Methods for augmenting and accelerating data visualization, using a vector tile map, with metadata from an external database

Kankkonen Markus

Degree Thesis
Information Technology
2022

DEGREE THESIS	
Arcada	
Degree Programme:	Information Technology
Identification number:	25330
Author:	Markus Kankkonen
Title:	Methods for augmenting and accelerating data visualization, using a vector tile map, with metadata from an external database
Supervisor (Arcada):	Fredrik Welander
Commissioned by:	Nokia Advanced Technology Group

Abstract:
Vector tile maps give us new and innovative ways of using maps to visualize data. The ability of the user to dynamically change and interact with the map can lead to interesting new user experiences. In this thesis we showed ways of using vector tiles as a database, a solution we call VectorTileDB, by storing data as metadata of features in the vector tiles. This data is fetched from the tile server whenever the viewport of the map changes. We tested how well VectorTileDB performed against an industry standard NoSQL database MongoDB. We also explored what technologies are used for creating, styling and serving vector tiles. The vector tiles used were in the Mapbox Vector Tiles format, which uses Google's *protobuf* technology for encoding data. This leads to a very fast and efficient fetching of data, that scales very well. There are still many questions concerning the use of VectorTileDB as a production ready database that were not covered in this thesis. But this work shows that this technology is something that has potential and should be further researched. The thesis also explored how to efficiently use the data from the VectorTileDB to augment a map, using either the client-side map library MapLibre or using an external library called Deck.gl to render data. Both solutions proved to work well with VectorTileDB.

Keywords:	Vector tiles, Vector maps, TileServer GL, VectorTileDB, MapLibre, Deck.gl, Mapbox
Number of pages:	47
Language:	English
Date of acceptance:	

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Information Technology
Identifikationsnummer:	25330
Författare:	Markus Kankkonen
Arbetets namn:	Metoder för att utöka och accelerera datavisualisering, med hjälp av en vektor *tile* karta, med metadata från en extern databas.
Handledare (Arcada):	Fredrik Welander
Uppdragsgivare:	Nokia Advanced Technology Group

Sammandrag:
Vektor *tile* baserade kartor ger oss möjligheten att på nya och intressanta sätt använda kartor för att visualisera data. Användaren kan röra sig fritt på kartan och ändra på dess innehåll dynamiskt, vilket leder till nya användarupplevelser som inte tidigare varit möjliga. I detta examensarbete visade vi hur vektor *tiles* kan användas som en databas, en lösning som vi kallar VectorTileDB, genom att lagra data som metadata till objekt i en vektor *tile*. Data, för de objekt som är synliga på kartan, hämtas från servern när användaren rör sig runt på kartan. Vi testade hur väl VectorTileDB klarar sig mot en industristandard NoSQL databas MongoDB. I detta arbete visade vi också hur man kan skapa, stilisera och servera vektor *tiles*. Vi använde oss av Mapbox Vector Tiles formatet, som använder Googles *protobuf* teknologi för att omkodad data till ett mera optimerat format. Vilket leder till att vektor *tilen* är väldigt små och det går snabbt att hämta data från servern. Detta arbete visade att VectorTileDB är en teknologi som har mycket potential och bör undersökas vidare. Det finns dock flera frågor, angående användningen av VectorTileDB som en databas, som inte togs upp i detta examensarbete. Vi testade också olika sätt att visualisera data från VectorTileDB på en karta. De två lösningarna som testades var MapLibre, som också används för att skapa kartan, och Deck.gl som är ett externt bibliotek för visualisering av data. Båda lösningarna visade sig fungera väl med VectorTileDB.

Nyckelord:	Vector tiles, Vector maps, TileServer GL, VectorTileDB, MapLibre, Deck.gl, Mapbox
Sidantal:	47
Språk:	Engelska
Datum för godkännande:	

CONTENTS

Figures

1 INTRODUCTION

1.1 Motivation

A modern-day telecommunications network is an extraordinarily complex system consisting of many physical and virtual layers, with a large amount of programmatic orchestration needed to make it all work. Even while moving at high speeds the user equipment needs to be able to communicate with different base stations and provide a continuous connection for the user. Since a telecommunication network is on many levels connected to the physical environment, the use of a map gives us an apt tool for visualizing data associated with the network. Places where the network is overloaded could be visualized to give operators a clear overview of the whole network. Metadata and associated *KPIs* (Key Performance Index) could be displayed either on the map or on a graph or dashboard. Zooming in on a particular area could reveal more specific data related to that area. (Hiisilä, 2022)

The standard way of correlating KPI data with a map has been calling multiple *APIs* and aggregating the data (fig. 1). The issue here is that this is not that fast while moving around on an interactive map and having to wait for multiple responses from different APIs. The idea to improve this would be to have the KPI data integrated into the vector tiles, which are used to render the map, that are served by the tile server (fig. 2). This would allow for all the necessary data to be automatically filtered for the current map viewport and loaded into memory, allowing for fast access to the metadata (Hiisilä, 2022)

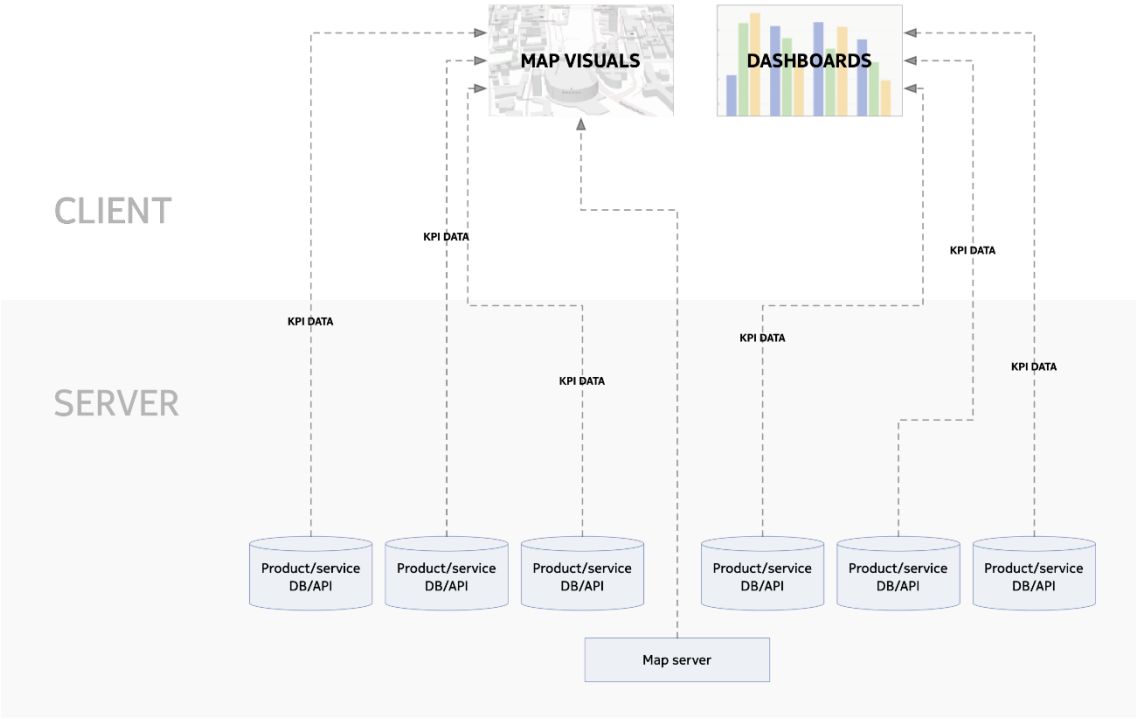

Figure 1. Correlating data with a map

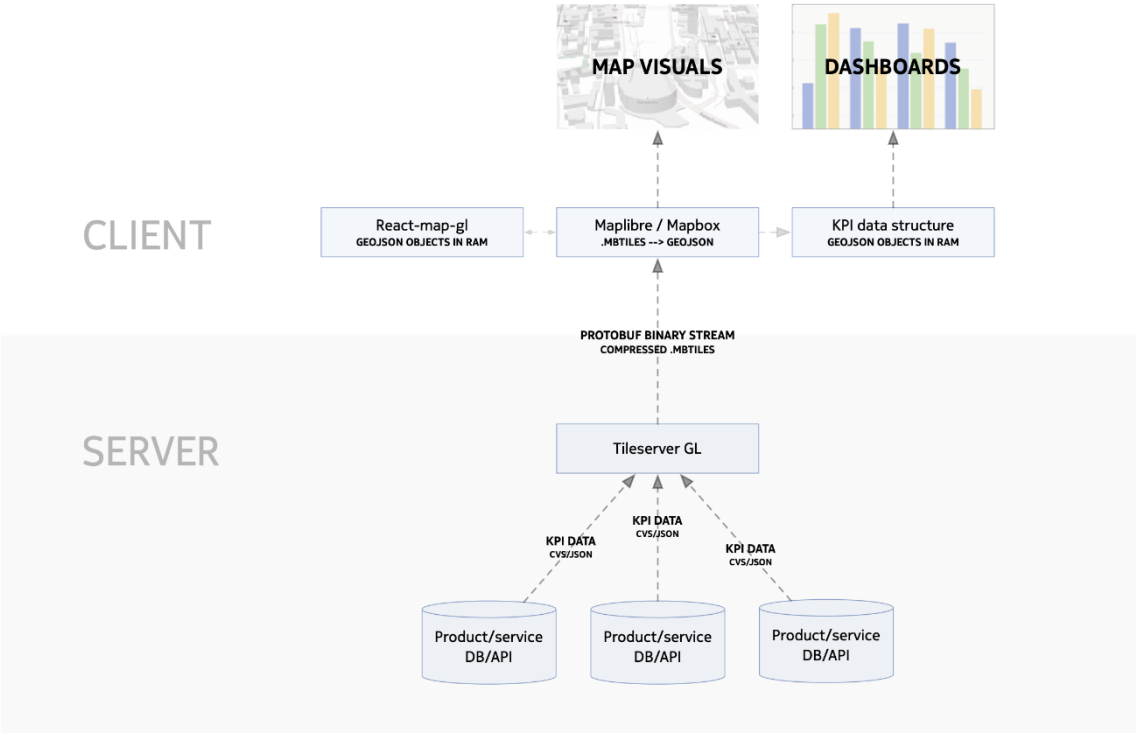

Figure 2. Using Tileserver GL and vector tiles for correlating data with a map

1.2 Background

This thesis was made at the request of Nokia's *ATG* (Advanced Technology Group). The purpose of ATG is to evaluate innovative technologies and their possible business applications through rapid prototyping.

The idea for this thesis was conceived while I was working as a trainee at Nokia ATG. The project I was working on was called *Twin-City*, where the goal was to create a digital twin of a city using *vector tile*-based maps. My job was to research and create a pipeline for creating and displaying vector tile maps.

Each feature on a vector tile-based map is represented by geometry: like a point, line or a polygon, and optional metadata. On the client-side, once a tile is inside the map viewport, the map loads all data from the visible tiles into memory, allowing for fast client-side access to that data. I realized that the vector tiles allow for any feature on the map to have any number of (key: value) pairs stored as metadata. These features could allow for vector tiles to be used as a geo-referenced database. I created a proof of concept that this idea works, and that was what led to this thesis.

1.3 Objective and Purpose

The objective of this thesis is to test how well vector tiles can be used as a database; this concept will be referred to as *VectorTileDB*. We will measure how VectorTileDB performs against an industry standard *NoSQL* database, in this case *MongoDB*. The hypothesis is that VectorTileDB will be able to outperform MongoDB when loading large datasets from the database. This is because the vector tiles served by VectorTileDB are relatively small, since they are encoded using Google's *protobuf* protocol buffers. And, since the tiles are being loaded into memory on the client side, the client has fast access to this data.

In this thesis we will also simultaneously test different ways of using vector tile data to augment an online map. The reason for this is to gain some insight into how to best

work with VectorTileDB, given that there are a few methods available for how to render the data. Testing VectorTileDB in different scenarios hopefully also gives a better understanding of its limitations. This thesis will compare two different tools used in rendering data on a map: using the underlying map library *MapLibre* versus using an external rendering library called *Deck.gl*.

1.4 Limitations

The main scope of this bachelor's thesis is to analyze the performance and ease of use of the different database and rendering methods. There are some possible limitations of VectorTileDB: how to update the database on the backend side, how to add completely new data to existing tiles and how well the solution scales. But these are not part of the scope of this thesis.

1.5 Structure

The rest of the thesis is structured as follows: The next chapter (chapter 2) will contain background information about digital maps and the relevant technologies. Chapter 3 will deal with the tools and technologies used in creating the framework for this thesis. Chapter 4 will deal with the methodology of performance testing. The final two chapters (chapters 5 and 6) will go through the results from the performance tests and the conclusions gained from the tests.

2 BACKGROUND

2.1 Digital Maps

Maps are a vital part of many applications that are used by consumers and companies daily. They offer an effective way of visualizing geo-referenced data that might otherwise be hard to explain.

The first online map service was the *Xerox PARC Map Viewer* which, in response to user input encoded in the requested URL, delivered a map as one image. The response

was either a GIF image or an HTML document. In the case of the HTML document, it included the map as an **<img>** element along with URL links for moving around and or zooming in on the map. This only allowed for minimal interaction with the map and was not particularly fast since the whole image needed to be re-generated on each new request. (Putz, 1994)

In 1998 former US vice-President Al Gore made a speech where he envisioned a "Digital Earth", a "multi-resolution, three-dimensional representation of the planet, into which we can embed vast quantities of geo-referenced data" (Gore, 1998 p.528). This was made a reality in 2005 with the launch of *Google Maps* and *Google Earth*. These maps made it possible for the user to pan and zoom without restrictions, this type of map is now referred to as *slippy maps*. Google Maps and Google Earth were developed using data Google had acquired through its purchase of the Keyhole Corporation in 2004, a digital mapping company with a vast database of mapping data and satellite images. (Dora, 2012, pp. 1-2,4).

Traditionally most map services, since the release of Google Maps, have been using *raster tile* maps, where the maps are pre-rendered on the server as multiple small tiles which are then sent to the client on demand. The client then stitches the tiles together to form the map the user interacts with. Raster tile maps are still widely used today but more services are switching to vector tile maps. (Netek, et al. 2020, pp. 1-2)

Unlike raster-based maps, that use pre-generated images to create a map, vector-based maps consist of vector representations of real-world features. The server sends the vector tiles to the client, and the client then renders the vector features in the browser. This allows for dynamic rendering of the map on the client side, which gives the user more flexibility to explore and allows for a more interactive map. Vector tiles also allow for 3D maps which offers a more immersive user experience and can further help with the visualization of various kinds of data.

2.2 Tiled Maps

Unlike earlier map services, that needed to re-generate the whole map when the user moved around, Google split their slippy maps into multiple pre-generated tiles. This meant that if a user moved around, or changed zoom level, only the missing tiles needed to be downloaded and then stitched together with the existing tiles. This greatly increased the performance of the map, compared to existing map solutions. (Netek, et al. 2020, p. 2)

Google also created the *Web Mercator coordinate system*, where the entire world can be shown as a square. This square, encompassing the entire world, is called the zero zoom-level tile. When a user zooms in one level each tile from the previous zoom level gets split into 4 new tiles. Each tile is labeled with a pair of (X, Y) coordinates, with the top left tile of each zoom level having the coordinates (0,0). This means that any tile in the coordinate system can be referenced by specifying a zoom level and a coordinate pair. (Netek, et al. 2020, pp. 2-3)

The Web Map Tile Service (WMTS) implementation standard, developed by the *Open Geospatial Consortium*, defines a standard protocol for serving geo-referenced map tiles on the web. It does not require the use of any specific tile scheme, resolution set or projection, but provides the means of defining these. It does however define addressing tiles using matrix coordinates; the top-left tile being (0, 0). (Sample & Elias, 2010, p. 15)

2.3 Raster Tiles

Raster-based maps are still quite common in online maps. The idea behind raster maps is that the data is stored as a grid of values, and each cell of the grid is a pixel on the map. Each pixel of the raster represents an area on the Earth's surface (Sample & Elias, 2010, p. 47). Raster maps uses a tile-based approach of creating maps. One tile usually contains 256x256 pixels, although other sizes are also possible, for example 64x64 or 512x512 are also in use (Netek, et al. 2020, p. 3).

The tiles of a raster-based map are pre-generated and stored on a server. The client only needs to display the tiles it receives from the server next to each other, no other processing required (Peterson, 2012, p. 4). Due to this the performance is mostly dependent on the download speed. This means that raster tiles also work well on older, not that performant, hardware. The downside of this is that it is not possible to customize the raster map on the client side, every time some change is made to the map all the tiles need to be re-generated and uploaded to the server.

Since the number of tiles quadruple at every zoom level you need close to 1 trillion tiles for the entire world at zoom level 20. By estimating the size of one tile to be around 15Kb the size of the world map, with 20 zoom levels, is around 20 Petabytes or 20 480 TB. (Peterson, 2012, pp. 5-6)

2.4 Vector Tiles

Vector tiles use the same tiling system popularized by Google, but instead of using pre-rendered images like raster-based maps, the features of the map are vector representations. These objects consist of the following data: a geometric primitive, the position of the object on the map and optional metadata; geometric primitives can be either: points, lines or polygons (Sample & Elias, 2010, p. 193). The server only sends the vector objects to the client, and the client decides how to render the features. Vector tile maps also allow for the rendering of features in 3D, and the map can be rotated and tilted to give the user a better ability to explore the map from any angle.

With client side rendering the storage needed on the server side is reduced. Creating a world map out of *OpenStreetMap's* dataset of the whole planet (60GB) yields a vector tile file of around 80-100GB. The size varies with what features are included and at what detail and at what resolutions you want the features to be visible. But it is nowhere near the 20 Petabytes required for the raster-based world map.

Another feature of vector-based maps is, since they are rendered by the client, the possibility of changing the appearance of the map dynamically. The client can modify features at will and change the overall look of the map. Since most of the front-end li-

braries for rendering vector tiles use *WebGL*, the map allows for smooth interaction while panning and zooming.

2.4.1 Mapbox Vector Tile

One of the most used vector tile specifications is the *Mapbox Vector Tile Specification* which is created by *Mapbox* and defines how to encode tiled vector data. Mapbox vector tiles use the .mvt file suffix, and are encoded using Google's protobuf protocol buffers, which are "language-neutral, platform-neutral, extensible mechanism for serializing structured data" (Google, 2022).

A Mapbox vector tile consists of geometric data and attribute data. Geometric information, such as latitude and longitude, must be converted into vector tile grid coordinates, (X, Y) pairs relative to the top left of the grid. Since this process converts geographic data into non-geographic data, there will be some simplification of the geometry. To mitigate some of this simplification of the geometry, and the visual impact of this, tiles are rendered with more complexity at higher zoom levels. (Mapbox, 2022e)

When encoding data into a vector tile all the keys and values in the dataset are separated into two sets, a set of all keys and a set of all values. Each (key: value) pair of a feature is then represented as a pair of tags pointing to the index values of these sets. In the example (fig. 3) the first (key: value) pair (`hello: world`) of the first object gets encoded with the tags 0 and 0. The first tag points to the value in the set of all keys with index 0 `hello`, and the second tag points to the value in the set of all values with index 0 `world`. The second object also has a (key: value) pair containing the key `hello`, so its first tag will point to the same index in the set of all keys as the first (key: value) pair in the first object. In large vector tiles this can save a lot of space, since most attributes are duplicated among similar features. (Mapbox, 2022e)

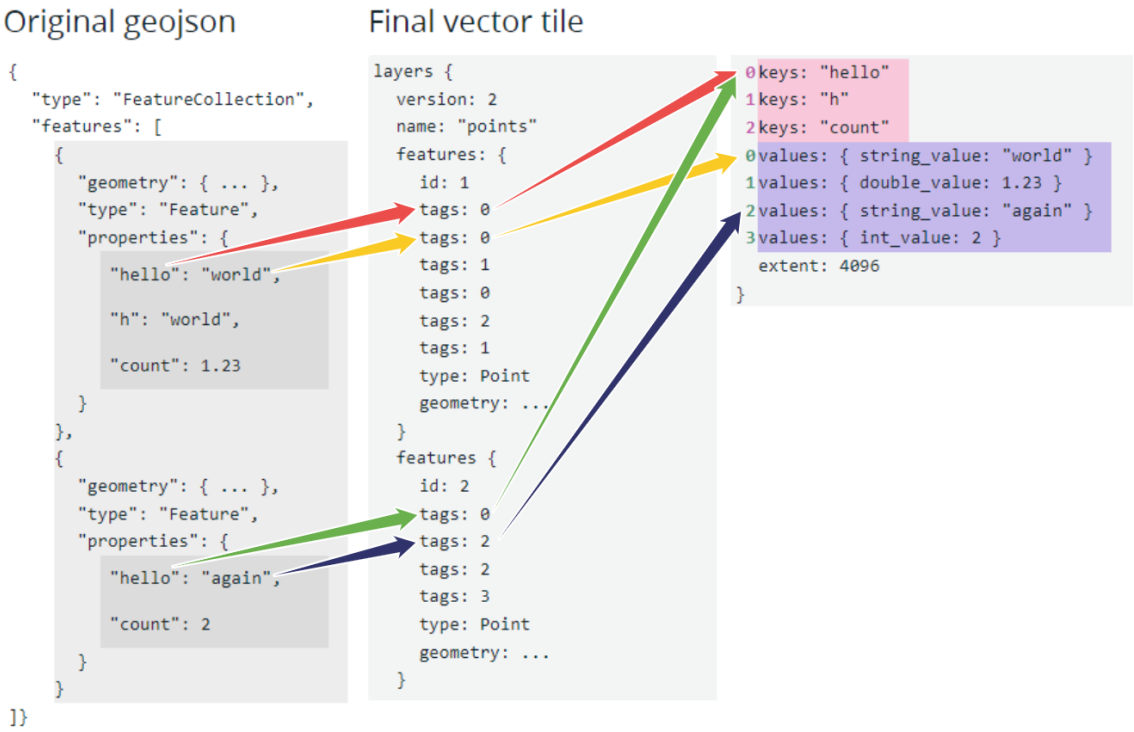

Figure 3.

2.4.2 MBTiles

The *MBTiles Specification* is created by Mapbox and specifies how to store multiple tiled data sources in a single file. One MBTiles file represents a single set of tiles and can include vector tiles, raster tiles and/or other MBTiles. MBTiles uses the .mbtiles file suffix and is technically an *SQLite* database. (Fischer, et al., 2018)

2.5 OpenStreetMap

The main goal of OpenStreetMap (OSM) is to be a collaborative project to create a free editable spatial database of the world. There are many other similar projects, but OSM is the most established and well known. Success can be attributed to a few factors. Firstly, the possibility of developing large-scale collaborative projects online, made possible due to *Web 2.0*. Secondly, the availability of GPS devices due to the rise of smart devices such as smartphones, allowing for consumers to easily collect geographic data. And lastly, the OSM project makes it easy for anyone to take part as a contributor, regardless of the skill level of the user. (Mooney & Minghini, 2017, pp. 38-39).

OSM features are made up of geometries with attached attributes, or tags. Geometries are made up of three different types: *nodes, ways,* and *relations*. Nodes represent any point on the map, like a tree, or stop sign etc. They are also often used to designate the name of a point of interest, establishment, city, or country. Ways are multiple nodes connected to form line features like roads or rivers. They can also have their starting point connected to their end point to form areas like parks or buildings. Relations are collections of multiple nodes and/or ways that form larger features. The most common type of relations are multipolygons, which are polygons that can have holes in them, for example: a building with an inner courtyard has an inner and outer wall, both represented as ways together forming a multipolygon. Tags are (key: value) pairs that describe a feature. Every node, way or relation can have any number of tags attached to it to describe different attributes about that feature. There are no restrictions on what tags to use, but the OSM wiki defines many conventions about how to use tags. (Minghini & Frassinelli, 2019, pp. 2-3)

3 DESIGN AND IMPLEMENTATION

The testing framework built for this thesis is based upon work I did for its predecessor, the Twin-City project. The frontend is written in Typescript, due to better type checking and error handling compared to vanilla JavaScript, using the React framework. The backend API is written in Python using the FastAPI framework. The tile_generator tool, which is used for converting OSM data into MBTiles, is written in Python and was also originally written by me for the Twin-City project. Tile_generator will be explained in more detail in the next section.

All tests will be performed on a Lenovo T14 laptop with an Intel i5 10th gen processor. And all tests will be done using the Microsoft Edge browser and the Edge DevTools.

3.1 Creating Vector Tiles

3.1.1 GeoJSON

GeoJSON is a format for encoding geographic data using JavaScript Object Notation (JSON). Vector tiles can be created out of GeoJSON data and when querying the vector tiles, with MapLibre, the data is returned in GeoJSON format.

A GeoJSON object must include a member with the name `type`, there are two categories of types: GeoJSON types and geometry types. The following geometry types are available:

- Point
- MultiPoint
- LineString
- MultiLineString
- Polygon
- MultiPolygon
- GeometryCollection

GeoJSON types can be either `Feature`, `FeatureCollection` or a valid geometry type. A `Feature` object is a spatially bounded thing. It has `geometry` and `properties` members. The value of `properties` is a JSON object with metadata about the feature. The value of `geometry` is a JSON object that has a `type` member with a valid geometry type. A `FeatureCollection` has a `features` member whose value is a JSON array. Every element of the array is a `Feature` object. (Butler, et al., 2016, pp. 4-5,11-12)

All GeoJSON objects except for GeometryCollection must have a `coordinate` member, the value of this member is an array of positions. A position is an array of numbers where the first two elements are longitude and latitude, an optional third element may be added to signify elevation. (Butler, et al., 2016, p. 6)

According to (Butler, et al., 2016, pp. 7-9) the different geometry types have different specifications on what their `coordinate` member must include:

- Point: coordinate is a single position
- MultiPoint: coordinate is an array of positions
- LineString: coordinate is an array of two or more positions
- MultiLineStrings: coordinate is an array of LineString coordinate arrays
- Polygon: coordinate is an array of linear rings.
    - A linear ring is LineString that has four or more positions, and the first and last positions are the same. This forms a closed area.
    - The first linear ring must be the exterior ring and all the following members of the array must be interior rings. These interior rings form holes in the polygon.
    - The first linear ring must be wound counterclockwise, while all the holes must be wound clockwise.
- MultiPolygon: coordinate is an array of Polygon arrays.
- GeometryCollection: does not have a coordinate member, instead it has a `geometries`: member that can be an array of different geometry types.

3.1.2 Tippecanoe

Tippecanoe is a command line interface tool created by Mapbox for converting GeoJSON data into MBTiles files. The tool has many options for specifying how to convert the data, filtering features, what zoom levels to generate tiles at, different algorithms for dropping or merging features, correcting bad source geometry and more. (Mapbox, 2022d)

Tippecanoe also comes with some additional useful tools:

- **tile-join**: A tool for copying and merging vector MBTiles files. Can also be used for updating features on existing tiles with attributes from a CSV file. One of the limitations of tile-join is that when merging vector tiles all tiles must have the same maximum zoom level for it to work.
- **tippecanoe-enumerate**: Lists all tiles defined in an MBTiles file.

- **tippecanoe-decode**: Can turn a MBTiles file, or even individual tiles, back into GeoJSON.
- **tippecanoe-json-tool**: Can extract GeoJSON features from a larger JSON file.

3.1.3 Tile_generator

Tile_generator is a tool I created at ATG for the Twin-City project. It is written in Python and is used for bulk conversion of OSM data into MBTiles. The process is divided into five parts: extract features based on geometry, process and clean the data, extract features into individual layers, convert the layers into MBTiles files and finally merge the MBTiles files into one MBTiles file.

For extracting features based on geometry tile_generator uses *ogr2ogr*, a tool from *GDAL* (The Geospatial Data Abstraction Library). Ogr2ogr can extract all features of a certain geometry from osm.pbf files into individual GeoJSON files. The default scheme of tile_generator only cares about points, linestrings and multipolygons. But there are also multilinestring and other_relations type of geometries available in OSM datasets. Multilinestrings contain features consisting of multiple linestrings, like bus and train routes that go from point A to point B and back to point A. Other_relations contain collections of multiple types of features, for example: routes including stops along the route, buildings including the roads and parking lots associated with that building, multiple surveillance cameras in an area and street signs etc.

After the geometries have been extracted into GeoJSON files they are then processed and cleaned. This is done by running them through a parser that fixes some formatting issues and adds or converts some necessary metadata in certain features. The parser is built using the Python *ijson* library, for streaming JSON files in Python. The process of streaming JSON files was necessary since some of the JSON files can be many tens of GB in size and loading a 2GB JSON into Python as a dictionary already takes over 10GB of RAM.

The parsed GeoJSON data is then split into layers based on feature types, meaning GeoJSON files containing only features relevant to that layer. For example: all the build-

ings, roads, parks, waterways, forests etc. are all put into separate files. This is done using the Unix *grep* tool. The GeoJSON layers are then converted into MBTiles using Tippecanoe. Finally, all the individual MBTiles layers are merged into one MBTiles file using Tippecanoe's tile-join tool.

It is possible to define different schemes for tile_generator, meaning which features will extracted into layers, what arguments will be used when converting layers into MBTiles etc. The different settings for each layer are defined in a configuration YAML file.

3.1.4 Styling Vector Tiles

Vector tiles styles are defined in a JSON file. The styles can be created using the open-source visual editor *Maputnik*, which allows the user to edit styles using a graphical interface (Maputnik, 2022).

A style consists of *sources* and *layers*. A source is the data that gets rendered on the map, usually a vector source is used but it can also be one of the following types: raster, raster-dem, GeoJSON, image or video (Mapbox, 2022c). The example (fig. 4) shows how to add a vector source in the style file. When defining a source with mbtiles:// the server will look for a local file when starting up. But it is also possible to load in a vector tile that is being served by another tile server, by using http:// instead of mbtiles://, in that case the suffix also has to be changed from .mbtiles to .json.

Layers define how features are rendered, a layer must be one of the following types: background, fill, line, symbol, raster, circle, fill-extrusion, heatmap, hillshade, sky (Mapbox, 2022b). Layers are written using Mapbox expression syntax (MapLibre, 2022b), which is very similar to the syntax of the Lisp programming language. The example (fig. 5) defines how buildings are rendered; for example, `fill-extrusion-height` defines how tall each building should be rendered, and the value is taken from the attribute `height` of each feature. The client can dynamically add and modify sources and layers at runtime.

```
"sources": {
  "composite": {
    "url": "mbtiles://finland.mbtiles",
    "type": "vector"
  },
```

Figure 4. Defining a source

```
{
  "id": "building-extrusion",
  "type": "fill-extrusion",
  "source": "composite",
  "source-layer": "building",
  "paint": {
    "fill-extrusion-color": [
      "case",
      ["boolean",["!=",["feature-state","color"],null],false],
      ["feature-state","color"],
      "hsla(218, 78%, 15%, 0.4)"
  ],
    "fill-extrusion-opacity": 0.75,
    "fill-extrusion-height": ["get", "height"]
  }
},
```

Figure 5. Defining a layer

3.2 Tileserver GL

Tileserver GL is an open-source server for serving vector tiles and associated style files. It can also be used to serve fonts and sprites to use on the map. It is also possible to render raster tiles out of the styles that the tile server serves (Klokan Technologies GmbH, 2016a). According to (Netek, et al. 2020, p. 11) Tileserver GL compared very favorably against other tile servers and was the quickest at downloading a single tile from the server while also performing very well on other metrics including map loading time, number of requests on the server and size of downloaded data.

Vector tiles, and other sources, can be linked to from other tile servers, meaning that it is possible to separate the concerns, having one tile server that only serves the styles and/or the tiles necessary for the actual map. While another tile server can act as a database for optional tiles that can be used on demand.

Any tile inside a vector tile, served by Tileserver GL, can be accessed using a simple HTTP request to the endpoint …/data/{mbtiles}/{z}/{x}/{y}.pbf where {mbtiles} is the name of the MBTiles file served by the server, {z} is the zoom level, and {x} and {y} are the coordinates of the tile (Klokan Technologies GmbH, 2022b). This will return a protobuf encoded tile, that will then need to be decoded.

3.3 Database

3.3.1 VectorTileDB

The concept of a VectorTileDB can be defined as: a tile server acting as a database, serving any number of *data tiles*, where the individual tiles can be accessed by a client-side map library or queried using an HTTP request. The difference between a normal vector tile and a data tile is that a normal vector tile can be thought of as a collection of geometric features, with some optional added metadata. While a data tile is a collection of geo-referenced data. The reason for making this distinction is simply to try to enforce a good practice of separating the concerns, having separate vector tiles used for rendering the map and storing data.

To access the data inside the tiles MapLibre has two functions available: *queryRenderedFeatures* and *querySourceFeatures*. The difference between the two is that queryRenderedFeatures only queries the features that are inside the viewport and visible when querying. While querySourceFeatures can query all features that are inside the viewport, even if they are not visible on the map when querying. QuerySourceFeatures is still dependent on zoom level, meaning it will only query those features that could be visible at the current zoom level. It will also return all the features of visible tiles, meaning if a tile is only partially visible inside the viewport the features that are not inside the viewport will still be returned. In contrast queryRenderedFeatures only gets the features that are inside the viewport. (MapLibre, 2022a)

When loading a data tile as a source there are a few things that need to be considered. A source will not be queryable unless there is a style layer that references that source. Data loaded into RAM will get garbage collected if it is not referenced anywhere, this will

also happen if all layers that reference a source have visibility set to none. A solution to this is to add a *dummy layer* that references the source but does not render anything, by for example setting the source-layer to a nonexistent layer. This prevents the garbage collector from removing the data from memory and allows for the data in the source to be queried.

One of the aspects of the VectorTileDB that can be both positive and negative, depending on the scenario, is that the data is always geographically filtered, only data inside the viewport is accessible. But there are also ways around this. When querying data using the querySourceFeatures function, it always returns all the data of partially visible tiles. This combined with the fact that it is possible to define at which maximum zoom levels tiles will be rendered at, means that it is possible to query data not inside the viewport from tiles rendered at lower zoom levels, if part of the tile is inside the viewport. In practice it's possible to render tiles with a maximum zoom level set to zero (0), since the zero tile includes the whole map, all data inside that tile will be queryable anywhere on the map at any zoom level.

3.3.2 MongoDB

MongoDB is a NoSQL database. Communication with MongoDB happens through a backend API, written in Python using the FastAPI framework. The API consists of three endpoints /stressmongo, /stressdeck and /stressgeofilter. The /stressdeck endpoint serves the small set dataset. The two others /stressdeck and /stressgeofilter both serve the large dataset, but /stressdeck contains only the building data for the area that is being tested, while /stressgeofilter contain all the buildings in Finland. When querying the /stressgeofilter endpoint it needs a multipolygon as input, and only the buildings inside that multipolygon are returned from the database. This is done using the MongoDB's built-in geoWithin function. The example (fig. 6) shows how the geoWithin function works with a hard coded multipolygon.

```
db.all_buildings.find({
    "geometry": {
        "$geoWithin": {
            "$geometry": {
                "type": "MultiPolygon",
                "coordinates": [[[
                    [24.609035192956696, 60.283349332509545],
                    [24.961455999842144, 60.28360461485241],
                    [24.96076935433365, 60.15229739678503],
                    [24.60886353158091, 60.152297396785286],
                    [24.609035192956696, 60.283349332509545]
                ]]]
            }
        }
    }
}
```

Figure 6. Geo filtering features from MongoDB with geoWithin

3.4 Rendering Libraries

3.4.1 MapLibre GL JS

MapLibre GL JS is an open-source JavaScript library used for displaying maps in the browser. It allows for fast, dynamic, and interactive maps due to GPU-accelerated vector tile rendering (MapLibre, 2022d). The MapLibre ecosystem is a result of Mapbox-gl-js abandoning their open-source license and adopting a proprietary license with the release of version 2.0 in 2021. This led to the creation of the MapLibre GL JS library, from here on referred to as MapLibre, which was forked from Mapbox-gl-js version 1.0 (MapLibre, 2022c).

Rendering data dynamically at runtime with MapLibre can be very costly performance wise. According to "Strategies for improving performance" by Mapbox (2022a) it is recommended to use the *feature-state* functionality for this. This can be done by accessing the map object of MapLibre and calling the *setFeatureState* function. This allows for injecting data as (key: value) pairs into a feature at runtime and avoids re-parsing all the geometries of the features when the state changes. To use the feature-state functionality, the features must be referenced via an id. If a feature contains an attribute with the

key `id`, it will automatically be used as the id. Otherwise, if features do not contain an `id` attribute, any attribute of a feature that is an integer, or string that can be cast to an integer, can be promoted to be used as an id. This is done using the *promoteId* attribute when defining a source. In the example (fig. 7) all the features of the `composite` source that are in the source layer `building` will have their `osm_id` attribute promoted to Id. (MapLibre, 2022a)

```
"composite": {
  "url": "mbtiles://finland.mbtiles",
  "type": "vector",
  "promoteId":{"building": "osm_id"}
},
```

Figure 7. Using promoteID when adding source

The setFeatureState function takes two objects as arguments. The first object defines which feature we want to target by the attributes: `source`, `sourceLayer` and `id`. While the second object defines what attributes to add to the feature (MapLibre, 2022a). In the example (fig. 8) the feature-state `color: green` is added to the feature with `id: 12345` in the source layer `building` that is found in the source `composite`.

```
map.setFeatureState(
  {
    source: "composite",
    sourceLayer: "building",
    id: 12345
  },
  {
    color: "green"
  }
)
```

Figure 8. Setting a feature-state

Feature-state attributes can only be used with paint properties that support data-driven styling (MapLibre, 2022a). In the example (fig. 9) a `case` expression is used to test if

the feature-state `color` exists on a feature. The predicate, meaning the test condition, of the `case` states that: if the feature-state `color` is not equal to null then return true, otherwise return false. The first line after the predicate of the `case` defines what happens if the `case` returned true, in this case the value of the feature-state `color` is returned. The final line defines what happens if the predicate of the `case` returned false, in this case the string `red` is returned. When rendering features that have this rule applied to it, MapLibre will test for the presence of a feature-state `color` and render it accordingly.

```
"paint": {
  "fill-extrusion-color": [
    "case",
    ["boolean",["!=",["feature-state","color"],null],false],
    ["feature-state","color"],
    "red"
],
```

Figure 9. Case expression for rendering colors

The library *react-map-gl* is part of the *Vis.gl* suite, developed by Uber (Uber, 2022), provides React wrappers for Mapbox-gl-js and its forks, including MapLibre. This allows for a functional reactive programming style, that React lends itself to, to be used when building a map client based on Mapbox-gl-js, or one of its forks.

3.4.2 Deck.gl

Deck.gl is part of Uber's Vis.gl suite (Uber, 2022) and is a highly performant web-based visualization framework, built with scalability and usability in mind. It is aimed at being able to handle large-scale dataset, while being easy to use for both developers and designers. (Yang, 2019, p. 1)

Deck.gl is built on top of the layer composition concept. The user can compose layers together as needed to get the desired visualization quickly and easily (Yang, 2019, pp. 1-2). Deck.gl offers a collection of pre-defined layers for different use cases that can

show data in diverse ways: arcs, heatmaps, polygons, scatterplot etc. It is also possible to create your own custom layers through creating a composite layer, subclassing a layer, or creating your own layer from scratch, which requires WebGL and shader programming (Deck.gl, 2022).

Deck.gl is developed in parallel with react-map-gl, which is also part of the Vis.gl suite, and the two work seamlessly together. This allows for the possibility to easily create React style Deck.gl components for adding new layers of data or to animate features on top of a map built with react-map-gl. The example (fig. 10) shows how to create a React component that creates a Deck.gl GeoJSON layer.

```
const DeckBuildingsLayer: FC<props> = ({ buildings }) => {
  return new GeoJsonLayer({
    id: "geojson-layer",
    data: buildings,
    pickable: true,
    stroked: false,
    filled: true,
    extruded: true,
    getElevation: (d: { properties: { height: number } }) => d.properties.height,
    getFillColor: (d: { properties: { color: string } }) => d.properties.color,
    getLineWidth: 100,
  });
};
```

Figure 10. Component for creating Deck.gl GeoJSON layer

4 PERFORMANCE TESTING

4.1 Methods

Stress testing will be done on a map encompassing parts of the Helsinki Metropolitan area. The area chosen contains around 85 000 buildings, metadata for these buildings will be fetched from an external database and rendered on the map. The scenarios that will be tested are split up into two categories: *small set* and *large set*. The small set scenarios will test augmenting already existing features on the map. Rendering will therefore be done only with MapLibre, as it is not possible to use Deck.gl to modify existing

MapLibre features. While the large set scenarios will test adding new features to the map, this will include using both MapLibre and Deck.gl as the rendering libraries.

Testing will be done in the *Microsoft Edge* browser and using the browsers integrated *DevTools*. The *Performance tool* (fig. 11) lets us monitor Network, CPU, GPU and more. For this thesis we are interested in the time it takes for the features to be fetched from the database, processed, and rendered on the map. DevTools also allows for disabling of the cache, so the results will not be skewed because of it.

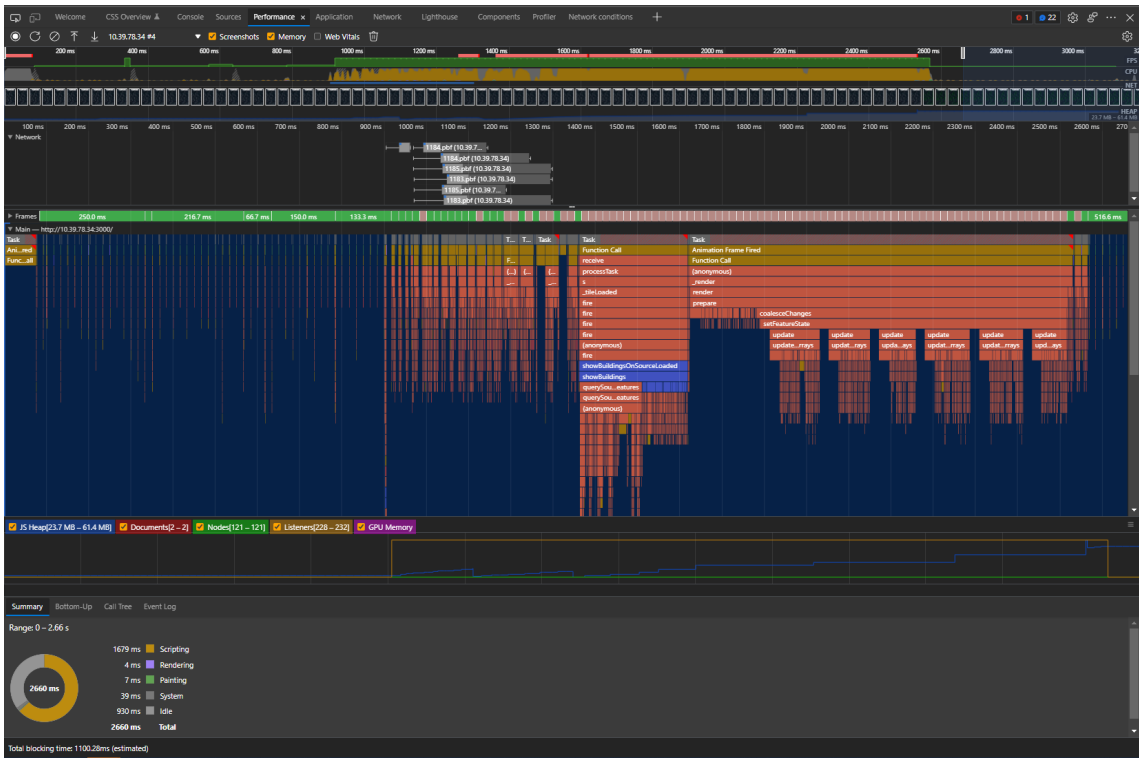

Figure 11. Edge Dev Tools Performance tool

Each scenario is run 10 times and an average score is calculated from all the runs. Initially a reset map button was implemented to reset the map between runs. But this proved to be problematic, since there seems to be a clear discrepancy between runs started after a complete page reload and those started after a soft reset. The soft reset yields on average around 15-20% faster load times on the VectorTileDB small set scenario, and around 10% faster load times on the MongoDB small set scenario. Because of this it was decided to do a complete page reload before all runs.

There is a small difference in the number of features loaded in the VectorTileDB small set 84 356 features compared to the other scenarios where 84 690 features are loaded. This is due to the geometry of the features in the large set being polygons and the features in the small set being points. If the border between two tiles intersects a polygon, then that feature can be queried from both tiles. But a point will be on one side of the border between the tiles and only be part of that tile.

For these tests, the MongoDB dataset has been optimized via *geo-filtering* the data, meaning that the database only contains those buildings that are needed for the tests. This optimal MongoDB database is on average around 40% faster (fig. 12) than using MongoDB's geoWithin function, for filtering the data by geometry, when querying the database.

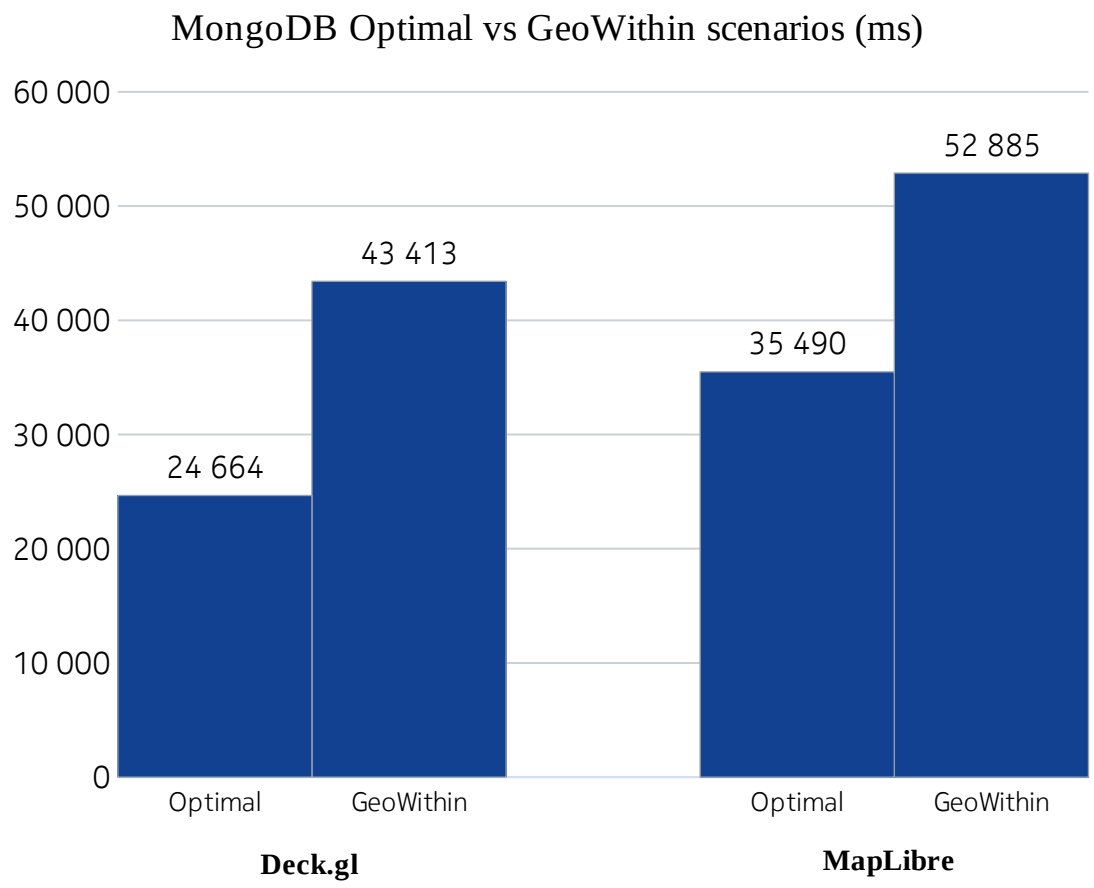

Figure 12.

4.2 Small Set Scenarios

The small set scenarios tests how to augment existing features on the map using MapLibre as the rendering library. Since the existing features of the map are rendered using MapLibre they cannot be modified using Deck.gl.

The dataset only contains the relevant metadata and/or geometry needed for augmenting features. The metadata was reduced into only the `osm_id`, for connecting the data to a feature on the map, and a `color` attribute, which defines in what color the building should be displayed. To save space in the VectorTileDB dataset, the polygon geometry was reduced into a single point. This was done by taking the first coordinate position of the features geometry and converting it into a point. Since the MongoDB dataset is not concerned with geo-filtering, the dataset includes no geometry.

The VectorTileDB small set (fig. 13) contains a bit more information than the MongoDB small set (fig. 14), since it needs some geographical data to be able to place it on the map. While the geo-filtered MongoDB small set only needs the `osm_id` attribute to be able to match with an already existing feature. If one also wanted a way to filter MongoDB data by geometry, the same coordinate data would need to be added to the database.


```
{
  "type": "Feature",
  "properties": {
    "osm_id": "184758483",
    "color": "green"
  },
  "geometry": {
    "type": "Point",
    "coordinates": [
      24.697266,
      60.239811
    ]
  }
}
```


Figure 13. VectorTileDB small set feature

Figure 14. MongoDB small set feature

4.2.1 VectorTileDB

In this scenario a data tile, containing the Finland building dataset, is loaded on demand from the tile server and then added as a source. Once the source has been loaded into memory it is queried using the *querySourceFeatures* function. Queried features are looped through and matched with existing features, via the `osm_id` tag, setting a feature-state `color` to each matched feature (fig. 15).

```
const features: GeoJSON[] = map.querySourceFeatures("finland_data_tile", {
  sourceLayer: "data_tile",
  filter: undefined,
});

features.forEach(({ properties }) => {
  map.setFeatureState(
    {
      source: "buildings",
      sourceLayer: "building_color",
      id: properties.osm_id.toString(),
    },
    {
      color: properties.color,
    }
  );
});
```

Figure 15.

4.2.2 MongoDB

In this scenario the MongoDB API is queried for all its features, which are returned as a list GeoJSON objects. And, similarly to the VectorTileDB scenario, the features are looped through setting a `color` feature-state for each existing feature with a matching `osm_id` (fig. 16).

```
const data: Response = await fetch("http://127.0.0.1:8000/stressmongo/");
const buildings: Building[] = await data.json();

buildings.forEach((element: Building) => {
  map.setFeatureState(
    {
      source: "buildings",
      sourceLayer: "building_color",
      id: element.osm_id.toString(),
    },
    {
      color: element.color,
    }
  );
});
```

Figure 16.

4.3 Large Set Scenarios

The large data sets consist of complete building features, extracted from the Finland OSM dataset. Each feature has one `color` attribute added to it. The MongoDB large set element (fig. 17) is identical to the VectorTileDB element, except for MongoDB adding a unique MongoDB generated `_id`, which adds an extra 50KB/element.

Figure 17. MongoDB large set element

4.3.1 VectorTileDB with Deck.gl

In this scenario a data tile, containing the Finland building dataset, is loaded on demand from the tile server and then added as a source. Once the source is loaded into memory, it is queried using the querySourceFeatures function for all features inside the viewport. The resulting collection is a valid GeoJSON object, and can be passed to a Deck.gl component, without any extra processing. The Deck.gl component creates a GeoJSON layer that is rendered by Deck.gl on the map.

4.3.2 VectorTileDB with MapLibre

In this scenario a data tile, containing the Finland building dataset, is loaded on demand from the tile server and then added as a source. Once the source is loaded into memory a style layer, that defines how that source should be styled, is added.

4.3.3 MongoDB with Deck.gl

In this scenario the MongoDB API is queried for all features, which are returned as a list of GeoJSON objects. The list is then passed into a Deck.gl component, that creates a GeoJSON Deck.gl layer. Deck.gl renders this layer on the map.

4.3.4 MongoDB with MapLibre

In this scenario we query the Mongo API for all features, which are returned as a list GeoJSON objects. The features are then added as a GeoJSON source. Once the new source has been added a style layer, that defines how that source should be styled, is added.

5 RESULTS

5.1 Network Performance

Of the two tested backend solutions VectorTileDB and MongoDB, VectorTileDB has a much lower network overhead when fetching data. Comparing transferred data (fig. 18) we can see that the VectorTileDB small set is 790 KB in transferred data, while the MongoDB small set is 6 800 kB, the MongoDB set being 8,6 times larger. When comparing the large sets, the MongoDB set is almost 14,9 times larger than the VectorTileDB set, 37 200 kB and 2 500 kB, respectively. One of the main reasons for the VectorTileDB sets being so much smaller than the MongoDB set, is due to the use of Google's protobuf technology for encoding tiles. Compared to the JSON data being sent from MongoDB, the protobuf encoded vector tiles scale better.

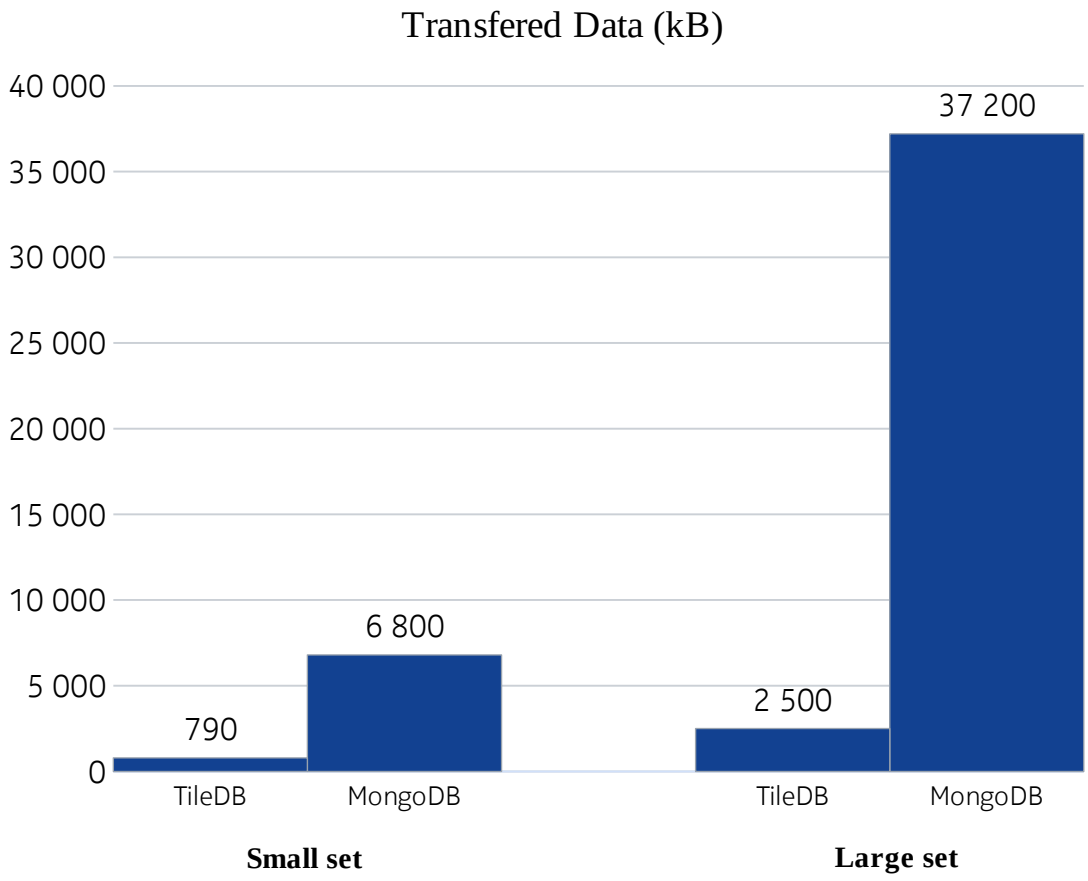

Figure 18. Amount of transferred data over network for each scenario

The difference in size between data from the two databases (fig. 18) also leads to a noticeable difference in network transfer times. Looking at the results (fig. 19) MongoDB small set took almost 7 times longer compared to the VectorTileDB small set, 3 407 ± 1% ms vs. 509 ± 25% ms respectively. Looking at the large set MongoDB took about 34 times longer compared to VectorTileDB, 24 538 ± 1% ms vs 720 ± 17% ms respectively. Other than the small size, MongoDB takes some time when collecting data, indexing and sending it to the API, whereas with VectorTileDB each tile is a separate request to the tile server, and the tiles are then decoded by the client.

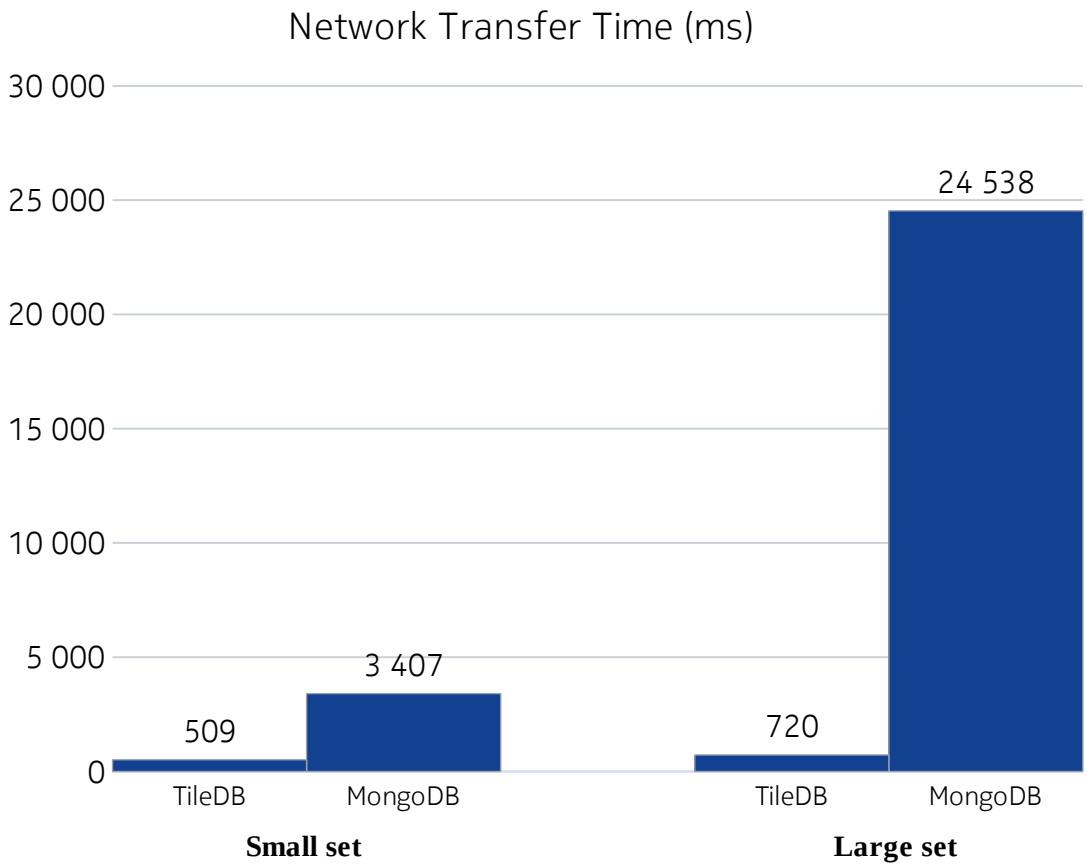

Figure 19. Amount of time for transferring data over network for each scenario

5.2 Processing and Rendering

The processing and rendering statistics are computed by taking the total runtime – network runtime. Looking at processing and rendering times (fig. 20), the most performant way of showing data on the map was creating a Deck.gl GeoJSON layer, with data from MongoDB. Using the same Deck.gl method with VectorTileDB did not perform as well, taking almost 4 times longer, 620 ± 4% ms vs. 2 418 ± 31% ms respectively. This is mostly due to the added overhead of the VectorTileDB approach: having to first add a new source, wait for the source to be fully loaded, query the source and then pass the data to Deck.gl. Whereas the data from a MongoDB database does not have to be processed in any way and can just be passed to Deck.gl. This same pattern can also be observed in the small set test, where the VectorTileDB scenario took almost 2 times longer than the MongoDB scenario, 2 712 ± 13% ms vs 1 570 ± 3% ms respectively.

The one outlier in this test was using MongoDB to create a new GeoJSON source, which took over 10 seconds to process and render. This also shows how effective vector sources are compared to GeoJSON sources.

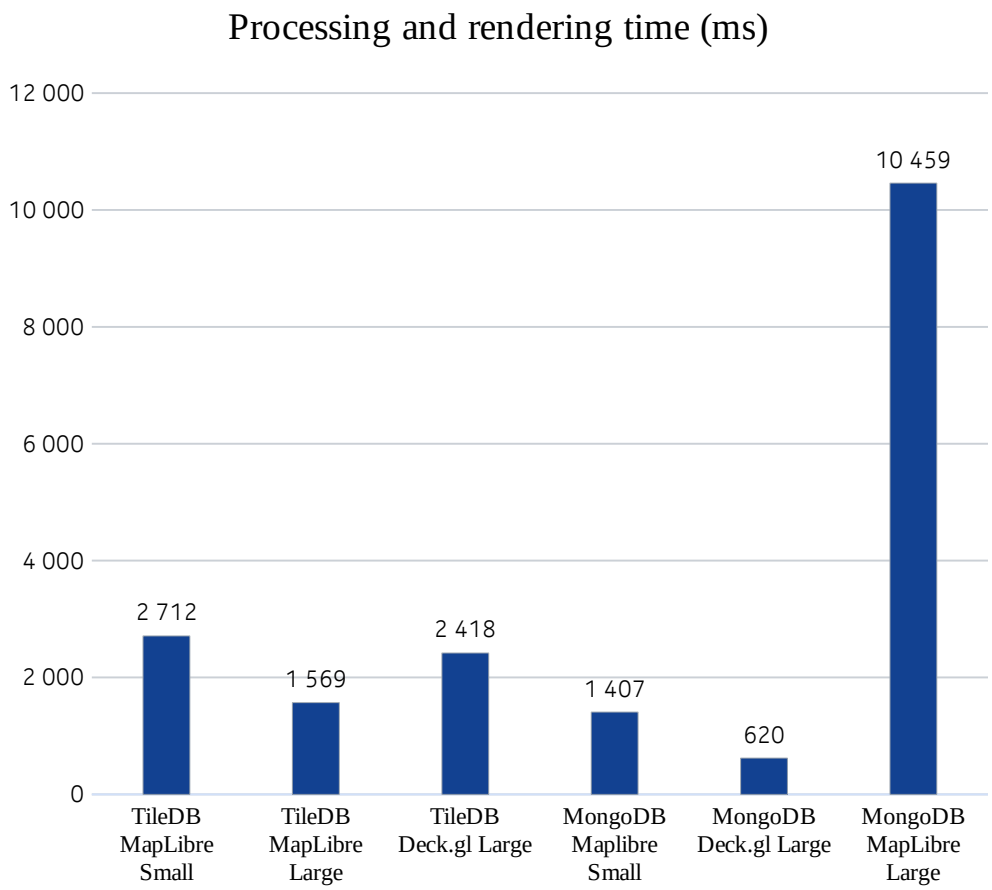

Figure 20. Process and rendering times for each scenario

5.3 Small set

The process for both VectorTileDB and MongoDB small set scenarios is to fetch data from the database and iterate through each feature returned, setting feature-states for matching features found on the map. The code for the MongoDB implementation is shorter and more concise than VectorTileDB's implementation. And MongoDB is a bit faster in rendering and processing time. This lead is severely negated when looking at network speed, where VectorTileDB is a lot faster compared to MongoDB. Looking at the total runtime of both scenarios (fig. 21), VectorTileDB has an average of 3 221 ± 12% ms/run vs MongoDB's 4 814 ± 1% ms/run , VectorTileDB being about 1.5 times faster.

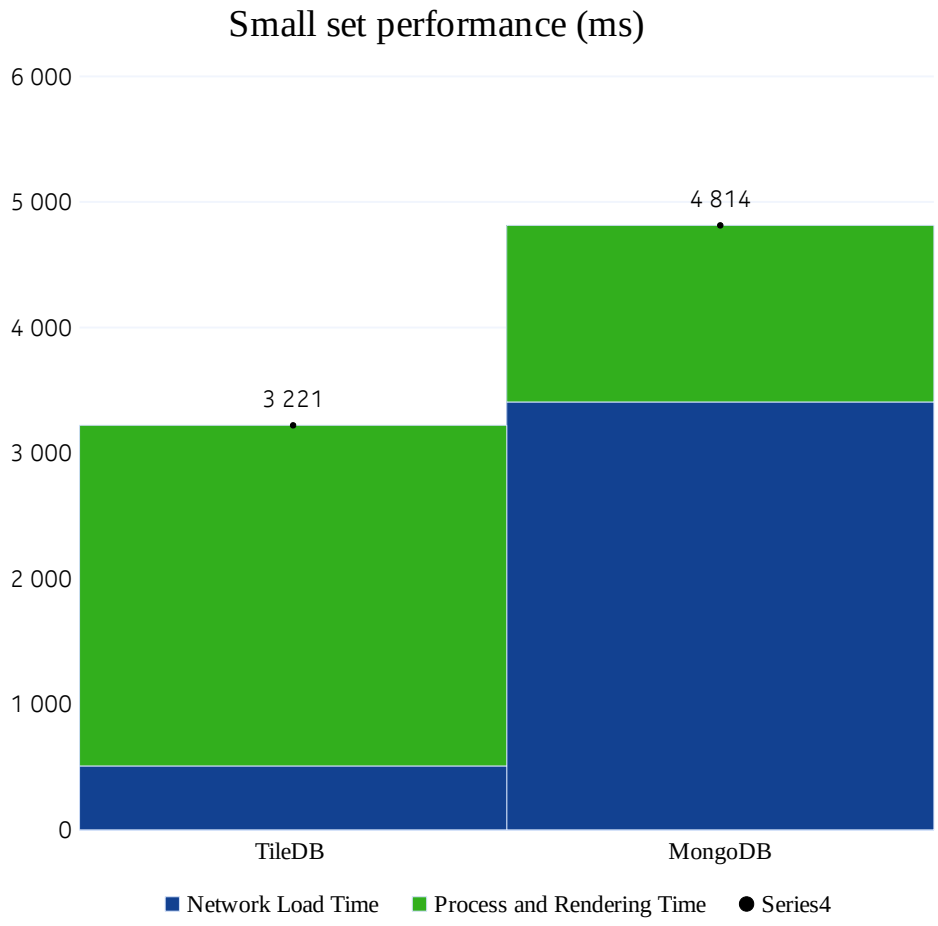

Figure 21. Small set performance

5.4 Large Set

The results of the large set performance tests show a dramatic difference in average runtime between the two database solutions (fig. 22), with the VectorTileDB scenarios averaging a runtime of 3 083 ± 27% ms with Deck.gl and 2 343 ms ± 26% , while the MongoDB scenarios have an average runtime of 24 664 ms ± 1% with Deck.gl and 35 490 ± 2%. If we discard the worst performing large set scenario, MongoDB with MapLibre, and only compare the MongoDB with Deck.gl scenario to the VectorTileDB scenarios, VectorTileDB is around 9-10 times faster.

The best performing large set scenario is the MapLibre with VectorTileDB scenario (fig. 22), with an average runtime of 2 340 ± 25% ms/run, this is due to it having a low performance runtime compared to the other VectorTileDB scenarios. It only needs to load in a new source and add a style layer. Compared to the other VectorTileDB scenarios that need to do some querying of the data before it can be rendered.

The MongoDB with MapLibre scenario is clearly the worst performing scenario, being over 14 times slower than the fastest scenario, VectorTileDB with MapLibre. It combines the slow network speed of MongoDB on larger datasets with a slow process and rendering runtime, due to the lengthy process time of creating a GeoJSON source. This highlights how fast using a vector tile source is compared to using other types of sources. The result is the same, a source that can be used by the map library, but the GeoJSON source is far less performant than using a vector tile source.

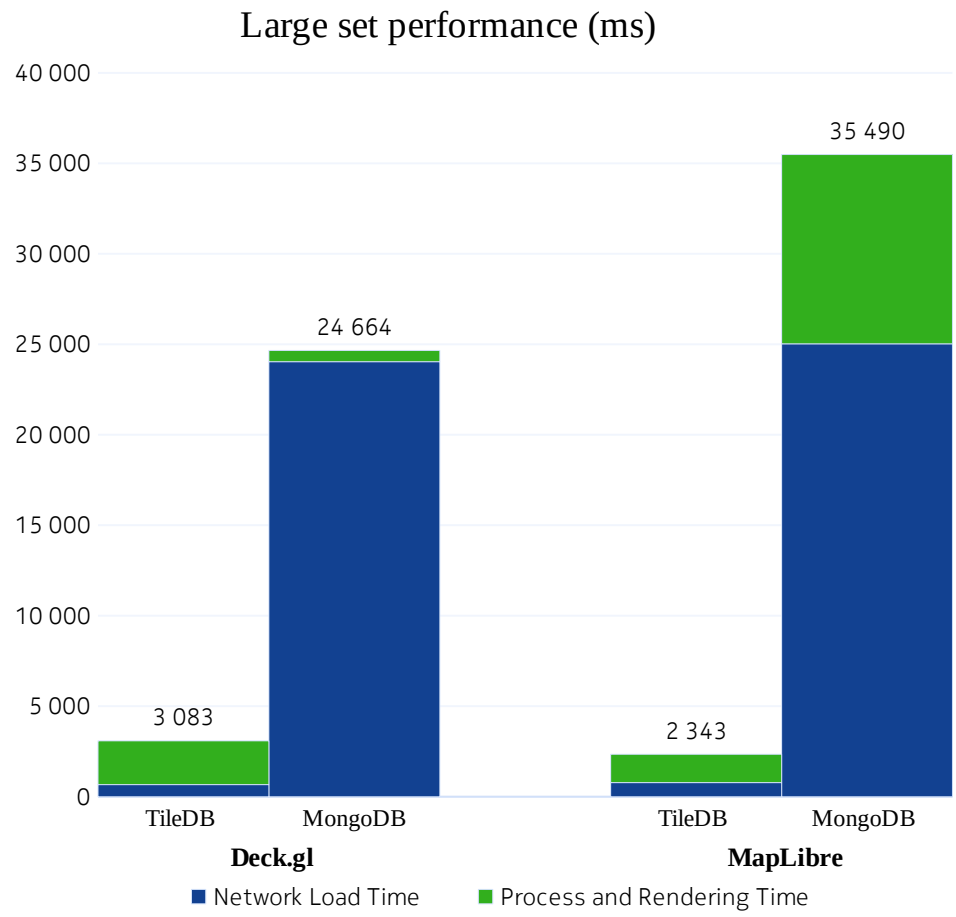

Figure 22. Large set performance

5.5 Overall

In all scenarios VectorTileDB outperformed MongoDB in total runtime (fig. 23). Even though MongoDB scenarios have a shorter process and rendering time, except in the case of adding data as a GeoJSON source, VectorTileDB still vastly outperforms MongoDB on network load time, leading to a much faster average runtime. It also had a much higher uncertainty, around 12% on the small set and 26% on the large sets, compared to MongoDB, around 1% uncertainty on all scenarios. But the slowest measured VectorTileDB run is still faster than the fastest measured MongoDB run.

If we look at the average total runtime for all VectorTileDB scenarios, it is about 2-3 seconds, while the fastest MongoDB based scenario, MongoDB small set, has a runtime of about 5 seconds. MongoDB large set scenarios have a runtime of 25-35 seconds, meaning they are about 10 times slower than the average VectorTileDB scenario. The major factor in this seems to be the conversion of (keys:values) into index tags, and also the use of the protobuf technology while encoding the vector tiles, leading to much smaller data being sent from the server. Also, the amount of data in VectorTileDB's large set is about 3 times larger than the small set, while the network transfer time is only about 1,5 times larger. From this we can conclude that VectorTileDB scales well on network speeds.

The tests also show that it is more performant to add new features to the map, instead of augmenting already existing ones. The VectorTileDB scenario for augmenting existing features, VectorTileDB small set with MapLibre, is about 1,3 times slower than the VectorTileDB scenario for adding new features, VectorTileDB large set with MapLibre. Even though the large set is over 3 times larger than the small set (fig. 18), this is because the VectorTileDB with MapLibre large set scenario does not have to query the tiles at all, only add a source and a layer, leading to faster processing and rendering times.

With MongoDB however, the augmenting of existing features is a lot faster than adding new features (fig. 23). This is because the MongoDB large set is drastically larger than the small set, leading to much longer network time. Adding new features, using MongoDB as database, is about 5 times slower than the augmenting of features.

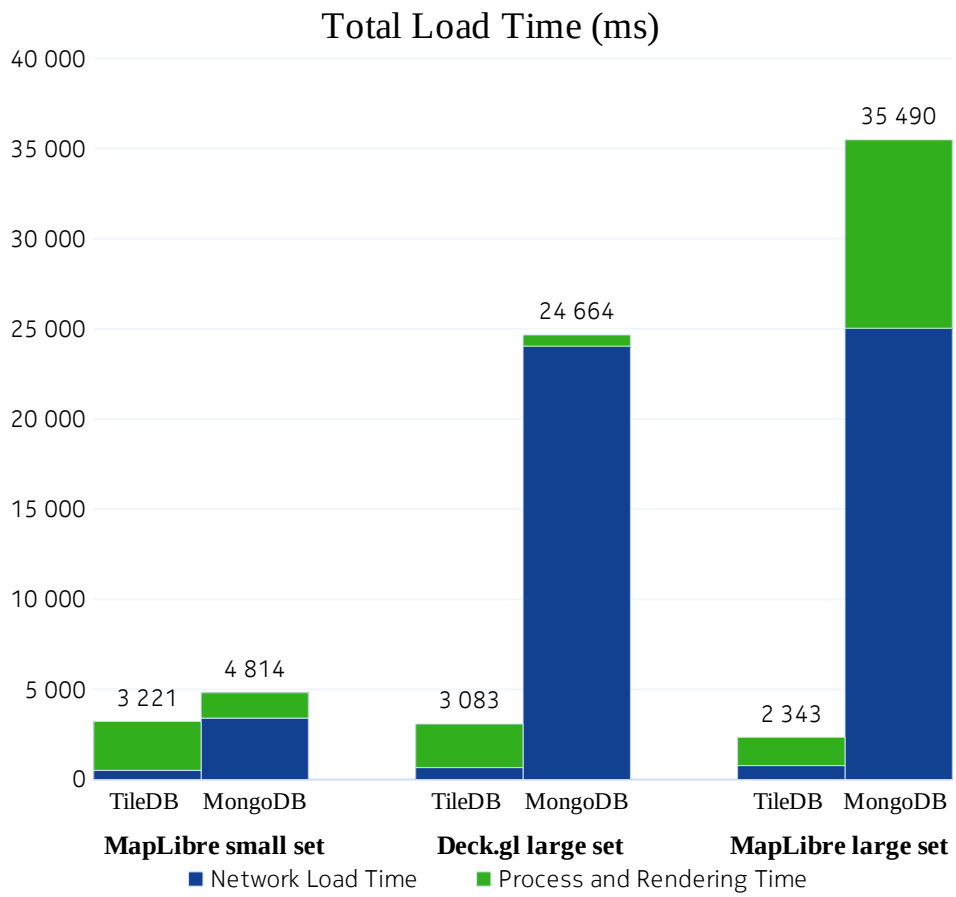

Figure 23. Average total runtime for each scenario

6 CONCLUSION

Both VectorTileDB and MongoDB based solutions were able to successfully be used in dynamically augmenting a map, using both MapLibre and Deck.gl as the rendering li-

braries. MongoDB based solutions on average required less boilerplate code and processing of the data before rendering. But when measuring network transfer time VectorTileDB proved to be significantly faster than MongoDB, resulting in a faster average total runtime.

In this thesis I tested MongoDB based scenarios using an optimized dataset, where the aggregation and filtering of data from MongoDB was not considered. In a real-world scenario the MongoDB would contain a larger dataset of features, and it would be necessary to filter out the features not located inside the viewport. This would skew the results even more in favor of VectorTileDB.

Out of the tested rendering techniques, adding new features to the map is on average faster than augmenting existing features with data. Though many of the techniques are dependent on the use-case. The overall fastest solution, using MapLibre to dynamically add a new VectorTileDB source and style layer, might not always be possible in a real-world use-case, sometimes the goal might for example be to augment already existing features. In the case of augmenting existing features on the map using MapLibre with VectorTileDB small set was the fastest solution.

Loading data from VectorTileDB is done by creating a data tile, containing only the feature data we are interested in, that is added as a source to the map when needed. In a real-world scenario it might be preferable to define the data tile as a source directly into the maps style.json file. This would speed up the querying of data, since once you have the area of the map in the viewport the data is already loaded into memory. Data can be queried without the need to load in a new source and transfer more data over the network. The downside is that the data, from the data tile, would be loaded whenever the viewport changes, which could lead to the map being a bit slower overall.

6.1 Further Development and Research

The results of this thesis are interesting and further research into VectorTileDB should be considered. The possibility of having a fast location-based database can be relevant for many applications and use-cases. Some areas that still need to be researched are: the best ways of modifying data tiles on the backend, what is the performance of modifying data on the backend, how well does the backend scale and testing use-cases of a headless mode of VectorTileDB without the need of a map client. I also see a need to test VectorTileDB in different scenarios to find possible issues that might yet not have been identified.

If VectorTileDB were to be used on a larger scale, there is a need for a client-side framework or library to simplify the VectorTileDB functionality. This would get rid of a lot of boilerplate code that is now required, and it would make it easier to reason about the code. There are also features that could be implemented to TileServer GL to make it better suited for the VectorTileDB use-case: automatic updating and loading of MBTiles and preventing features with geometries on multiple tiles getting split into several features when querying.

REFERENCES

Butler, H., Daly , M., Doyle, A., Gillies, S., Hagen, S., & Schaub, T. (2016). *The GeoJSON Format*. Available from: https://datatracker.ietf.org/doc/html/rfc7946 Accessed 8.2.2022

Deck.gl. (2022). *Writing Your Own Layer*. Available from: https://deck.gl/docs/developer-guide/custom-layers Accessed 9.2.2022

Dora, V. d. (2012). *Transatlanica | A World of "Slippy Maps": Google Earth, Global Visions, and Topographies of Memory*. Available from: https://journals.openedition.org/transatlantica/6156 Accessed 18.1.2022

Fischer, E., Kaefer, K., MacWright, T., Miller, J., Norman, P., Thompson, B., & White, W. (2018). *MBTiles specification*. Available from: https://github.com/mapbox/mbtiles-spec Accessed 23.3.2022

Gore, A. (1998). The Digital Earth: Understanding Our Planet in the 21st Century, *Photogrammetry and Remote Sensing,* 65(5):528 Available from: https://www.asprs.org/wp-content/uploads/pers/99journal/may/1999_may_highlight.pdf Accessed 15.5.2022

Google. (2022). *Protocol Buffers*. Available from: https://developers.google.com/protocol-buffers/ Accessed 8.2.2022

Hiisilä, P. (2022). *Motivation for VectorTileDB*. [telephone conversation], 3.23.2022

Klokan Technologies GmbH. (2016a). *Configuration File*. Available from: https://tileserver.readthedocs.io/en/latest/config.html Accessed 1.4.2022

Klokan Technologies GmbH. (2022b). *Available endpoints*. Available from: https://tileserver.readthedocs.io/en/latest/endpoints.html Accessed 1.4.2022

Mapbox. (2022a). *Improve the performance of Mapbox GL JS maps*. Available from: https://docs.mapbox.com/help/troubleshooting/mapbox-gl-js-performance/#use-feature-state Accessed 14.2.2022

Mapbox. (2022b). *Layers*. Available from: https://docs.mapbox.com/mapbox-gl-js/style-spec/layers/ Accessed 8.2.2022

Mapbox. (2022c). *Sources*. Available from: https://docs.mapbox.com/mapbox-gl-js/style-spec/sources/ Accessed 31.3.2022

Mapbox. (2022d). *Tippecanoe Github*. Available from: https://github.com/mapbox/tippecanoe Accessed 4.3.2022

Mapbox. (2022e). *Vector tiles standards*. Available from: https://docs.mapbox.com/data/tilesets/guides/vector-tiles-standards/ Accessed 8.2.2022

MapLibre. (2022a). *API Reference Map*. Available from: https://maplibre.org/maplibre-gl-js-docs/api/map/ Accessed 24.3.2022

MapLibre. (2022b). *Expressions*. Available from: https://maplibre.org/maplibre-gl-js-docs/style-spec/expressions/ Accessed 1.4.2022

MapLibre. (2022c). *Github MapLibre GL JS*. Available from: https://github.com/maplibre/maplibre-gl-js Accessed 1.4.2022

MapLibre. (2022d). *Projects*. Available from: https://maplibre.org/projects/#js Accessed 1.4.2022

Maputnik. (2022). *Maputnik*. Available from: https://maputnik.github.io/ Accessed 31.3.2022

Minghini, M., & Frassinelli, F. (2019). OpenStreetMap history for intrinsic quality assessment: Is OSM up-to-date?. *Open Geospatial Data, Software and Standards, 4*(1), 1-17.

Mooney, P., & Minghini, M. (2017). A Review of OpenStreetMap Data. In G. Foody, L. See, S. Fritz, P. Mooney, A.-M. Olteanu-Raimond, C. C. Fonte, & V. Antoniou (Eds.), *Mapping and the Citizen Sensor* (pp. 37-59). London: Ubiquity Press Ltd.

Netek, R., Masopust, J., Pavlicek, F., & Pechanec, V. (2020). Performance Testing on Vector vs. Raster Map Tiles—Comparative Study on Load Metrics. *ISPRS International Journal of Geo-Information., 2*(101), 9.

Peterson, M. P. (2012). Online Mapping with APIs. In M. P. Peterson (Ed.), *Online Maps with APIs and WebServices* (pp. 3-12). Berlin: Springer, Berlin, Heidelberg.

Putz, S. (1994). *Interactive Information Services Using World-Wide Web Hypertext*. Available from: https://web.archive.org/web/20110628194820/http://www2.parc.com/istl/projects/www94/mapviewer.html Accessed 18.1.2022

Sample, J. T., & Elias, I. (2010). *Tile-Based Geospatial Information Systems: Principles and Practices.* Berlin: Springer.

Stefanakis, E. (2015). Map Tiles and Cached Maps Services. *Magazine of GoGeomatics Canada, December*.

Uber. (2022). *Vis.gl*. Available from: https://vis.gl/ Accessed 1.4.2022

Yang, W. (2019). *Deck. gl: Large-scale web-based visual analytics made easy*. Available from: https://arxiv.org/pdf/1910.08865 Accessed 9.2.2022

APPENDICES

APPENDIX 1 SWEDISH SUMMARY

Introduktion

Uppdragsgivare för detta arbete är Nokias *Advanced Technology Group* (ATG) som är en avdelning inom Nokia som arbetar med att evaluerar nya teknologier.

Idén till arbetet föddes när jag arbetade på ett projekt vid ATG som hette *Twin-City*, där målet var att skapa en digital tvilling av en stad med hjälp av vektorkartor. All data om det området som visas på kartan laddas in i RAM, vilket leder till att det går snabbt att få tillgång till denna data. Ett objekt i en vektorkarta kan också innehålla hur många (nyckel: värde) par som helst. Detta ledde till idén att använda vektorkartor som en databas för objekt som har geografiska egenskaper. Jag byggd en prototyp av denna idé, vilket ledde till detta examensarbete.

Målet med detta arbete är att testa hur väl vektorkartor lämpar sig för att användas som databas, detta koncept kommer jag att kalla *VectorTileDB*. Arbetet kommer att jämföra hur väl VectorTileDB klarar sig mot en industristandard NoSQL databas, nämligen *MongoDB*.

Bakgrund

De tidigaste online kartorna var väldigt simpla, den kartvy klienten var intresserad av genererades på servern och skickas till användaren vid behov. Google introducerade år 2005 Google Maps och Google Earth, vilka tillät användaren att panorera och zooma kartan utan restriktioner. Den här nya typen av kartor kallades för *slippy maps*. (Dora, 2012, s. 1-2,4)

Google skapade också *Web Mercator koordinatsystemet*, där hela världen kan visas som en fyrkant eller en *tile*. Denna tile kallas för noll zoom-nivån. När användaren zoomar in en zoom-nivå så delas alla tiles från den föregående nivån in i fyra nya tiles. Varje *tile* på en zoom-nivå har unika (X, Y) koordinater, där den *tile* som är upp i vänstra hörnet av kartan har värdena (0, 0) (Sample & Elias, 2010, s. 15). Servern skickar

endast de *tiles* som krävs för den nuvarande kartvyn, och klienten placerar alla *tiles* på rätt plats bredvid varandra för att forma kartan som visas för användaren.

Sedan Googles *slippy maps* lanserades så har *raster tile* kartor varit den mest använda typen av online kartor. Rasterkartor är uppbyggda av en matris av pixelvärden, där en *tile* oftast innehåller 256x256 pixlar, men även 64x64 eller 512x512 används. Alla *tiles* måste genereras på förhand och lagras på servern, om en ändring görs till kartan så måste alla *tiles* genereras på nytt.

Flera online kartor har börjat använda sig av vektorkartor. De använder sig av samma *tile* system som tidigare rasterkartor, men i stället för att kartan är representerad i form av pixlar så används vektorobjekt. Ett objekt består av något geometriskt objekt: antingen en *punkt, linje* eller *polygon,* objektet kan också innehålla extra metadata. Servern skickar endast en lista på vektorobjekt till klienten, och klienten bestämmer hur objekten skall visas på kartan. Vektorkartor är således mera beroende på klientens hårdvara, till skillnad från rasterkartor som genereras på servern. Objekt på en vektorkarta kan också visas i 3D vilket leder till mera interaktiva kartor. Det är också möjligt för klienten att ändra på hur kartan ser ut dynamiskt, något som inte är möjligt med rasterkartor.

Ett av de vanligaste formaten på vektorkartor är *Mapbox Vector Tiles,* som använder sig av Googles *protobuf* teknik för att omkoda data till ett mera kompakt binär format. För att lagra flera vektor *tiles* i en fil används MBTiles formatet, som också är gjort av *Mapbox*.

Design Och Implementation

Testramverket för detta arbete baserar sig till en del på Twin-City projektet. Klienten är byggd med React och skriven i Typescript. Backend APIn för MongoDB är byggt med FastAPI och skriven i Python. För att generera vektorkartor används tile_generator, vilket är ett verktyg jag byggde åt ATG till Twin-City projektet. Tile_generator tar emot *OpenStreetMap* (OSM) data och genererar en vektor *tile* i MBTiles formatet. Den konverterar först OSM data till GeoJSON med verktyget *ogr2ogr*. GeoJSON filerna kan sedan fixas och konverteras till MBTiles med ett verktyg som heter Tippecanoe, också byggt av Mapbox. MBTiles filerna serveras sedan med TileServer GL servern.

Konceptet VectorTileDB kan definieras som en *tile* server, som fungerar som en databas. Den serverar en eller flera *data tiles,* som kan begäras via en klients kartbibliotek, eller via en HTTP- förfrågning. Kartbiblioteket MapLibre har två inbyggda funktioner för att söka efter data från en vektor *tile*, *queryRenderedFeatures* och *querySourceFeatures.* Skillnaden mellan funktionerna är att queryRenderedFeatures endast returnerar sådana objekt som är synliga på kartan, medan querySourceFeatures returnerar även sådana objekt som finns i en vektor *tile* men inte är synliga på kartan.

Prestandatest

För testandet av databaserna och renderingsbiblioteken används ett område av Huvudstadsregionen i Finland som innehåller ca. 85,000 byggnader. Data, om hur byggnaderna skall renderas, hämtas från databasen och visas sedan på kartan. Dataseten är uppdelat i två olika set, *small set* och l*arge set*. *Small set* används för att ändra färg på byggnader som redan visas på kartan. Ett objekt i *small set* innehåller endast en *id*, för att koppla ihop rätt data med rätt objekt, och ett färgvärde. *Large set* innehåller fullständiga vektorobjekt för byggnader och används för att rendera nya byggnader på kartan.

De olika scenarion som testas är följande:

Small set:

- VectorTileDB med MapLibre
- MongoDB med MapLibre

Large set:

- VectorTileDB med MapLibre
- VectorTileDB med Deck.gl
- MongoDB med MapLibre
- MongoDB med Deck.gl

Resultat

Den mängd data som skickas från servern visade sig vara mycket större med MongoDB jämfört med VectorTileDB. Med *small set* var MongoDB 8,6 gånger större och i *large*

set 14,9 gånger större än VectorTileDB. Detta på grund av att VectorTileDB använder sig av Googles protobuf. Denna stora skillnad i mängden data som överförs leder också till att VectorTileDB är mycket snabbare än MongoDB. VectorTileDB var ca. 1,5 gånger snabbare än MongoDB på *small set* scenarion och ca. 10 gånger snabbare på *large set* scenarion.

Scenarion som använder MongoDB som databas visade sig vara snabbare på bearbetning och rendering av data. Men ingen större skillnad fanns mellan de två renderingsbiblioteken MapLibre och Deck.gl. Dock visade det sig att det är snabbare att rendera nya objekt på kartan en att ändra på existerande objekt.

När man ser på helheten, hur länge det tog att hämta data från servern och visa resultatet på kartan, så var VectorTileDB baserade scenarion snabbare i varje test.

Slutsats

I detta arbete visade det sig att både VectorTileDB och MongoDB kan användas som databaslösningar, för att visualisera data på en vektorkarta. VectorTileDB visade sig dock vara mycket mera effektivare än MongoDB, detta på grund av storleken av filerna som skickas från servern. Både MapLibre och Deck.gl visade sig vara användbara för att rendera objekt på kartan.

Resultatet från arbetet är intressant och visar att det vore lönsamt att forska mera i VectorTileDB. Även om det är en väldigt snabb databaslösning, som kan vara användbara i flera olika användningsfall, så finns det dock flera frågor att forska i när det gäller VectorTileDB. Bland annat hur kan man uppdatera data *tiles* effektivt och hur väl skalas lösningen när man uppdaterar stora mängder element.