3D Kadaster Project Report

Gamma Andhika O
VU Amsterdam
g.a.o.octafarras@student.vu.nl

Tavneet Singh
VU Amsterdam
t.s.tavneet@student.vu.nl

Tom Siebring
VU Amsterdam
t.j.siebring@student.vu.nl

1 Introduction

With the growth of cities and rapid urbanisation, the city landscape is constantly changing. Having a 3D model of the buildings and topological cover can assist city planners and governments in planning new urban zones, detecting illegal buildings and flooding analysis for low lying areas. The Netherlands, being a popular migration destination with limited land resources, can greatly benefit from such a 3D model.

The aim of the project is to construct 3D polygons from two data sources. The first is 2D Kadaster data *Basisregistratie Adressen en Gebouwen* (BAG) - the national registration containing all addresses and buildings in the Netherlands [1]. The second is point cloud LiDAR data for the entire Netherlands, taken from the Actueel Hoogtebestand Nederland (AHN3) website [2].
The research question for the project and the corresponding sub questions are :

Can a data product be built and visualised using HTML5 by merging point cloud data with Kadaster polygons?

- *What is an efficient method to compress, filter and process polygon data on Spark?*
- *How can the merged point cloud data be visualised?*
- *Can the processed area encompass the entire Netherlands, given the data product and budget?*

This is a challenging big data task due to size of AHN2 point cloud LiDAR data being 1.6 TBs, AHN3 being 2.6 TBs, and the unpacked Kadaster data being 50 GB. Both data sets need to be shortened, cleaned, mapped and rendered to 3D polygons on Databricks[3] using Apache Spark[14] to create a data product that can be viewed as a static HTML website with a short(< 1 minute) loading time.

The remainder of the report is as follows. First, the related work and literature is explored and compared to our work in Section 2. Subsequently, the initial data exploration of the data sets is presented in Section 3, followed by the Data Pipeline in Section 4. Section 5 details the technical challenges encountered followed by the Conclusion.

2 Related Work

LiDAR technology has been frequently used for building and landscape reconstruction. [8] [12] [9]. Most of these works introduce new optimisation techniques to increase the building contours modelling accuracy. In contrast, our work is more dependent on dealing with the big data aspect of the problem given the constraints and modelling simple but accurate polygons for buildings. The aim is not to have a rich level of detail to detect windows or create models with a high resolution but to create building polygons with flat roofs. The techniques introduced in the earlier works give an insight into LiDAR data and filtering that could be applied for our use case.

3dfier [1] - is an open source tool developed and maintained by researchers at TU Delft to convert 2D data to 3D models using LAZ files and XML BAG data as input. Rob [4] has used the tool for 3D reconstruction of trees and Dukai et. al. have used the tool on a combination of AHN2 and AHN3 data [5] to generate elevated polygons for the entire Netherlands. The tool has been used for large data on a single multi-threaded server, showing that it can be adapted to run for our use case. Also, some of the techniques related to segmentation and data cleaning presented in the papers are highly relevant for data cleaning and compression.

Shome et. al have worked on a previous iteration of this project [11] but they worked with Actueel Hoogtebestand Nederland2 data and on the SurfSARA cluster. Our work can be considered as an extension of theirs as their final building polygons were not generated for the entire country and the visualisation was compute intensive leading to high lag while viewing on the browser.

3 Data Exploration

3.1 AHN3 Point Cloud

The AHN data set is a collection of precise elevation data (8 points per square meter) for the whole Netherlands taken by planes and helicopters with LiDAR technology. AHN3 Point Cloud is the latest generation of Point cloud data which consists of 1374 LAZ files adding up to 2.5TB fully completed in 2019. The file name of each file represents a single tile, an example file name would be *C_25AN1*, broken down to:

- C_ = prefix of all files
- 25A = the tile identifier
- N1 = position of the tile, where N1(Noord 1) would be top-left. bottom-right would be Z2(Zuid 2).

The position of *C_25AN1* and its surrounding tiles can be seen using the PDOK viewer[4], an example can be seen in Figure 1.

Moreover the AHN3 contains additional attributes per point which were empty in the AHN2 data like *scan_angle_rank*, *intensity* and *classification.*
Classification is an important attribute which consists of the integers:

1 = unclassified (or other)
2 = ground
6 = buildings
9 = water

We can thus remove vegetation, water and unclassified data from the data set, as our goal is to create building polygons and reduce the data size. A considerable disadvantage that accompanies this

[1]https://www.kadaster.nl/
[2]https://www.ahn.nl/historie
[3]https://databricks.com/
[4]https://www.pdok.nl/viewer/

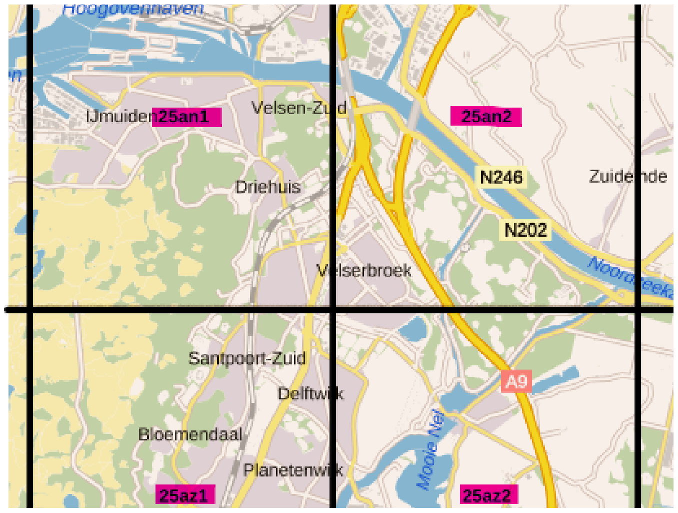
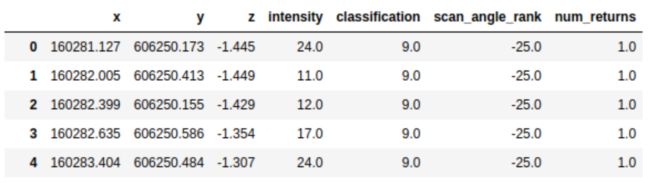


Figure 1: Example of tile distribution of AHN3

method, is that height differs across regions. Since houses in Limburg are not 300m high, a correction must be applied. Also, it leads to a reduction accuracy of height by 3dfier as it uses building and ground points to improve height calculation.



	x	y	z	intensity	classification	scan_angle_rank	num_returns
0	160281.127	606250.173	-1.445	24.0	9.0	-25.0	1.0
1	160282.005	606250.413	-1.449	11.0	9.0	-25.0	1.0
2	160282.399	606250.155	-1.429	12.0	9.0	-25.0	1.0
3	160282.635	606250.586	-1.354	17.0	9.0	-25.0	1.0
4	160283.404	606250.484	-1.307	24.0	9.0	-25.0	1.0

Figure 2: AHN3 Sample data

LAZ format [5] is a compressed version of LAS file format, used for storing LiDAR (LD) data. The structure of LAS file are :

- Header - Contains Point Cloud Format, range of point coordinates, Number of Points.
- Variable Length Record - An optional field containing metadata regarding the points.
- Point Data Records - Data Corresponding to each point in the point cloud.

Since, Spark does not have an inbuilt interface to read LAZ files, python files like Pylas [6] and Laspy[7] were used to read the data. Pylas was found to be 10 times faster on the local laptop environment, hence it was used in the pipeline.

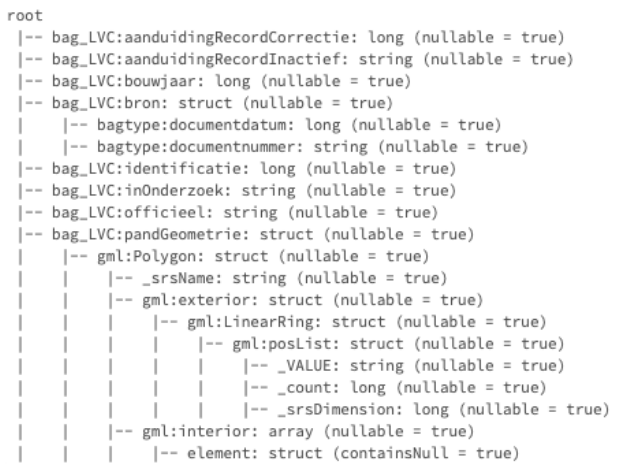
The coordinate values are in RijksDriehoeks (RD) coordinates [8], a geodesic coordinate system used in The Netherlands.

3.2 Kadaster BAG

The Kadaster's BAG is the national registration containing all addresses and buildings in the Netherlands. It is freely available as a 1.8GB[9] ZIP file at *pdok*[10], and has an unpacked size of 50 GB. In total there are approximately 3000 XML files containing all *BAG objects* in the Netherlands. There are seven different classes of which these objects can be instances. For a full definition of the Dutch class names, please refer to [2]. To see how the classes relate - the class diagram is shown in Appendix 1, Figure 8. For the reader, the class names are listed below, accompanied with a simple English description:

- *Woonplaatsen*: place to live
- *Openbare ruimten*: public space
- *Nummeraanduidingen*: address number of *verblijfsobject, standplaats* or *ligplaats*
- *Panden*: building
- *Verblijfsobjecten*: building suited for living/working/leisure
- *Ligplaatsen*: area suited for floating object such as a boat
- *Standplaatsen*: area suited for building

Since the goal of the project is to turn 2D building polygons into 3D models, we are solely interested in the *pand* objects, which reduces the raw data to 12GB. The corresponding XML schema of the *pand* data is shown in Figure 3. After applying the filters described in the next section, the only relevant data that remain are polygon identification and polygon specification, i.e. the coordinates. These are easily retrieved using `'bag_LVC:identificatie'` and `gml:posList._VALUE`. Like the AHN data, the polygon coordinates are also specified in RD coordinates, i.e. *CRS* = 28992, making integration of the two straightforward.



```
root
 |-- bag_LVC:aanduidingRecordCorrectie: long (nullable = true)
 |-- bag_LVC:aanduidingRecordInactief: string (nullable = true)
 |-- bag_LVC:bouwjaar: long (nullable = true)
 |-- bag_LVC:bron: struct (nullable = true)
 |    |-- bagtype:documentdatum: long (nullable = true)
 |    |-- bagtype:documentnummer: string (nullable = true)
 |-- bag_LVC:identificatie: long (nullable = true)
 |-- bag_LVC:inOnderzoek: string (nullable = true)
 |-- bag_LVC:officieel: string (nullable = true)
 |-- bag_LVC:pandGeometrie: struct (nullable = true)
 |    |-- gml:Polygon: struct (nullable = true)
 |    |    |-- _srsName: string (nullable = true)
 |    |    |-- gml:exterior: struct (nullable = true)
 |    |    |    |-- gml:LinearRing: struct (nullable = true)
 |    |    |    |    |-- gml:posList: struct (nullable = true)
 |    |    |    |    |    |-- _VALUE: string (nullable = true)
 |    |    |    |    |    |-- _count: long (nullable = true)
 |    |    |    |    |    |-- _srsDimension: long (nullable = true)
 |    |    |-- gml:interior: array (nullable = true)
 |    |    |    |-- element: struct (containsNull = true)
```

Figure 3: XML schema of *pand* data

4 Data Pipeline

For most of the duration of the project, the goal was to use 3dfier to handle most of the processing. We have tried for many weeks to get it working on the cluster (and the challenges are detailed in Section 5), but in the final week we had to decide to set aside this goal. In the weeks before we worked on an alternative plan, which was then set to work. Because significant time was spent on both

[5]https://laszip.org/
[6]https://pypi.org/project/pylas/
[7]https://github.com/laspy/laspy
[8]https://www.nsgi.nl/rdinfo

[9]as per September 2020
[10]https://www.pdok.nl

3dfier and the eventual data pipeline, considerations for both data pipelines are discussed. However, since 3dfier was not used for our final data product, the focus lies on the latter pipeline.

Our pipeline as seen in Figure 4 consists of two main phases:

- Data preparation - filtering unused attributes from both the BAG and the AHN and writing them as parquet files. All the operations in this phase are done in Spark.
- Transformation - intersecting the two data sets in Spark, and running Point in Polygon in GeoMesa to obtain a single height for each polygon to be visualised.

The infrastructure used by Dukai et. al. [5] was a single server using 20 CPUs and 120 GBs of RAM to perform the necessary calculations described in the paper. It roughly translates to 6x as many calculations with respect to this paper's project goal. Therefore, especially in phase 1, it is crucial to remove as much data as possible that is not required.

4.1 Filtering

4.1.1 ***AHN*** The raw AHN data contains height points for all land area in the Netherlands. We are interested in only a small subset of this - buildings. Fortunately, the AHN3 contains the classification *building*, so the remaining data is removed to reduce the size of the LAZ.

The filtering was done on the entire LAZ dataset and was done with pyspark as we had to utilize the python library pylas for reading the LAZ files. The resulting data was then saved as Parquet and LAZ files (as input to 3dfier). The parquet files were generated by using the Apache Arrow[11] library to convert the Numpy output from Pylas to a columnar Parquet format which could then be saved. The advantage of using Parquet was optimised reading on spark cluster, leading to faster processing in the remaining pipeline.

Some LAZs are 2GB+ in size and unpacks to 15GB while being read by Pylas. These large files failed due to out of memory errors when running them on the 4 core 32GB RAM Workers. The OOM appeared as the Spark config was set to 1 core per executor, in which each executor effectively had 8GB RAM (before memory overhead) if 4 executors were running simultaneously. The first filter successfully filtered 900 files. The files that failed were then logged for a second iteration, in which the each executor were set to work with 2 cores, effectively each executor would have 16GB RAM. The remaining failed files were then ran on the driver node successfully.

Filtering the whole dataset took approximately 15 hours in total. With the first iteration parallelized on 64 executors and the second on 32 executors.

4.1.2 ***Kadaster*** The raw XML data contain many rows that are irrelevant for our purpose, such as demolished buildings. Therefore the following filters were applied:

- start date validity < 202009
- end date validity > 202009 or null
- building in use
- inactive == false
- building exterior only, filtering out the interior if present

[11]https://arrow.apache.org/

Furthermore, only the building identification, year_built, valid_from, number of coordinates and the coordinates themselves were selected and added to the polygon DataFrame. Applying this process on each of the 3000 XML files could easily be done in the driver and greatly reduced the data set.

4.2 Data Transformation

The data product is height map of buildings in several large Dutch cities. To find a building height, a measure can be taken on all height points that have (x,y) coordinates within the building polygon, for example the average. Therefore, all BAG polygons and AHN height points need to be intersected.

4.2.1 ***Option Exploration*** 1. The naive approach is to find, for each polygon in the BAG, all AHN points that lie within the polygon. However, after reducing the AHN data to 250GB, the average tile size is 250GB / 1374 = 180MB. Scanning a whole tile for each of the 10 million buildings implies processing 1.8PB of tile data. Alternatively, the 30km^2 tiles can be split up further to e.g. 1km^2 tiles. This would reduce the processed AHN data to 60TB, 30 times smaller. However, aside from the 60TB still remaining, two other disadvantages of this method are:

- More BAG polygons lie on AHN tile boundaries. In such cases, the AHN data from the adjacent tile has to be loaded, which is expensive.
- The number of AHN files increases from 1,374 to 40,000 in case of 1km^2 tiles.

Decreasing the tile size further makes these disadvantages worse.

2. Instead of joining AHN height data on polygon data, the reverse approach is to join polygon data on height data. This ensures that each height point is processed only once, so 250GB of data. The disadvantage of this approach is that for each point, 10 million / 1374 / 2 = 3.6k polygons need to be checked before a match is found. [12] For each of these checks, a *point-in-polygon* operation needs to be performed, which is computationally expensive. Given the 250GB of point data, this naive approach is also unfeasible.
Common techniques improve on aforementioned techniques include the use of trees, grid systems such as Uber's H3[13], and space filling curves such as Google's S2[7] but due to time constraints these were not looked into. 3dfier uses for example R-trees to quickly look up the geometry data.

3. For geographic operations in Python, GeoPandas is commonly used. However, due to it overloading the driver and not being optimised for distributed processing on Spark we switched to Spark native tools like GeoMesa [13], GeoSpark[14], and RasterFrames [15]. In terms of performance, GeoSpark looked most promising. However, lack of proper documentation and technical difficulties led us to switch to GeoMesa.

[12]The division by 2 assumes that the height point building classification is exactly equal to that of the polygons, and the division by 1374 assumes buildings are equally distributed amongst the tiles are false.
[13]https://www.geomesa.org/
[14]https://github.com/apache/incubator-sedona/
[15]https://rasterframes.io/

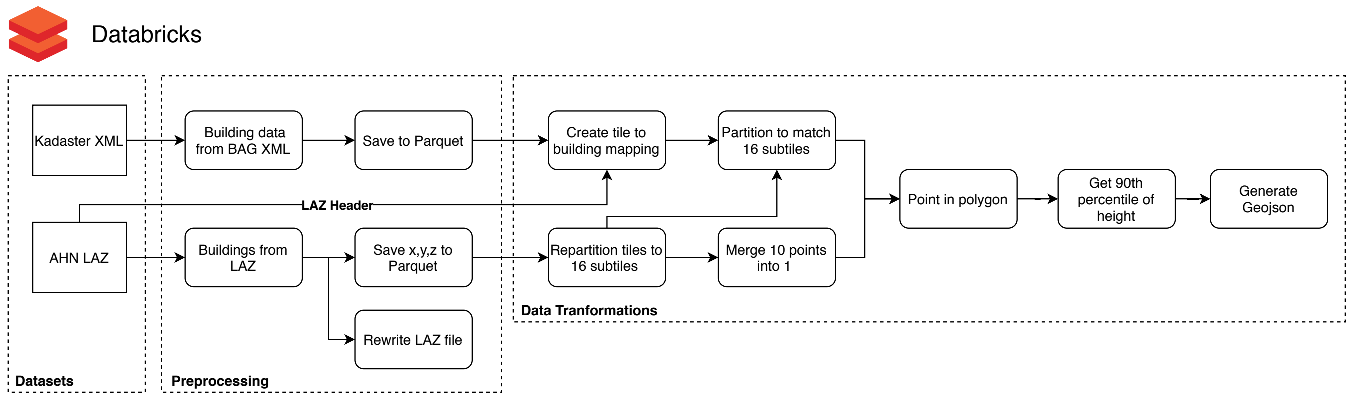


Figure 4: Final pipeline

4.2.2 ***Transformation Pipeline*** To ensure that points from different AHN tiles are correctly mapped to polygons spanning multiple tiles, the minimum and maximum (x,y) coordinates were determined for each polygon. The base tile of a polygon corresponds to the location of min(x,y), i.e. the left bottom corner of the enclosing rectangle in Figure 5. In case max(x,y) differs from the base tile, it is automatically clear whether the polygon spans 2 tiles or all 4. In case of 2, max(x,y) is either located in the tile above or on the right. In case of 4, the tile must be located at the diagonal tile on the top right. Now add a list attribute to each polygon that contains its base, top, right and diagonal tile, respectively. For most tiles, these will be equal. Exploding this column quadruples the number of rows. Removing duplicates subsequently provides the mapping between polygons and tiles.

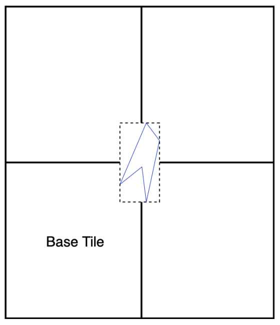


Figure 5: Example of polygon covering four tiles.

Running GeoMesa's *point-in-polygon* operation [16] on 80MB of point data took 2 hours. Inspired from the talk on scaling spatial data analysis [6] to scale Geomesa on Spark, only the relevant polygons should be used for faster processing, we partitioned the polygon and corresponding point cloud data into 16 sub tiles, each having a size of 2km^2. Delta Lake [17] Table partitioned on subtile name was used for polygon data for faster filtering but it was observed that it led to creation of small files, which is why it was not used for point cloud data. However, this did not speed up the process sufficiently, so the decision was taken to merge 10 LiDAR points and by computing the average height of these 10 points. Provided an average area of $100m^2$ per Dutch building [3], this results in 80 height points per building on average.

For each polygon, the 90th percentile of the average heights was stored as max height and the minimum was stored as the height of the building is relative to the ground elevation of that area.

The resulting data of polygons with single height were then ran through a scala UDF to convert the RD coordinates to the WGS84[18] (latitude, longitude) format which is later used in the visualisation. The files are then saved to parquet for the GeoJSON generation.

The data transformation from subpartition to parquet generation took about 20 minutes per tile. The subpartition of tiles and *point-in-polygon* were run as a pipeline with *point-in-polygon* taking more time ; hence point in polygon could be started as soon as one tile subpartition were generated and then both notebooks could be run concurrently as subpartition generation would finish generating the next subtiles before point in polygon required them as input.

For the GeoJson[19] generation, as the format does not compress well and is fairly large, we try to remove as much unneeded data as possible. All coordinates are rounded to 5 decimal points from the original 10 points, height points are rounded to 1 decimal point and unused metadata are removed. Unneeded whitespaces are also removed. These steps bring down the size of the resulting GeoJson by approximately twofold (150MB to 74MB for Amsterdam). GeoJson generation operations were done in pyspark and took <2 minutes to run for a building heavy region such as Amsterdam.

4.2.3 *3dfier considerations* To prepare for 3dfier, all AHN data was filtered on buildings. Furthermore, to speed up the tile joins necessary for polygons located on tile boundaries, for each AHN tile 3 small sub tiles were stored: 1) having all data from the left 500m 2) having all data from the bottom 500m and 3) the intersection of these. Provided the largest building in the Netherlands is smaller than $1,000,000m^2$, these 500m sections should cover nearly all polygons in the accompanying tile.

[16]https://www.geomesa.org/documentation/stable/tutorials/dwithin-join.htmld-within-join
[17]https://docs.databricks.com/delta/index.html
[18]https://epsg.io/4326
[19]https://geojson.org/

4.3 Visualization

The goal of the visualization is to create an interactive tool that enables the user to inspect building heights on different zoom levels and on different locations. The buildings rendered will have different colors based on its height and will only render as 3D when the user is zoomed in (Figure 6) to reduce the number of buildings rendered in a single frame. When zoomed out, a 2D representation of the building is shown (Figure 7). Moreover, as we could not process every city of the Netherlands, we added a list that specifies which cities we have, along with information on how many building polygons are available in each city and how many square kilometers they cover. The user are also able to interact with the list to *fly* to the selected city.

The visualisation was created with HTML+CSS+JavaScript and used MapBox[20] for rendering the 3D buildings on the 2D map.

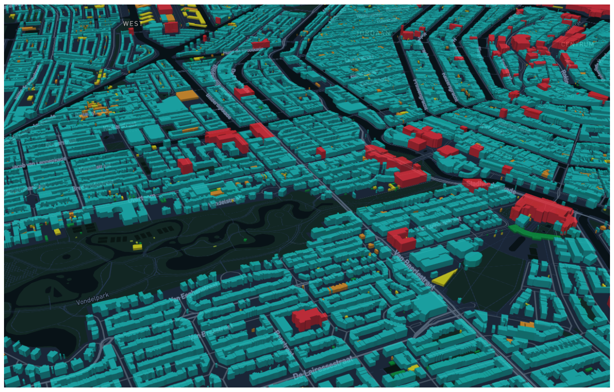


Figure 6: 3D buildings of Amsterdam, zoomed in

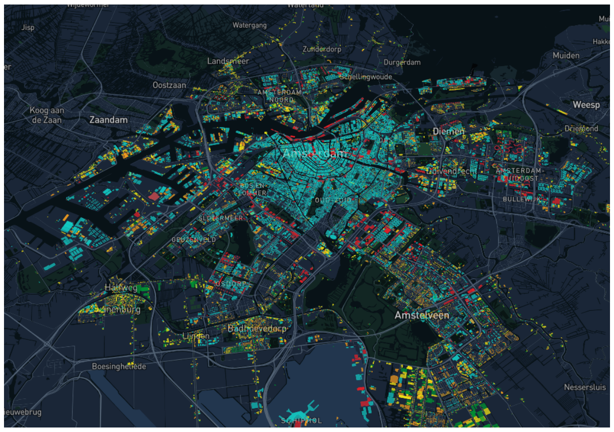


Figure 7: 2D buildings of Amsterdam, zoomed out

The dataset that the visualisation used are in form of GeoJson files. Which is a format based on JSON that is suited for storing geographical data structures but does not have the best compression and can lead to large files. The resulting GeoJson files of the cities might seem small at around 70MB in average, when considering that each file contains hundreds of thousands of buildings, rendering them on a static HTML page becomes a challenge.

Initially, we loaded local GeoJson files directly, this approach had a longer initial loading time and was a bit heavy on browsers when loaded as the rendered buildings always shown. This approach worked well when rendering single cities, but had long initial load times when loading multiple cities (10 minute load time for 8 cities).

The second approach was to convert the GeoJson files to the vector tiles which MapBox uses. This was done by using the Mapbox Tiling Service[21], in which the GeoJson files are pre-processed and converted into mapbox vector tiles as opposed to the previous approach which converted them on the fly on the clients browser. The vector tiles are then hosted on MapBox Studio[22].

Our final approach consisted of a mixture of the previously mentioned approaches. For the rendering 3D buildings, we pre-process them to vector tiles. These buildings will only appear at high zoom levels. For the zoomed out 2 polygons, we host the GeoJson files on Github Pages[23] instead of loading them locally to reduce the client memory overhead. These polygons will be rendered at all zoom levels but take longer to load.

5 Challenges

5.1 AHN LAZ Filtering

A challenge that we faced early on was finding ways to make reading LAZ files scaleable as it was the main bottleneck in the filtering process of the AHN. The only library that supports LAZ reading in spark was spark-iqmulus[24], which was outdated (spark 1.6.2) and has not been maintained for 4 years according to their github activity.

5.2 3dfier

Setting up 3dfier on Databricks and on local to work with our data was a major challenge. Since considerable work was put into making the tool work, a section has been included to help out future teams.

- The Ubuntu build file for 3dfier was outdated with the last commit made in 2018. Building 3dier on Ubuntu 18.04 led to unmet package dependencies and it could not be compiled onto a JAR on databricks.
- 3dfier had a docker setup which ran on the local environment but using it on Databricks was not possible due to Docker in Docker issue [10]. The Docker can only be run with the privileged tag but Databricks system security does not permit it and an AppArmor permission denied is received instead.
- Databricks Container Services was explored to create a custom Docker Image on top of the standard Dockerfile by rewriting the 3dfier Dockerfile to work on a Ubuntu container. The docker build failed for 3dfier as as LibLAS was . The main issues while recreating the Dockerfile were the incompatibility of versions of build tools used in the original Alpine image and the Ubuntu 16.04 image, only some of

[20]https://www.mapbox.com/

[21]https://www.mapbox.com/mts

[22]https://www.mapbox.com/mapbox-studio

[23]https://pages.github.com/

[24]https://github.com/IGNF/spark-iqmulus

which were resolved by building packages from source. For example, Apline 3.1 image using yaml-cpp package version 0.6.2 but Ubuntu 16.04 having support for version 0.5.1.

- 3dfier supports OGR compliant polygon input but the container throws a non compatibility with XMLs due to LASLiB not being comiled with Xerces, a C++ parser [25].

6 Conclusion

The first two sub questions of the research question could easily be answered. Thanks to the clean, well-documented and well-structured data, preprocessing and understanding the data was relatively straightforward. For the visualization, the level of detail highly depended on the density of the point cloud. Given only 8 points per m^2 in the AHN data, most building details had to be left out, resulting in flat roofed buildings whose shape is determined by its Kadaster polygon.

Answering the last sub question of the research question was more difficult. Our final methodology did not allow for processing all of The Netherlands within the constraints - the limiting factor being the point-in-polygon operation. However, a significant speed up for this operation can probably be achieved using a coordinate grid. Therefore, we recommend future related projects to step away from 3dfier and focus on techniques that are particularly fast at handling this bottleneck operation. Another option is to use 3dfier on a Large AWS machine outside Databricks. At the start of the project a setup was created such that scaling down should be fast. Therefore, although time constraints were tight at the end, it was still possible to deliver a small scale data product and visualization. Nonetheless, the main lesson learned during this project is to drop any tool that is unable to produce a quick first iteration using the final setup. At the start of the project it became clear 3dfier was the most commonly used tool in previous works. Therefore, it seemed to make the most sense to use that tool for the processing. Similar to the group of 2018, it turned out very challenging to get little-documented third party software working on the cluster. A significant part of the project had to be invested in getting the software to work, without success. Dropping 3dfier in the third week and focussing on a grid-based point-in-polygon approach would probably have maximized our chances of being able to confirm our main research question.

7 Acknowledgements

We would like to thank Dr Peter Boncz for providing us a separate cluster to work on and answering our questions. We would like to thank Balasz for having a video call with us to explain about 3dfier and the project. We would also like to thank Mr Frank Van Der Heijden from CAD Accent for providing us with us with domain knowledge.

7.1 Task division

The task division is presented in Table 1.

[25] http://xerces.apache.org/xerces-c/

	Report	Code	Visualization
Gamma	Data Pipline (Filtering), Visualisation	Filter AHN LAZs, initial point in polygon, GeoJson generation and Visualization	Visualization
Tavneet	Introduction, Related Work, Data Pipeline, Challenges	Initial AHN Exploration, Automating Data Pipeline, 3dfier Dockerfile and SQLite conversion scripts	-
Tom	Kadaster, Budget, Data Pipeline, Conclusion	(Pre)processing BAG XML, polygon to tile, BAG/AHN tile to subtiles, point in polygon	-

Table 1: Task division.

References

[1] 3dfier 2020. The open-source tool for creating of 3D models. https://github.com/tudelft3d/3dfier.

[2] BAGobjects 2020. Handleiding bij de catalogus van de Basisregistratie Adressen en Gebouwen. https://imbag.github.io/praktijkhandleiding/objecttypen.

[3] CBS. [n.d.]. Small and relatively expensive housing in Amsterdam. https://www.cbs.nl/en-gb/news/2016/14/small-and-relatively-expensive-housing-in-amsterdam.

[4] Rob de Groot. 2020. Automatic construction of 3D tree models in multiple levels of detail from airborne LiDAR data. http://resolver.tudelft.nl/uuid:3e169fc7-5336-4742-ab9b-18c158637cfe.

[5] Balázs Dukai, H. Ledoux, and J. Stoter. 2019. A MULTI-HEIGHT LOD1 MODEL OF ALL BUILDINGS IN THE NETHERLANDS. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* IV-4/W8 (09 2019), 51–57. https://doi.org/10.5194/isprs-annals-IV-4-W8-51-2019

[6] Anthony Fox. [n.d.]. Applying SparkSQL to Big Spatio-Temporal Data Using GeoMesa. https://databricks.com/session/applying-sparksql-to-big-spatio-temporal-data-using-geomesa.

[7] Google. [n.d.]. S2 Geometry. https://s2geometry.io/devguide/s2cell$_hierarchy.html.$

[8] Xuelian Meng and Le Wang. 2009. Morphology-based Building Detection from Airborne Lidar Data. *Photogrammetric Engineering Remote Sensing* 75 (04 2009). https://doi.org/10.14358/PERS.75.4.437

[9] George Miliaresis and Nikolaos Kokkas. 2007. Segmentation and object-based classification for the extraction of the building class from LIDAR DEMs. *Computers Geosciences* 33, 8 (2007), 1076 – 1087. https://doi.org/10.1016/j.cageo.2006.11.012

[10] Jérôme Petazzoni. [n.d.]. Using Docker-in-Docker for your CI or testing environment? Think twice. https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/.

[11] Arumoy Shome, Cees Portegies, and Shruti Rao. [n.d.]. 3D Kadaster of the Netherlands. ([n. d.]).

[12] Shaohui Sun and Carl Salvaggio. 2013. Aerial 3D Building Detection and Modeling From Airborne LiDAR Point Clouds. *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of* 6 (06 2013), 1440–1449. https://doi.org/10.1109/JSTARS.2013.2251457

[13] Uber. [n.d.]. H3: Uber's Hexagonal Hierarchical Spatial Index. https://eng.uber.com/h3/.

[14] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

8 Appendix

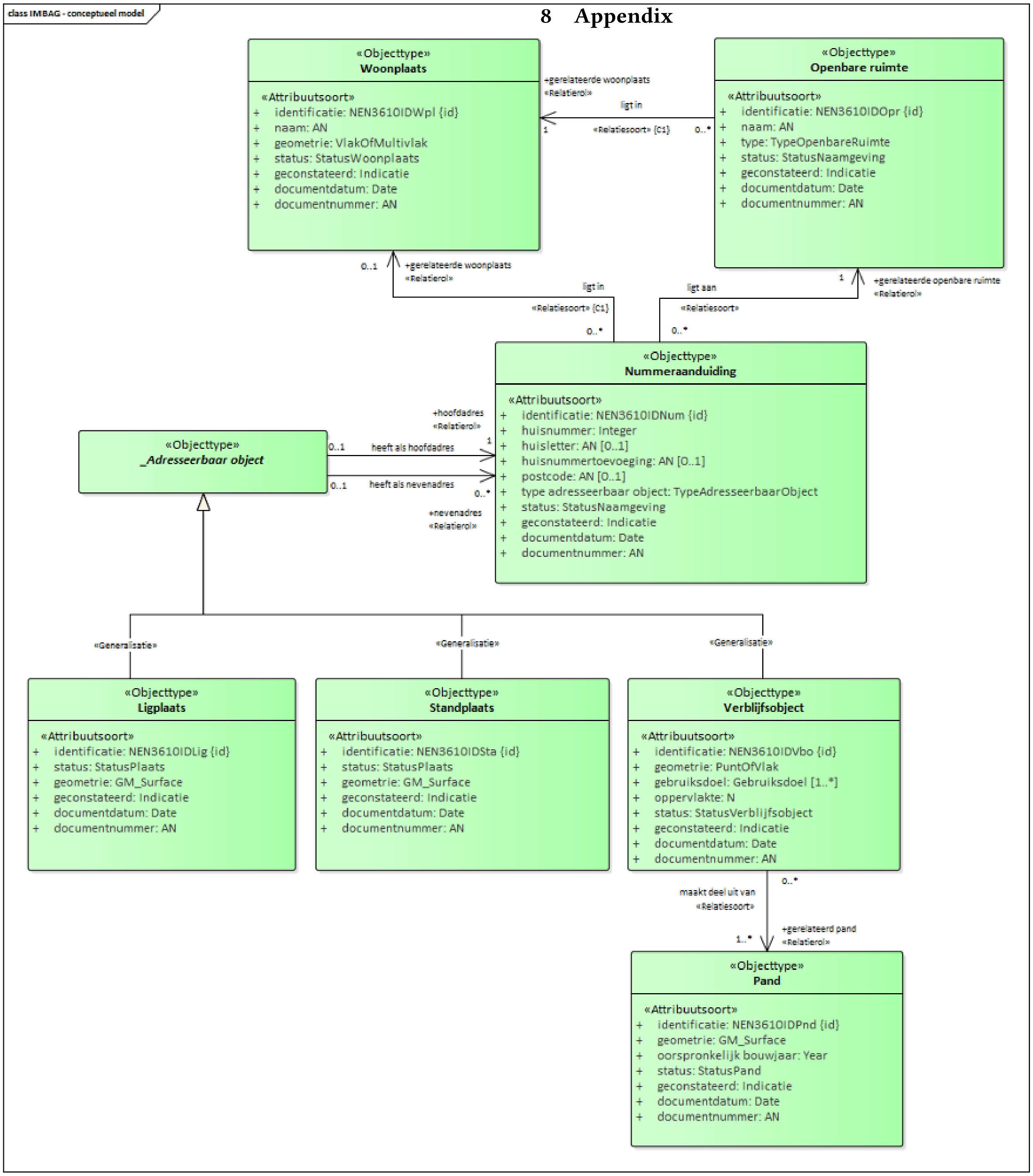


Figure 8: BAG class diagram