMLID_CODESPACE	SURFACE_DATA
Normal	GMLID, GMLID_CODESPACE	ADDRESS

Figure 7: Spatial and normal indexes affected by the index operation[4]

5.3.2 Comparison using Delft dataset (21 tiles)

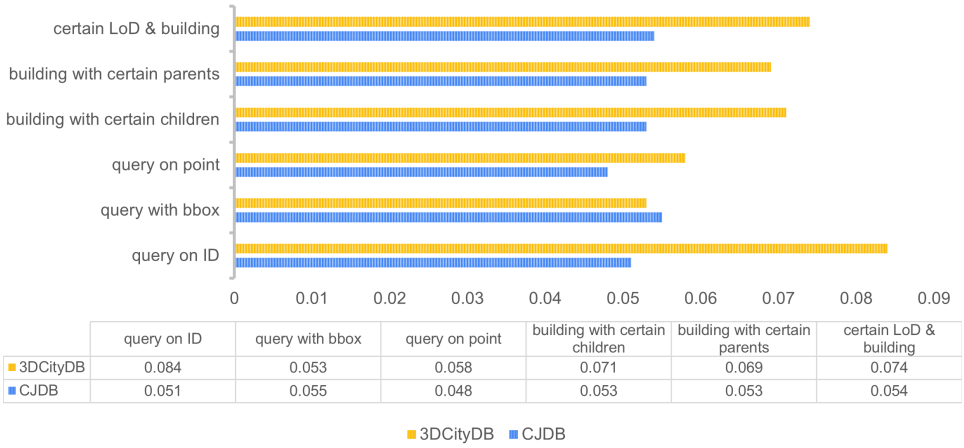
The first relevant comparison we made between CJDB and 3DCityDB is when we used the 'Delft' dataset, 21 tiles which cover Delft. We did this after comparing the different versions of CJDB, and decided that v4 was the best. The results of the comparison can be found in the Figures 8, 9 and 10. The results were gotten by running the query we wanted to test around 10 times, to ensure the cache was warm, and until the query time did not fluctuate as much anymore.



	add attribute	add value to attribute	remove attribute
3DCityDB	3.949	0.588	0.97
CJDB	3.968	4.184	3.66

Figure 8: Comparison of CJDB and 3DCityDB with Delft dataset part I (unit: s)

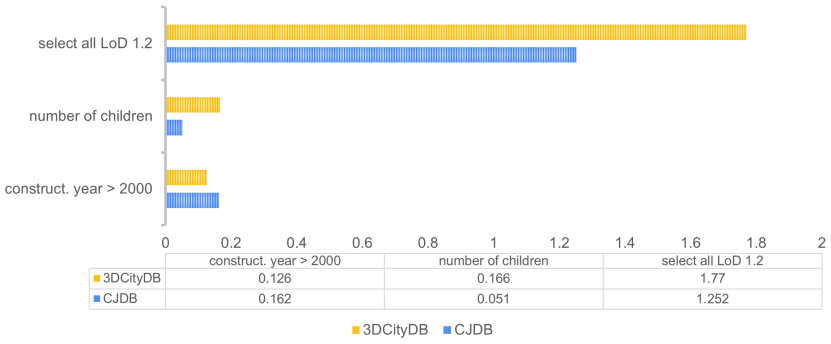
In Figure 8 it can be seen that adding an attribute is only 0.019s slower in CJDB compared to 3DCityDB, but updating and removing the attribute values are much slower in CJDB. We think that this has to do with the partial index created on the attribute column of CJDB. Which makes changing these values take longer than in 3DCityDB, because the index needs to be updated as well. In 3DCityDB the attributes are in a separate table and the values of each attribute for each building have their own single row, and there are no indexes in the 3DCityDB attribute table. Also, all the attributes of one object in CJDB are compacted as one JSONB element, so it takes more time to read the attributes compared to 3DCityDB's structure. This also affects CJDB querying a building on attribute, see Figure 10, where we used querying by construction year as an example.



	query on ID	query with bbox	query on point	building with certain children	building with certain parents	certain LoD & building
3DCityDB	0.084	0.053	0.058	0.071	0.069	0.074
CJDB	0.051	0.055	0.048	0.053	0.053	0.054

Figure 9: Comparison of CJDB and 3DCityDB with Delft dataset part II (unit: s)

Furthermore, in Figure 9 we can see that the query on ID is faster in CJDB. We think that the reason for this is because 3DCityDB has more rows in its cityobject table because it includes each building's thematic surfaces. Naturally, the query on ID is then slower in 3DCityDB because it has to go through more rows. However, thematic surfaces have a NULL value for their envelope in 3DCityDB, which is the geometry column and also the column which is queried with bbox and point. We think that, even though 3DCityDB has to go through these thematic surface rows, it takes minimal time because the thematic surface rows have NULL values for their envelope (geometry) column. This is why the query on point is still faster in CJDB, but we could not explain why the query with bbox are so similar, we would have to test this with an even bigger dataset.



	construct. year > 2000	number of children	select all LoD 1.2
3DCityDB	0.126	0.166	1.77
CJDB	0.162	0.051	1.252

Figure 10: Comparison of CJDB and 3DCityDB with Delft dataset part III (unit: s)

The building queries by certain parents and certain children in CJDB are faster because of the family table, see Figure 2. While 3DCityDB also have each building's relations in a separate building table, the buildings are referenced by a key and that needs to be linked with the citobject table to find the building ID's. We see similar results for querying the number of children, CJDB is faster because of the family table.

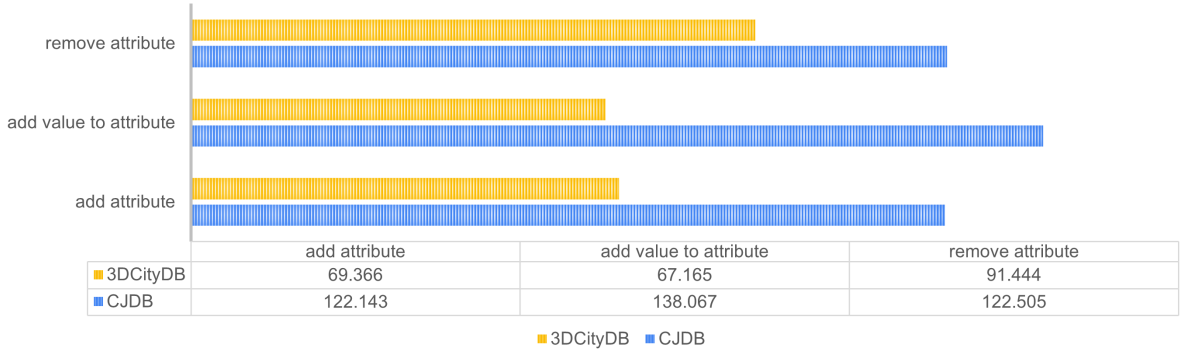
CJDB's LoD queries are also faster than 3DCityDB's ones, we suspect that this has to do with the fact that 3DCityDB's LoD information are in a separate table called building. Not only is this table filled with all the surfaces, the information for the LoD of a geometry are also split into multiple columns one for each LoD. Which makes it take longer to query the LoD compared to CJDB, where the LoD information are found in the main cj_object table and the information for all LoD's of a geometry are in one column in one row packed together as a JSONB element and the query speed is then also increased by adding a GIN index.

These are however still preliminary results. To confirm these findings we will have to increase the input data size, so use an even bigger subset than the Delft dataset. To be able to do this we will import the data to a database on our server. Also, we will have to run the queries more than 10 times and take the average query times of these runs.

5.3.3 Comparison using data of a province (500 tiles)

Initially, to confirm our findings in Section 5.3.2, we wanted to use the whole of 3D BAG as input data, which is about 8000 tiles. The amount of tiles were cut down several times from 8000 tiles to 4000 tiles and 4000 tiles to 2000 tiles, since our server only had space for about 200GB. It took CJDB about 7 hours to import 2000 tiles and 3DCityDB about 35 hours. At this point the server was 100% full, with CJDB taking up about 26GB and 3DCityDB about 170GB, which did not allow us to connect to the database using pgadmin. So we finally decided to use 500 tiles. It took CJDB 1 hour and 55 minutes to import 500 tiles and 3DCityDB 5 hours and 34 minutes. CJDB was 5.61GB in size and 3DCityDB 46GB.

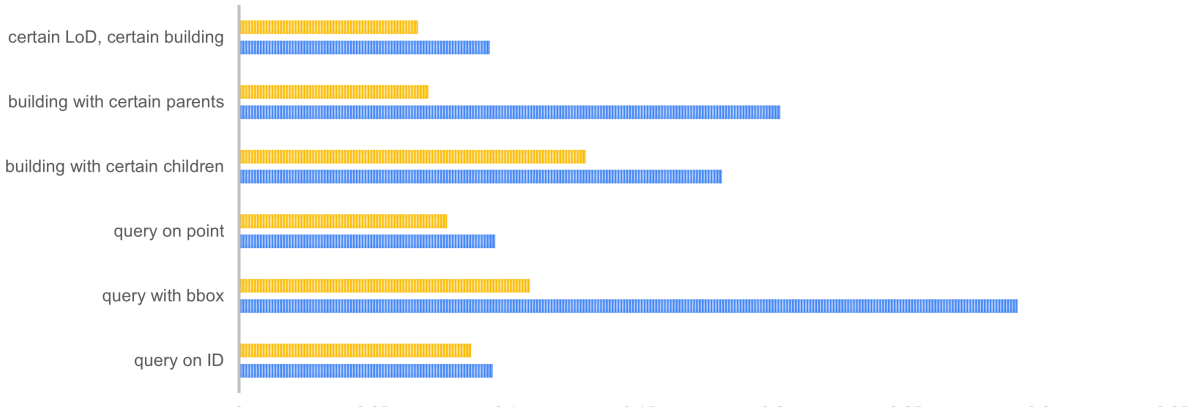
With the 500 tiles imported to both a CJDB database and 3DCityDB database on our server we first ran quick tests, similar to the tests in Section 5.3.2, where we ran each query 10 times. The results can be found in Figure 11, 12 and 13. The update attribute queries in Figure 11 are similar to the results in Figure 8 in Section 5.3.2, were CJDB performs slower than 3DCityDB. However, the other results were not as we expected, since CJDB performed worse than 3DCityDB on the rest of queries as well. We suspected (and hoped) that because we increased the input dataset size, but still ran the query 10 times each, we were getting more fluctuations in query times. And with running the queries only 10 times, we were not getting accurate results. In order to get accurate results we wanted to run the queries 100 times and take the average query time of all these 100 runs for each query. We did this by using pgbench.



	add attribute	add value to attribute	remove attribute
3DCityDB	69.366	67.165	91.444
CJDB	122.143	138.067	122.505

3DCityDB CJDB

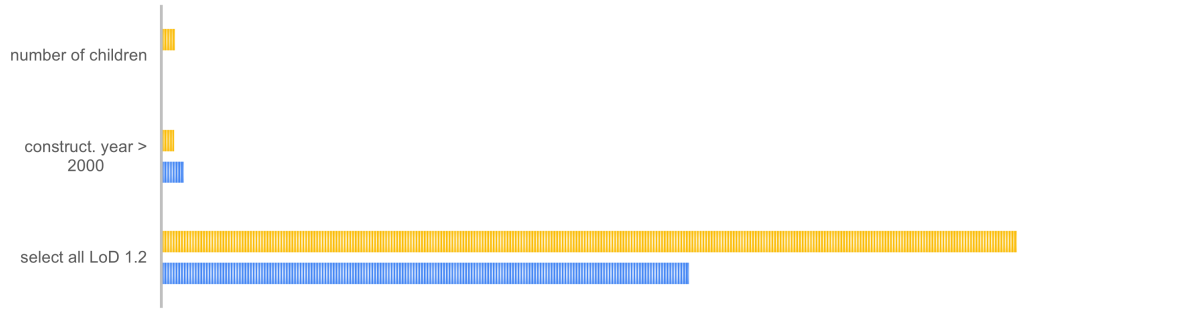
Figure 11: Comparison of CJDB and 3DCityDB part I (unit: s)



	query on ID	query with bbox	query on point	building with certain children	building with certain parents	certain LoD, certain building
3DCityDB	0.087	0.109	0.078	0.13	0.071	0.067
CJDB	0.095	0.292	0.096	0.181	0.203	0.094

3DCityDB CJDB

Figure 12: Comparison of CJDB and 3DCityDB part II (unit: s)



	select all LoD 1.2	construct. year > 2000	number of children
3DityDB	1034.563	15.12	16.356
CJDB	637.872	26.983	0.115

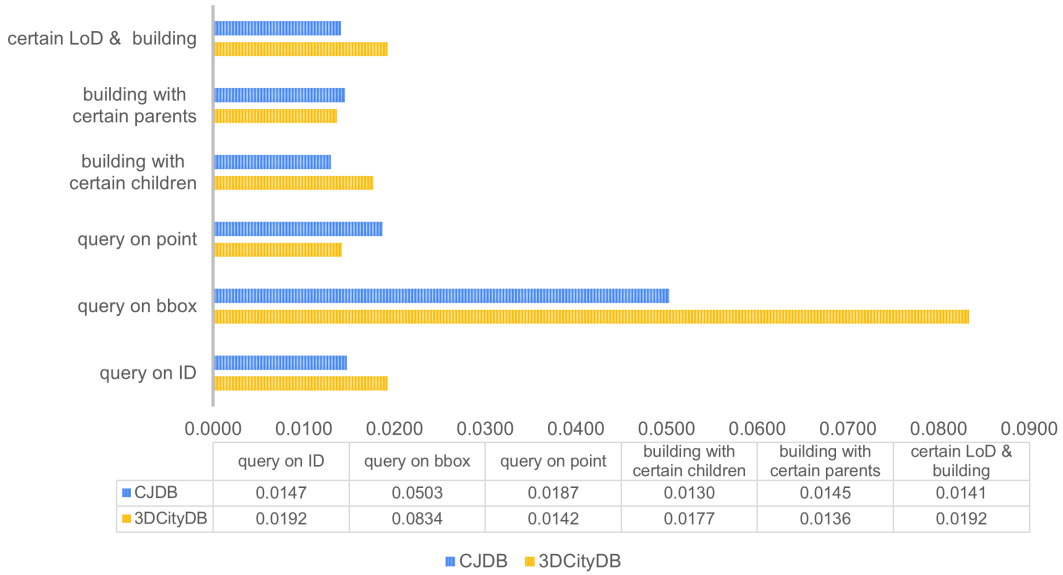
3DityDB CJDB

Figure 13: Comparison of CJDB and 3DCityDB part III (unit: s)

5.3.4 Comparison with 500 tiles (using pgbench)

pgbench is a program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction.[5]

For our benchmark tests we made scripts for each of the CJDB and 3DCityDB queries to be used with pgbench. pgbench allowed us to tune the parameters so it performs each query 100 times, and returns the average query time. These results can be found in Figure 14 and 15. We were also testing adding an attribute, adding a value to an attribute and removing an attribute with pgbench. However, we experienced some complications with the server and could not carry out all of these tests and complete the comparison. We are hoping to resolve the server issues and add these tests to the next version of this report.



	query on ID	query on bbox	query on point	building with certain children	building with certain parents	certain LoD & building
CJDB	0.0147	0.0503	0.0187	0.0130	0.0145	0.0141
3DCityDB	0.0192	0.0834	0.0142	0.0177	0.0136	0.0192

Figure 14: Comparison of CJDB and 3DCityDB with pgbench part I (unit: s)

Continuing with the comparison on Figure 14, querying on ID is again faster in CJDB. Which we think has to do with the simplicity of CJDB's data model compared to 3DCityDB. Surprisingly, the bbox query is now faster and the query on point is slower instead, for CJDB in this test setup. We assume that the bbox query is faster, also because of the simplicity of CJDB's data model. However, we do not have a clear explanation for why the query by point is faster now. This might be cause of the coordinates of the point being used for the query and the result it returns, but also the difference in the geometry. Since the ground geometry is used in the query by point for CJDB and in 3DCityDB the bounding box, which is the envelope column, is used. See Figure 6 for a comparison between bounding box and ground geometry. The number of children and query building by certain children are again faster in CJDB, see Figure 15 and 14 respectively. We assume that this has to do with the simpler family table in CJDB. However, we expected that querying a building by certain parents would have similar results as querying by certain children, but this time it performed slower. We have not been able to find a reason for this and with the difference between the query speed only being less than a millisecond, we assume that these results included many outliers. To confirm this further testing needs to be carried out.

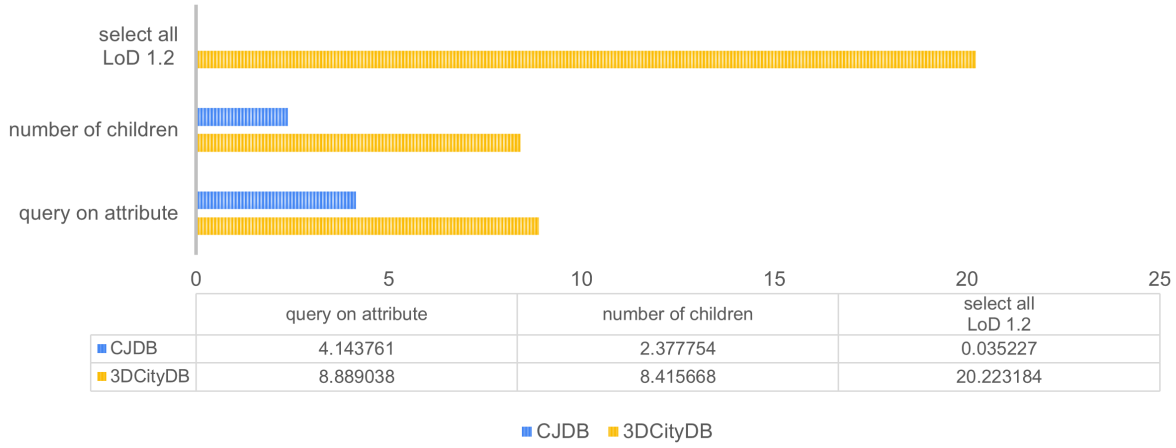


Figure 15: Comparison of CJDB and 3DCityDB with pgbench part II (unit: s)

Also note, in Figure 13, how the query on attribute is two times faster in CJDB due to the usage of the partial index discussed in Section 5.2.3.

For the LoD queries the results in Figure 14 and 15 are similar to the results shown in Section 5.3.2. And like before, we suspect that this has to do with how the LoD information is stored in CJDB and 3DCityDB. 3DCityDB stores its LoD information in a separate surface geometry table and in separate columns for each standard LoD (LoD1, LoD2, LoD3 etc.). While CJDB stores its LoD information in the cj_object table and all type of LoD into one column, the geometry column. This also allows CJDB to store all types of LoD including the refined TUDelft LoDs.[2]

Finally, we made a table and a scaled graph to summarise the comparison of CJDB and 3DCityDB with all the benchmarking queries mentioned in Section 5.1. See Table 9 for the size and import time comparison, and Figure 16 for the scaled comparison of the all queries between CJDB and 3DCityDB. In the figure it can be observed that the results of 3DCityDB is used as the benchmark and the results of CJDB are the scale of 3DCityDB's benchmark. Table 9 shows us that CJDB is smaller in size and also takes less time to import compared to 3DCityDB. Figure 16 shows us that overall CJDB performs better. With the exception of the attribute queries, which are slower because the usage of a JSONB element to store all attributes of an object in one row and one column. On the other hand, the usage of a JSONB element allows CJDB to store all LoD information in one row and one column supporting also the refined TuDelft LoDs, while still performing better than 3DCityDB.

Table 9: Size and import time comparison of CJDB and 3DCityDB

database	size	import time
CJDB	5.61GB	1h 55m 44s
3DCityDB	46GB	5h 34m 50s

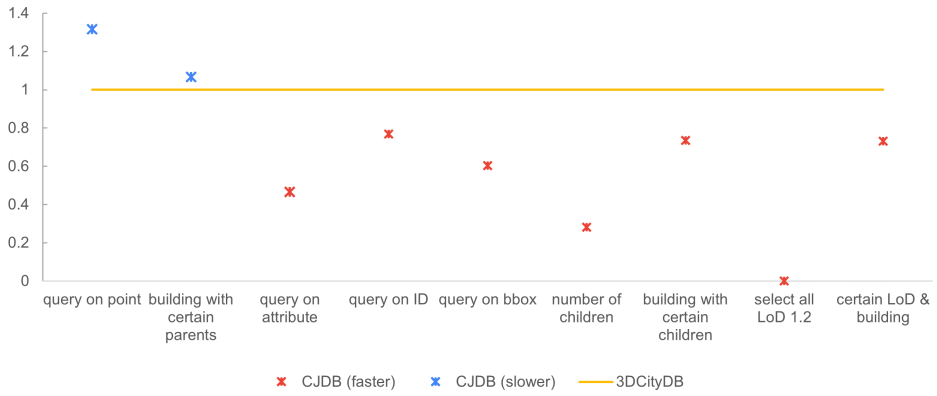


Figure 16: Scaled comparison of CJDB and 3DCityDB with pgbench

5.4 Conclusion

CJDB is designed to improve query efficiency by keeping the data model as simple as possible. The advantages of CJDB are its simple table structure, small storage size and easy to work with: easy data import, simple query design and no need for multiple table joins to get the target result. However, it is difficult to balance the complexity of the data model with the efficiency of the query, so CJDB has not been able to achieve an overall superiority over 3DCityDB in the query part, but from the pgbench query, we have achieved that most of the queries are faster than 3DCityDB. at the same time, there is some potential to improve CJDB subsequently, such as index optimization, which is an important way to improve the query efficiency. Overall, we believe that CJDB is a successful database product when evaluated in terms of storage size, data model structure, query efficiency and other dimensions.

References

[1] F. Biljecki, J. Stoter, H. Ledoux, S. Zlatanova, and A. Çöltekin. Applications of 3D city models: State of the art review. *ISPRS International Journal of Geo-Information*, 4(1):2842–2889, 2015.

[2] F. Biljecki, H. Ledoux, and J. Stoter. Refined LoDs for CityGML Buildings, 2016. URL "https://3d.bk.tudelft.nl/lod/". [Online; accessed 7 November 2022.

[3] H. Ledoux, K. Arroyo Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis. CityJSON: A compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards*, 4(1):1–12, 2019.

[4] C. Nagel, B. Willenborg, F. Kunde, Z. Yao, S. Nguyen, and T. Kolbe. 3D City Database User Manual. 2022.

[5] PostgreSQL. Documentation: 15: pgbench - PostgreSQL, 2022. URL "https://www.postgresql.org/docs/current/pgbench.html". [Online; accessed 7 November 2022].

[6] TOAST. "TOAST — PostgreSQL Wiki", 2016. URL "https://wiki.postgresql.org/wiki/TOAST#:~:text=TOAST%20is%20a%20mechanism%20PostgreSQL,upper%20limit%20on%20row%20size". [Online; accessed 6 November 2022].

[7] Z. Yao, C. Nagel, F. Kunde, G. Hudra, P. Willkomm, A. Donaubauer, T. Adolphi, and T. Kolbe. 3DCityDB-a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, 3(1):1–26, 2018.