OST
Eastern Switzerland
University of Applied Sciences

Cloud-Optimized OSM GeoParquet Data Service for Switzerland and Beyond

Department of Computer Science
OST - University of Applied Sciences
Campus Rapperswil-Jona

Bachelor Thesis
Spring Term 2025

Author(s)	Fadil Smajilbasic, Nils-Robin Grob, Matthias Hersche
Advisor	Stefan Keller
Project Partner	Adrian Bona
External Co-Examiner	Claude Eisenhut
Internal Co-Examiner	Urs Baumann

Revision

Creation Date	2025-02-18
Last Revision	2025-06-13

Imprint

Fadil Smajilbasic
fadil.smajilbasic@ost.ch

Nils-Robin Grob
nilsrobin.grob@ost.ch

Matthias Hersche
matthias.hersche@ost.ch

OST
Eastern Switzerland
University of Applied Sciences

Abstract

OpenStreetMap (OSM) is one of the most comprehensive openly licensed geospatial vector datasets, containing an estimated 60–90 million points of interest (POIs). While this is comparable to Overture Maps ~61 million POIs, OSM distinguishes itself through its data richness, openness, and crowdsourced quality assurance. However, its raw structure based on a graph of nodes, ways, and relations combined with a flexible tagging system, presents significant challenges for scalable querying and analysis.

This thesis presents a reproducible, open-source pipeline designed to transform country-scale OSM extracts, such as those from Geofabrik, into simplified, analysis-ready GeoParquet files. The files are aligned with Overture Maps Places and Divisions themes and converted into a tabular format optimized for geographic information systems. The solution is built on a modular Extract–Transform–Load (ETL) architecture using osm2pgsql with Lua scripts, PostgreSQL/PostGIS for schema alignment and spatial processing, and DuckDB with PyArrow for high-performance GeoParquet conversion.

Multiple spatial file partitioning strategies, including KDB Tree and S2, were evaluated to support efficient downstream interoperability and client-side filtering. The pipeline operates as a CI/CD enabled DataOps workflow, orchestrated via GitLab, containerized with Docker, and hosted on S3-compatible MinIO storage. A vandalism detection module prototype supports data quality by flagging anomalies in stable administrative names.

The result is Cadence Maps, a fully automated and publicly accessible data service for the D-A-CH-LI region, updated weekly and accompanied by release documentation. GeoParquet files can be queried directly via DuckDB or QGIS without requiring full downloads. For example, queries can filter specific features such as restaurants using hive-compatible S3 prefixes. Full dataset updates can be completed in under 24 hours, demonstrating the system's performance and scalability. This work establishes a reliable and extensible framework for delivering cloud-native geospatial data services with global potential.

Management Summary

Introduction

OpenStreetMap (OSM) represents one of the most comprehensive openly licensed geospatial vector datasets, maintained and enriched by a global community of contributors. With an estimated 60–90 million points of interest (POIs), it rivals proprietary solutions such as Overture Maps, which advertises approximately 61 million POIs. The distinguishing features of OSM include its openness, data richness, and the crowdsourced validation that ensures continual improvements and relevance across domains including urban planning, logistics, environmental monitoring, and emergency response.

However, this openness and flexibility also introduce challenges. OSM's core data model is based on a graph structure, comprised of nodes, ways, and relations, and uses a highly flexible tagging system. While this allows the community to map a vast diversity of features, it makes the data inherently unstructured and inconsistent, potentially presenting significant obstacles for users who require standardized, analysis-ready formats.

In contrast, modern cloud-native file formats like GeoParquet offer significant advantages for analytical workflows. These formats enable scalable, SQL-based querying without the need for dedicated server or database infrastructure. Yet, leveraging these benefits requires extensive preprocessing and transformation of raw OSM data—a process that has traditionally relied on custom, ad-hoc solutions. While initiatives like Overture Maps aim to provide unified schemas, they fall short in terms of openness and transparency, relying on proprietary software and data curation processes that are not publicly verifiable.

Problem Statement

The central problem addressed by this project is the lack of a reproducible, scalable, and open-source pipeline that transforms raw OpenStreetMap extracts into fully analysis-ready, partitioned GeoParquet datasets, accessible directly from cloud storage and usable without dedicated backend infrastructure.

The utilization of OpenStreetMap data at scale presents multiple obstacles for analysts, developers, and researchers:

- Manual preprocessing is time-consuming and prone to inconsistencies.
- Raw data lacks a uniform schema and cannot be directly queried using standard analytical tools.
- Most available datasets require full downloads, leading to substantial storage and bandwidth requirements.
- Existing solutions do not guarantee traceability or data quality checks, particularly with respect to vandalism or inconsistencies between updates.

This project seeks to bridge these gaps by developing a solution that not only transforms data into an optimized analytical format but also ensures data integrity, scalability, and transparency.

Approach / Technology

The solution implements a modular Extract–Transform–Load (ETL) pipeline built entirely on open-source technologies and optimized for country-scale datasets. The pipeline starts with country-specific extracts, primarily from Geofabrik, and performs a series of transformations to produce high-quality, analysis-ready GeoParquet files.

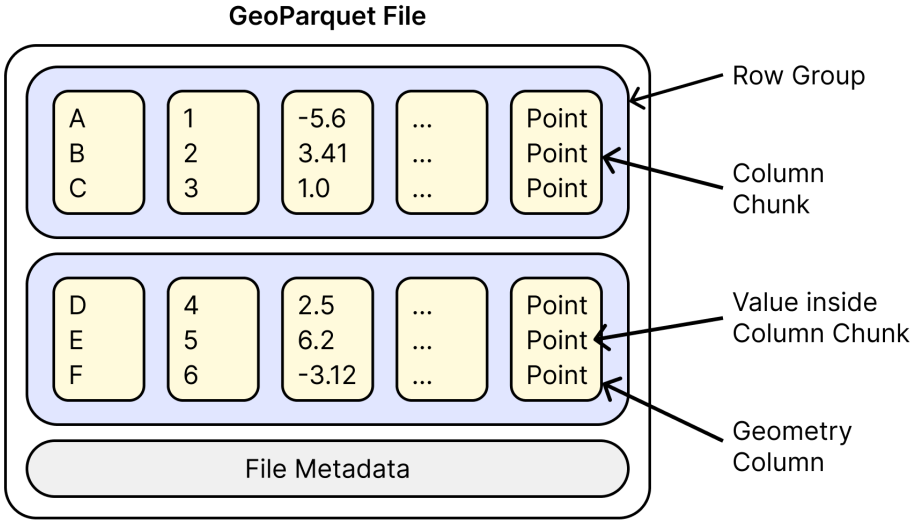


Figure 1: Anatomy of a GeoParquet File

Key components and technologies include:

- osm2pgsql with custom Lua scripts to ingest and map OSM features into structured layers.
- PostgreSQL/PostGIS to handle spatial operations and enforce consistent data schemas aligned with the Places and Divisions themes from Overture Maps.
- DuckDB and PyArrow to convert the cleaned data into partitioned GeoParquet files.
- Spatial file partitioning using KDB Tree to ensure optimal performance for large-area queries.
- GitLab CI/CD for orchestration and automation, ensuring regular updates and reproducible builds.
- Docker for containerization of the entire system, enabling consistent deployment across environments.
- MinIO, an S3-compatible object storage service, to host the output data publicly and enable remote SQL access.

To ensure data integrity and mitigate the risk of vandalism, the system incorporates a lightweight, rule-based validation mechanism prototype. It monitors key indicators, such as administrative name changes, that would otherwise go unnoticed and flags suspicious updates. If anomalies are detected, datasets are staged rather than published, requiring manual approval through GitLab pipelines.

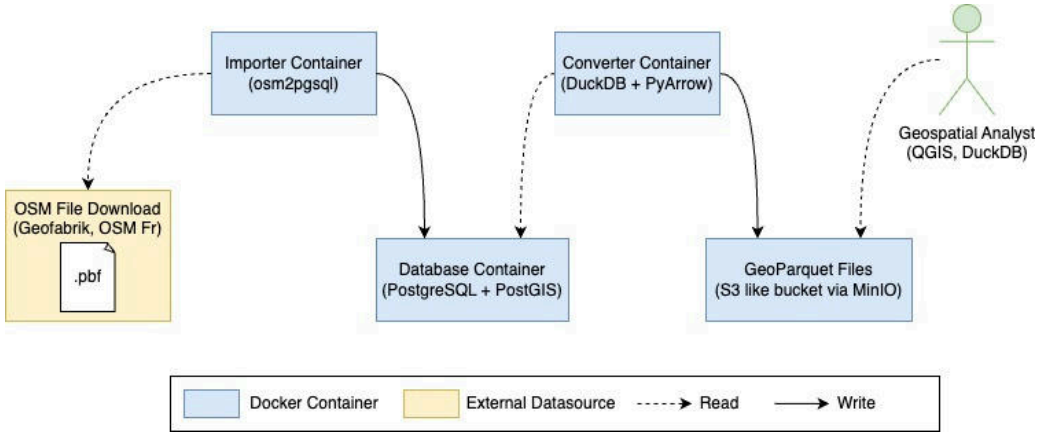


Figure 2: ETL pipeline diagram illustrating the OpenStreetMap import, conversion to GeoParquet files and access by geospatial analysts

Result

The resulting product, Cadence Maps, is a fully automated, cloud-hosted data service providing OpenStreetMap derived GeoParquet datasets for Switzerland, Germany, Austria, and Liechtenstein. The system processes and publishes weekly updates through a dedicated website and S3-compatible storage endpoint.

The generated files are structured to support hive-style partitioning, enabling users to execute queries directly against remote datasets without downloading the full files. For example:

```
SELECT names.primary
FROM read_parquet(
  's3://cadencemaps/release/2025-05-13/theme=places/type=place/country=CH/*',
  filename=true, hive_partitioning=1)
WHERE categories.primary = 'restaurant';
```

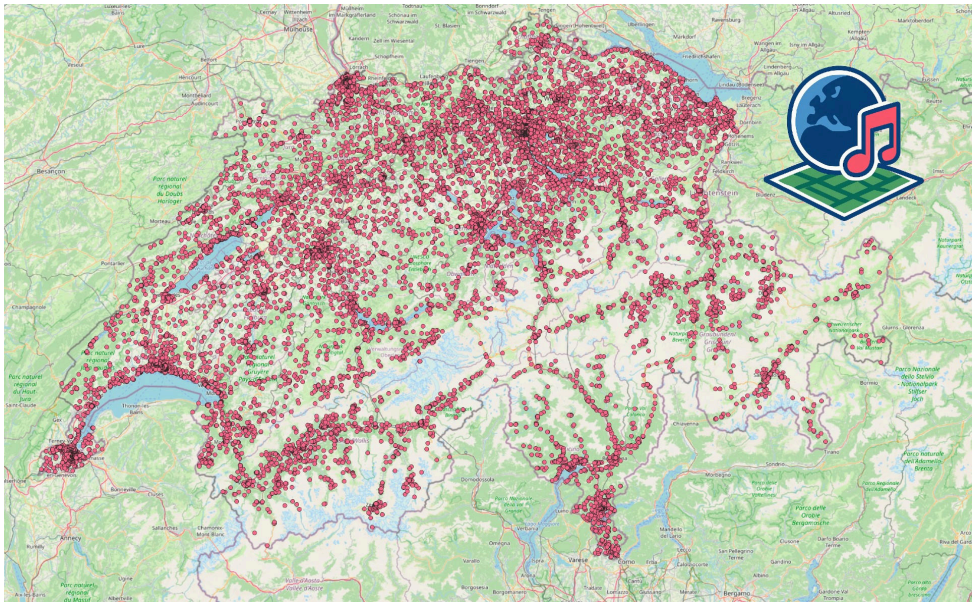


Figure 3: Map of Switzerland showing all restaurants as red dots accessed via our Cadence Maps

This structure facilitates both efficient client-side filtering and seamless integration with tools such as DuckDB, QGIS, and Python-based tools, making the data accessible to a broad range of users.

Performance evaluations demonstrate that complete updates for the entire D-A-CH-LI (Germany, Austria, Switzerland, and Liechtenstein) region are processed in under 24 hours. This makes the solution suitable for near real-time applications and extensible for broader geographical coverage.

Outlook

The current implementation provides a robust and extensible foundation for scalable, reliable, and open access to structured geospatial data derived from OpenStreetMap. Several forward-looking enhancements are envisioned to increase its reach, efficiency, and usability.

A key area of expansion lies in scaling beyond the D-A-CH-LI region. The pipeline can be extended to process additional countries, broadening its relevance across Europe and globally. This would allow international users to benefit from consistent, partitioned GeoParquet data that is immediately usable in tools such as DuckDB or QGIS, without requiring local infrastructure.

To strengthen trust in the data, future iterations could enhance validation logic with checks for anomalies in geometry, naming, or data density. Integration of rule-based systems, community tools like Clearance, or even machine learning would improve the detection of vandalism and ensure higher data integrity, particularly important as public usage grows.

Operationally, the pipeline's deployment could benefit from decoupling key components like PostgreSQL and the proxy server from container environments in favor of native hosting. This would improve runtime performance and ease debugging. Additionally, introducing real-time monitoring solutions such as Prometheus or Sentry would support proactive maintenance, alerting, and long-term stability.

Finally, for larger-scale or enterprise-grade use cases, integrating Apache Iceberg can offer advanced features like ACID-compliant versioning, time travel, and efficient data pruning. This would position the system as a long-term, cloud-native platform capable of managing both current and historical geospatial data at scale.

In summary, the pipeline is not just a functional solution for today's analytical needs—it also serves as a strategic base for future-ready, globally accessible geospatial data services.

Contents

Glossary and Acronyms

BBOX	Bounding Box; a rectangular area used for spatial filtering in queries.
C4 Model	A model for visualizing and documenting software architecture, focusing on the context, containers, components, and code.
CI/CD	Continuous Integration / Continuous Deployment; automates testing and deployment.
Confluence	A collaboration tool used for documentation and project management.
Converter	The part of the pipeline that includes the process of partitioning and sorting the Postgres / PostGIS dataset into Parquet files.
D-A-CH-LI	An abbreviation referring to the countries Germany (D), Austria (A), Switzerland (CH), and Liechtenstein (LI).
DataOps	DataOps is an agile, process-oriented methodology focused on automating and monitoring data pipelines to deliver high-quality data quickly, efficiently, and reliably for business analytics and AI.
DevOps	DevOps is a modern approach in the IT world that combines software development (Dev) and IT operations (Ops) to shorten the system development lifecycle and provide continuous delivery of high-quality software.
Docker	A platform designed to help developers build, share, and run modern applications using containerization technology.
DuckDB	An in-process OLAP database optimized for analytical queries, especially with Parquet files.
ETL	Extract, Transform, Load; a data pipeline model for processing and storing data.
Flex Mode	A flexible mode in osm2pgsql that uses Lua scripting for schema customization.
GDAL	The Geospatial Data Abstraction Library; a library for reading and writing geospatial data in various formats.
GeoArrow	A binary format under development to efficiently encode geometries within the Arrow ecosystem.
GeoHash	A geocoding system that encodes geographic coordinates into short strings of letters and digits.
GeoJSON	A JSON-based format for encoding geographical data structures.
GeoParquet	A geospatial extension of the Apache Parquet format, designed for efficient storage and querying of geospatial data.
Geospatial Data	Data that identifies the geographic location and characteristics of natural or constructed features and boundaries on the Earth.
Git	A version control system for tracking changes in computer files and coordinating work among multiple developers.

GitLab	A web-based DevOps lifecycle tool that provides a Git-repository manager providing wiki, issue-tracking, and CI/CD pipeline features.
H3	A geospatial indexing system developed by Uber, designed for efficient spatial queries and analysis.
Hilbert Curve	A space-filling curve used for sorting and partitioning spatial data while preserving locality.
Importer	The part of the pipeline that includes the process of downloading PBF Files and converting them into a Postgres / PostGIS compatible dataset
JIRA	A project management tool used for issue tracking, bug tracking, and agile project management.
KDB Tree	A data structure for spatial partitioning that splits space recursively based on data density.
MinIO	An object storage solution with S3-compatible API, used for storing output datasets.
OSM (OpenStreetMap)	A collaborative, open-source mapping project providing global geospatial data.
Overture Maps	An open mapping initiative that combines authoritative and open data, including from OSM.
Overpass API	A read-only API for querying specific data from the OSM dataset.
PBF	Protocolbuffer Binary Format; compact binary format for storing OSM data.
Partitioning	The process of dividing a dataset into smaller, more manageable parts, often used to improve query performance and data management.
PostGIS	A spatial extension for PostgreSQL that enables geospatial queries and operations.
PostgreSQL	An open-source relational database management system.
PyArrow	A Python interface to Apache Arrow, used to write GeoParquet files.
QGIS	A free and open-source GIS application used for viewing and analyzing spatial data.
QuadTree	A tree data structure in which each internal node has exactly four children, used for spatial indexing and partitioning.
S3	A scalable object storage service provided by Amazon Web Services (AWS), commonly used for storing and retrieving large amounts of data.
S2	Hierarchical global spatial indexing systems for partitioning and analyzing spatial data. S2 was developed by Google.
SCRUM+	A hybrid project management methodology combining SCRUM and Rational Unified Process (RUP).

Shapefile	A legacy vector data format used in GIS software, developed by Esri.
Spatial Indexing	A method to improve the performance of spatial queries by indexing geometries.
STAC	SpatioTemporal Asset Catalog; an open specification for geospatial data that provides a common language for describing geospatial assets.
Validation	The process of checking and ensuring the accuracy, consistency, and quality of data.
Vandalism	The deliberate addition of incorrect or misleading data, often a concern in collaborative and open data projects like OSM.
osm2pgsql	A tool for importing OSM data into PostgreSQL/PostGIS databases.

Part I - Product Documentation

1. Introduction

1.1. Problem Statement and Vision

Geospatial data forms the backbone of modern decision making in areas such as urban planning, transportation, environmental monitoring, and logistics. OpenStreetMap (OSM), a community-driven and openly licensed dataset, offers a rich source of global geospatial information. However, its raw format, designed for flexibility rather than efficiency, poses significant challenges for scalable analysis. The data is inconsistently tagged, lacks a standardized tabular structure, and requires complex preprocessing to be usable in analytical workflows.

Early solutions for processing and querying OSM data typically relied on maintaining a constantly running PostgreSQL/PostGIS database. While powerful, this approach introduces several drawbacks: the database must be continuously available, requires significant maintenance effort, and demands infrastructure that is often over-provisioned for sporadic querying workloads. These setups are difficult to scale, hard to reproduce, and often unsuitable for lightweight, serverless analytics.

Modern workflows, by contrast, benefit from cloud optimized, queryable file formats such as GeoParquet. In this model, the heavy lifting of parsing and transforming the OSM data is done once during preprocessing. The result is a structured, columnar dataset that can be stored in object storage and queried directly by tools like DuckDB or QGIS, without needing to set up or maintain any database infrastructure. This not only reduces operational complexity but also aligns with cloud native principles such as statelessness and scalability.

The vision of this thesis is to bridge that gap: to create a modular, scalable, and cloud optimized pipeline that transforms raw OSM data into structured GeoParquet files. The pipeline is designed to support efficient analysis, reuse, and integration across modern data systems.

1.2. Objectives and Sub-Objectives

The overarching objective of this thesis is to design and develop an automated, reproducible, and scalable data processing pipeline that transforms OSM data into a structured, cloud optimized format suitable for geospatial analysis.

To achieve this, the project is structured around the following sub objectives:

- **Schema Definition**: Define a data model that enables efficient representation of OSM data in a tabular geospatial format. The schema should preserve key semantic elements while enabling compatibility with modern query engines.
- **Pipeline Development**: Develop a modular and maintainable processing pipeline that can ingest, transform, and output geospatial data in a standardized and analysis ready form.
- **Validation and Integrity Checks**: Introduce mechanisms for ensuring data quality and integrity, with a focus on detecting structural inconsistencies and preserving trust in the dataset over time.
- **Performance Optimization**: Optimize the processing pipeline to support full country scale datasets and enable efficient spatial querying through suitable partitioning and system design.

1.3. Constraints, Scope, and Definitions

The scope of this thesis is limited to the design and implementation of a country scale data processing pipeline for converting OSM data into GeoParquet format. While the architecture

is designed to be globally scalable, the evaluation focuses on the D-A-CH-LI region, with Switzerland as the primary reference dataset.

Key constraints include:
- All tools used must be open-source.
- The pipeline should be executable on institutional infrastructure.
- Processing time for an update cycle should not exceed 24 hours.

1.4. Methodology and Research Approach

This thesis combines applied research with practical software engineering to build a state-of-the-art geospatial data pipeline. The development process was guided by an iterative approach: design decisions were continuously evaluated through prototyping, performance testing, and comparison against current standards and community tools.

Given the novelty of applying cloud native technologies like GeoParquet and DuckDB to raw OSM data, significant effort was dedicated to evaluating data schemas, partitioning strategies, and validation logic. This involved not only selecting existing tools, but in many cases extending or adapting them for the specific requirements of scalable geospatial processing.

The result is a bleeding-edge system that bridges gaps between open-source mapping data and modern analytical workflows. Rather than remaining theoretical, this project delivers a tangible, fully functional implementation that can be adopted, extended, and deployed by data engineers.

2. State of the Art

This chapter outlines open geospatial datasets, standards, processing tools, and data access methods used in modern spatial data pipelines. Emphasis is placed on scalable, cloud-native technologies aligned with the goals of this thesis.

2.1. Map Data

This section covers open map data sources. Mainly OpenStreetMap and Overture Maps, used for analysis and integration. Commercial platforms like Google Maps or Swisstopo are excluded due to restrictive licensing.

2.1.1. OpenStreetMap

OpenStreetMap is a global, collaborative project that provides openly licensed geospatial data. Founded in 2004, OSM has become a foundational dataset for navigation, urban analysis, and spatial applications worldwide. Unlike proprietary platforms, OSM allows anyone to contribute, edit, and access data freely [1].

The platform operates under the Open Database License (ODbL), ensuring that its data can be used, modified, and redistributed, provided that derivative datasets remain equally open.

Core Data Model

OSM's data structure is organized around three key primitives:

- **Nodes**: Point features with latitude and longitude. Used to represent standalone entities like trees or traffic signs.
- **Ways**: Ordered lists of nodes. These define linear features (e.g., roads, rivers) or area boundaries (e.g., buildings, lakes).
- **Relations**: Groupings of nodes and ways that form more complex features such as multi-polygons, bus routes, or administrative boundaries.

Each element uses a flexible key-value tagging system, enabling contributors to encode rich metadata. While this flexibility supports global diversity, it can introduce inconsistency across regions and contributors.

Data Formats and Accessibility

OSM data is available in multiple formats:

- .osm (XML) and .pbf (Protocolbuffer Binary Format) for raw dumps
- .shp, .geojson, and increasingly GeoParquet for analysis and cloud workflows

The Protocolbuffer Binary Format (PBF) is especially efficient for bulk import and processing in tools like osm2pgsql (see Section 2.2.1).

Data Quality and Research

Numerous studies have evaluated OSM's data quality. Findings show that urban areas benefit from high positional accuracy and feature completeness, while rural regions may be underrepresented [2].

Common quality metrics include:

- Positional accuracy: Compared against authoritative data sources.
- Completeness: Coverage of roads, POIs, and buildings.
- Consistency: Regional variation in tag usage and contributor activity.

Another metric is global road coverage, estimated to be above 80% [3], demonstrating the project's extensive coverage and reliability as a geospatial data source.

Efforts to improve OSM data include community validations, spatial integrity checks, and machine-learning-based error detection [4].

2.1.2. Overture Maps

Overture Maps is an open mapping initiative founded by Amazon Web Services, Meta, Microsoft, and TomTom, under the Linux Foundation [5]. Its mission is to build high-quality, interoperable map data by aggregating multiple open and authoritative sources, including OpenStreetMap data [6].

The project publishes datasets as partitioned GeoParquet files, hosted on S3 storage and structured for analytical queries in modern data tools such as DuckDB. Overture adopts a well-documented, evolving schema tailored to core map themes such as places, buildings, transportation, and administrative boundaries.

Data Licensing and Openness

Overture Maps embraces open data principles, primarily using CDLA-Permissive v2 for its datasets, with OSM-derived data under ODbL v1.0. Computational results using this data are exempt from license text requirements, enabling flexible commercial and non-commercial use [7].

Unlike OpenStreetMap, Overture is designed as a data-centric project rather than a community of individual map editors. The schema and release process are primarily driven by member organizations, with no direct editing mechanism for the general public. While individuals can contribute through GitHub or by providing feedback, the primary contribution model involves organizational membership [7].

Monthly Release Cadence

Overture Maps follows a monthly release schedule. As of this writing, the 2025-02-19.0 release introduced over 40,000 km of new roads, improved building footprints, updated division hierarchies, and a 12% expansion of the places dataset [8].

Each Overture data release includes:

- Partitioned GeoParquet files (ZSTD-compressed)
- Theme- and type-specific S3 key prefixes
- Hive-compatible folder structure for querying

Example:

```
s3://overturemaps-us-west-2/release/2025-02-19.0/theme=places/type=place/*
```

Use Cases and Strengths

- Cloud-native workflows: The file format and structure make it ideal for querying in cloud data warehouses and geospatial engines.
- Commercial adoption: The schema is stable and predictable, enabling integration into routing, search, and delivery platforms.
- Data unification: Combines authoritative sources (e.g., U.S. TIGER) with community-curated datasets like OSM, improving global coverage.

Limitations

Despite its structured approach and cloud-native format, Overture Maps has several noteworthy limitations:

- Limited community governance: Unlike OpenStreetMap, Overture is not a community-editable platform. Users cannot directly contribute or correct data; instead, updates rely on aggregated sources curated by corporate contributors.
- Opaque data sources: Portions of the dataset are derived from proprietary tools such as Meta's Map with AI, which are not publicly auditable or user-editable, reducing transparency and trust in the data.
- Questionable data accuracy: In some cases, the dataset includes large numbers of implausible or misplaced entries. For example, the latest dataset (2025-02-19.0) contains thousands of place entries scattered across the Atlantic Ocean without valid geographic justification (see Figure 4).
- Update delay and regional imbalance: Real-world changes can take weeks to appear, and data coverage outside North America and Europe remains incomplete or outdated.

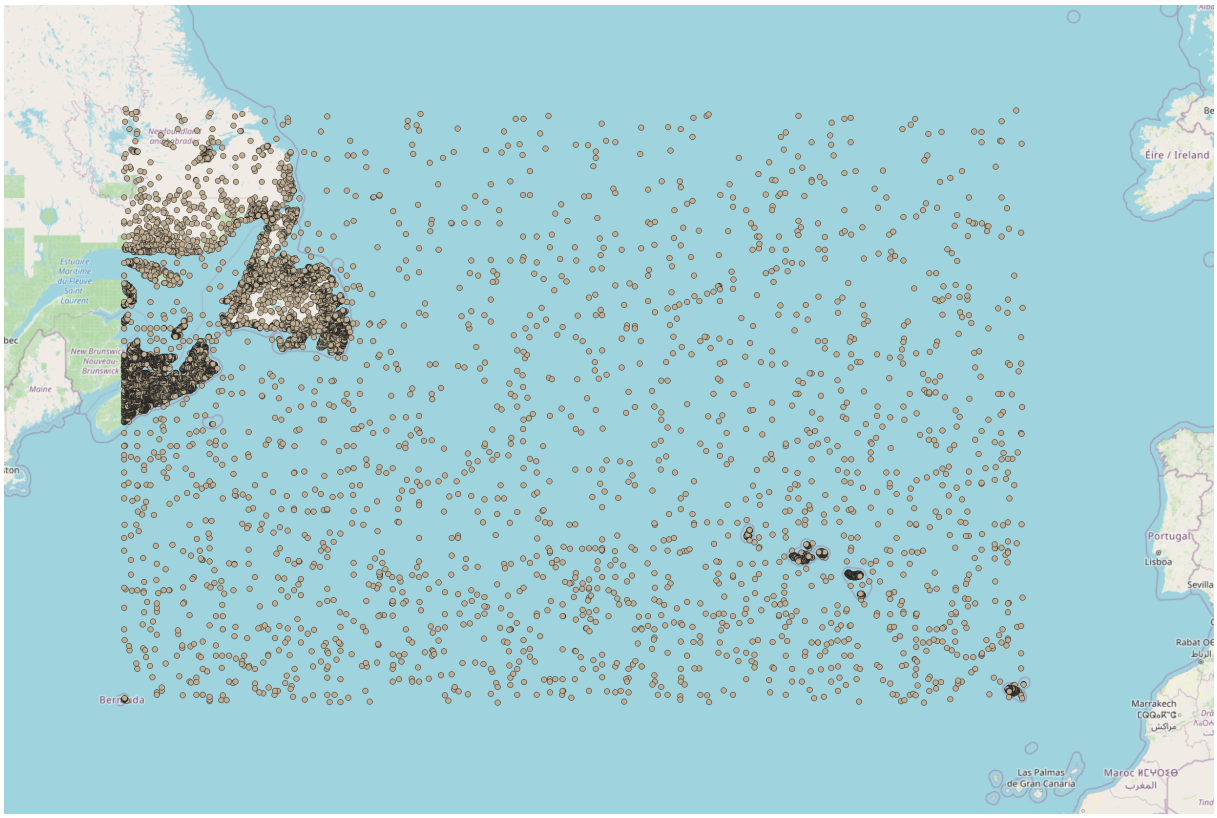


Figure 4: Example of invalid place entries in ocean areas (Overture Maps release 2025-02-19)

2.1.3. Other Map Data Providers

While this project focused exclusively on open and collaborative sources such as OpenStreetMap and Overture Maps, it is important to acknowledge several prominent commercial and national mapping platforms that represent the current state of the art in geospatial data:

- **Google Maps** and **Apple Maps** provide comprehensive, regularly updated global datasets. However, their proprietary licensing models impose restrictions on raw data extraction and integration with open-source workflows.
- **Swisstopo**, the Swiss Federal Office of Topography, offers authoritative and high-resolution datasets for Switzerland, predominantly used in governmental and infrastructure contexts.

The data is available under open licenses, primarily Open Government Data (OGD), with some high-resolution products requiring specific usage terms and attribution.
- **OpenGarmin** supplies map data tailored for Garmin devices, often derived from OSM, but with additional preprocessing for navigation use cases.

Due to licensing restrictions and limited raw data accessibility, these sources were not considered in the design or implementation of our data pipeline.

2.2. Data Management Technologies

This section provides an overview of modern technologies commonly used in geospatial data processing. Each tool plays a specialized role in transforming raw spatial data into analysis-ready formats. Together, they represent a state-of-the-art toolchain for scalable, modular, and efficient spatial data workflows.

2.2.1. osm2pgsql

osm2pgsql is a widely adopted tool for importing OpenStreetMap data into spatial databases. It converts .osm.pbf or .osm.xml files into structured tables within a PostgreSQL/PostGIS database. The tool supports two output modes: the traditional PgSQL output with a fixed schema defined by style files, and the modern Flex output with Lua-based configuration that allows for fully customizable database schemas and transformations.

Key Features
- Selective tag processing: Enables filtering of OpenStreetMap's extensive tag set to include only relevant attributes, reducing database size and complexity.
- Schema customization: Lua scripts allow advanced control over which objects are stored and how they are structured.
- High-performance ingestion: Designed for fast imports of large-scale extracts.
- Incremental updates: Supports applying .osc diff files for ongoing synchronization.

osm2pgsql is commonly used as the first step in OSM data pipelines, converting raw data into a relational format ready for spatial querying and further transformation. [9], [10]

2.2.2. PostgreSQL / PostGIS

PostgreSQL is a mature, open-source relational database management system (RDBMS). Extended with PostGIS, it evolves into a powerful spatial database with comprehensive capabilities for the storage, retrieval, and analysis of complex geospatial data structures. [11]

PostGIS adds native geometry types (e.g., POINT, LINESTRING, POLYGON) and over 300 spatial functions for filtering, measuring, and transforming geographic features.

Key Features
- Spatial indexing: R-tree–based GiST indexes for efficient spatial queries.
- Advanced SQL support: Complex joins, aggregates, and filtering logic for spatial and non-spatial data.
- Standards-compliant: Follows OGC and SQL/MM standards for spatial operations.
- Extensive ecosystem: Integrates well with tools like QGIS, GDAL, and Python (e.g., GeoPandas).

PostGIS often serves as a central staging area in geospatial workflows, supporting both heavy transformations and exploratory analysis prior to conversion into optimized formats such as GeoParquet. [12]

2.2.3. DuckDB

DuckDB is an in-process SQL OLAP database optimized for analytical workloads. It is designed for simplicity, speed, and seamless integration with modern data science workflows. Unlike traditional databases, DuckDB operates entirely within the host process (e.g., Python, R, or command-line scripts), eliminating the need for a separate server. [13]

Key Features

- Columnar storage: Efficient for analytical scans and aggregations over large datasets.
- In-process execution: Minimal setup, fast startup, and low overhead.
- SQL compatibility: Supports complex queries, joins, CTEs, and window functions.
- Cloud and file system integration: Reads directly from local files and cloud storage (e.g., S3), including .parquet, .csv, and .json.
- Spatial extension: Adds support for spatial data types and functions, including reading and writing various geospatial formats via GDAL [14].

DuckDB is particularly well-suited for transforming and exporting large spatial datasets into columnar formats like GeoParquet, and is gaining traction as a lightweight alternative to Spark or PostgreSQL for analytical tasks.

2.2.4. Other Technologies

Beyond the core tools described above, several alternative technologies are used in modern geospatial data pipelines. These tools are widely recognized for their capabilities and are commonly deployed in production environments for specialized use cases.

- **Osmium** is a high-performance C++ library and command-line toolset for working with OpenStreetMap data . It provides efficient, low-level access to OSM data structures (nodes, ways, relations, and changesets) and supports reading and writing various OSM file formats. The library is designed for performance and memory efficiency when processing large OSM datasets, making it well-suited for custom preprocessing, filtering, and format conversion tasks. Osmium is particularly valuable for scenarios requiring high-performance processing or tight integration with compiled applications. [15]
- **Apache Sedona** (formerly GeoSpark) is a distributed spatial processing framework that extends Apache Spark and Apache Flink with spatial data types and operations. It provides distributed spatial datasets and Spatial SQL for large-scale geospatial analytics. The framework supports various spatial data formats including GeoParquet, Shapefile, WKT, and WKB, and includes spatial indexing and query optimization for distributed environments. While primarily designed for big data scenarios, Sedona also offers local execution modes for development and testing. [16]

2.3. Standards

This section outlines key data standards governing geospatial data storage, exchange, and processing. These formats define how spatial features and attributes are encoded, compressed, and accessed, enabling interoperability across tools, libraries, and platforms.

2.3.1. Parquet

Apache Parquet is a columnar storage format widely used in modern data engineering and analytical workflows. It organizes data into row groups, column chunks, and data pages, enabling high-performance access patterns particularly suited for analytical workloads. This architecture

enables independent compression and encoding of each column, thereby optimizing both performance and storage utilization. [17]

A defining characteristic of Parquet is its comprehensive metadata framework. Beyond data storage, it maintains detailed structural information across multiple levels, including min/max statistics at the row group and column chunk levels.These metadata components enable predicate pushdown and data skipping, allowing query engines to bypass irrelevant data during execution, significantly improving query performance.

Parquet supports primitive types (e.g., int32, float, boolean) and extensible logical types, allowing for semantic richness without compromising structural simplicity.

2.3.2. GeoParquet

GeoParquet is an extension of the Apache Parquet columnar storage format, purpose-built for geospatial data. It combines the performance and compression benefits of Parquet with metadata and geometry support essential for spatial analytics. As a result, GeoParquet has emerged as a high-performance alternative to legacy formats such as GeoJSON and Shapefile, and serves as a cornerstone for modern, cloud-native geospatial workflows. [18]

Key Features

- Geometry Column Support: Spatial data is stored in dedicated geometry columns, typically encoded as Well-Known Binary (WKB), Well-Known Text (WKT), or GeoArrow. In contrast, standard Parquet treats such data as generic binary or string types without spatial awareness.
- Geospatial Metadata: GeoParquet embeds metadata describing geometry types, coordinate reference systems (CRS), bounding boxes, etc. This metadata, standardized by the Open Geospatial Consortium, enables geospatial tools to interpret and process spatial data accurately, addressing a limitation of standard Parquet.
- Spatial Query Optimization: Leverages Parquet's columnar storage to enable selective access to only the required attributes, minimizing I/O operations. Further enhanced by bounding box metadata and spatial indexing hints that allow query engines to skip irrelevant row groups or pages, significantly improving spatial query performance.
- Interoperability: GeoParquet integrates seamlessly with geospatial libraries and tools like GeoPandas, DuckDB (with the spatial extension), Apache Sedona, and QGIS, enhancing its utility in a variety of GIS workflows.
- Cloud and Big Data Compatibility: Like Parquet, GeoParquet is optimized for cloud-native environments (e.g., S3) and distributed processing frameworks, leveraging Parquet's columnar efficiency and compression.

GeoParquet's enhancements and compatibility with the broader Parquet ecosystem have made it the preferred format for scalable, portable, and maintainable geospatial data pipelines.

Anatomy of a GeoParquet File

The structure of a GeoParquet file builds on Parquet's hierarchical organization while adding support for geospatial data. At the top level, a GeoParquet file consists of one or more row groups, each containing column chunks for all columns including the geometry data (e.g., storing Points). The file metadata includes a `'geo'` key containing a JSON object with version information, primary geometry column designation, and a detailed columns object. This object specifies various geometry column properties such as encoding, geometry types, bounding box, etc. This enables spatial query optimizations and correct interpretation of the spatial data.

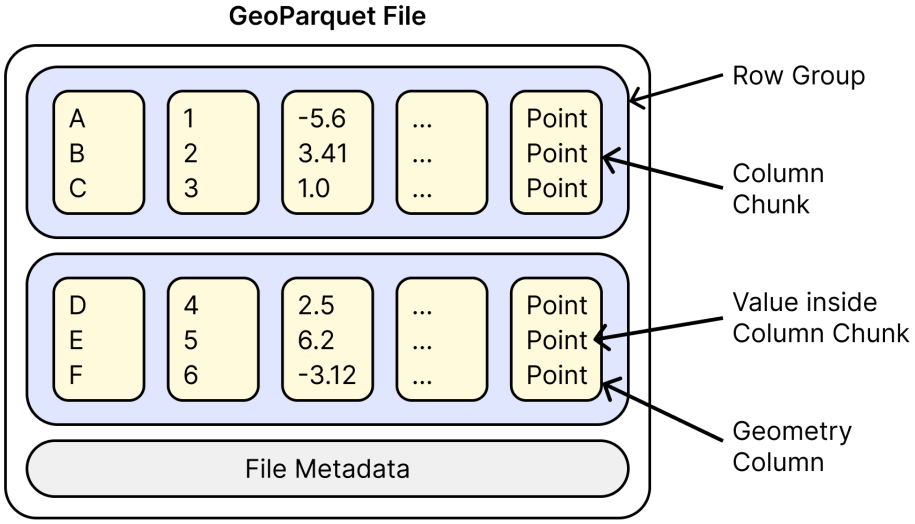


Figure 5: Anatomy of a GeoParquet File

GeoParquet's influence in Geospatial Processing

GeoParquet represents a paradigm shift from traditional database-centric geospatial processing to a cloud-native approach. This transformation addresses key limitations of conventional systems while offering new advantages.

Traditional Database Challenges:

- Infrastructure Overhead: Requires persistent database servers with 24/7 operation, leading to continuous cloud/hardware expenses.
- Operational Complexity: Demands specialized expertise for maintenance, optimization, and performance tuning.
- Scaling Challenges: Vertical scaling hits hardware limits while horizontal scaling introduces data partitioning complexity.
- Resource Intensive: Complex spatial operations on large datasets require significant computational resources.

GeoParquet Advantages:

- Cloud-Native Architecture: Utilizes object storage (e.g., S3) without managed database services, being more cost-effective and easier to operate.
- Optimized Query Performance: Leverages columnar storage and filter pushdown, allowing modern query engines like DuckDB to deliver SQL-like querying.
- Simplified Operations: Eliminates traditional database administration overhead while maintaining robust query capabilities through standardized file formats.

This approach makes geospatial processing more accessible and cost-effective, particularly for read-heavy analytical workloads. The combination of file-based storage with database-like querying addresses the needs of modern geospatial data workflows.

2.3.3. Other Standards

Beyond GeoParquet, several established and widely supported geospatial formats and standards play a key role in data exchange and interoperability.

- **GeoJSON** is a text-based format built on JSON for encoding geographic features like points, lines, and polygons. It is easy to use, human-readable, and well-supported in web mapping and APIs. However, it is inefficient for large datasets due to its verbosity and lack of binary or columnar optimization. [19]
- **Shapefile** format, developed by Esri, remains one of the most used vector formats in GIS. Despite lacking modern features like coordinate metadata or efficient storage, its widespread support across tools like QGIS and ArcGIS secures its ongoing relevance, particularly in legacy systems. [20]
- **SpatioTemporal Asset Catalog** (STAC) is an open specification that standardizes the description of geospatial data, enabling interoperable discovery and access to assets in cloud-native workflows. It uses a JSON-based format to organize spatiotemporal metadata, simplifying data management and integration. This structure ensures efficient search and compatibility across diverse platforms and tools. [21]

2.4. Data Access Tools

A variety of tools support accessing, querying, and visualizing OpenStreetMap and Overture Maps datasets. These tools facilitate cloud-native workflows and enable both local and remote data exploration in modern GIS environments.

2.4.1. QGIS

QGIS is a leading open-source desktop GIS application used for editing, visualizing, and analyzing geospatial data. Supporting a wide variety of formats, it features a strong plugin ecosystem for integrating external data sources such as OpenStreetMap or cloud-hosted datasets.

The GeoParquet Downloader Plugin allows QGIS to access GeoParquet files stored in cloud object storage, such as AWS S3. Users can specify a public or signed URL pointing to a GeoParquet file and load spatial features directly into QGIS. It supports spatial filtering, allowing users to view only relevant slices of large datasets such as Overture Maps. Integration with QGIS's core functionality ensures full support for styling, querying, and combining layers in a map project. [22]

2.4.2. DuckDB

DuckDB excels at querying remote GeoParquet files directly via HTTP(S) and S3 paths. Its spatial extension enables users to perform complex geospatial operations including filtering, aggregation, and spatial joins without downloading entire datasets. This makes DuckDB particularly suitable for cloud-native workflows that demand fast, columnar access to large geospatial datasets.

DuckDB's efficiency stems from several architectural features. It automatically recognizes Hive-style directory structures (e.g., `/theme=places/country=CH`), enabling partition pruning which allows skipping entire directory trees that don't match the query conditions. This is particularly valuable for geographically distributed datasets, as it significantly reduces I/O operations.

Filter pushdown further enhances performance by evaluating predicates at the storage level before loading data into memory. When combined with GeoParquet's spatial indexing, this ensures that only relevant row groups are processed, rather than scanning entire files. The columnar storage format further optimizes performance by allowing direct access to specific

attributes without reading entire rows, which is especially beneficial for analytical queries that typically operate on column subsets. [23]

Together, these features make DuckDB a powerful tool for cloud-native geospatial data processing, offering an optimal balance between file-based storage flexibility and traditional database query capabilities.

2.4.3. Other Tools

The following tools complement the main data access solutions, offering specialized functionality for working with geospatial data in various contexts.

- **ogr2ogr** is a command-line utility from the OGR Simple Features Library (part of GDAL) that specializes in vector data conversion and processing. It enables format-to-format transformations (supporting 200+ formats), coordinate system transformations, attribute filtering, and spatial operations. While it leverages GDAL's drivers, including the Parquet driver for GeoParquet support, ogr2ogr specifically provides the command-line interface for these operations, making it particularly useful for scripting and batch processing of geospatial data. [24], [25]
- **Overpass Turbo** is a web interface for the Overpass API that allows users to craft and execute custom spatial queries against OpenStreetMap data. Results can be visualized interactively and exported in formats like GeoJSON. Useful for lightweight, ad hoc data extraction. [26]
- **QuickOSM Plugin** provides querying OSM data via the Overpass API. It allows tag-based filtering and bounding box constraints, but relies on an external API and is more suited for exploratory, small-scale use cases. [27]
- **OvertureMaestro** is a Python library built on PyArrow that simplifies working with OvertureMaps data. It provides a high-level interface for processing the dataset, supports multiprocessing, and exports to GeoParquet. [28]
- **parquet-wasm** is a WebAssembly-based library that enables reading Parquet and GeoParquet files directly in the browser. Ideal for edge computing, offline scenarios, or browser-based geospatial apps that need local, lightweight access to structured data. [29]

3. Requirements

Due to the autonomous, server-side nature of the pipeline, a detailed use case diagram was not required. The system runs without direct user interaction, processing OpenStreetMap (OSM) data and publishing it in GeoParquet format for geospatial analysts. Instead of modeling interactions, the thesis focuses on functional and non-functional requirements to describe the system's expected behavior and performance.

3.1. Actors

- **Geospatial Analysts**: professionals who specialize in analyzing spatial data to extract meaningful insights about the world. They possess strong domain knowledge and are skilled in using geospatial tools and libraries. Their primary focus is on analyzing, interpreting, and visualizing data rather than managing infrastructure.
- **Data Engineers**: technical experts who are responsible for the maintenance and development of the pipeline. They are skilled in writing code and have a deep understanding of computer systems. They are responsible for ensuring that the pipeline is running smoothly and that the data is being processed correctly.

3.2. Functional Requirements

Following are the functional requirements of the system in the form of user stories:

ID	FR-1
Actor	Geospatial Analyst
Priority	High
StoryPoint	As a geospatial analyst, I want to access OSM data for the D-A-CH-LI region pre-processed in GeoParquet, so that I can begin my analysis immediately without handling complex data transformations.
Acceptance Criteria	The data is available in the GeoParquet format.

ID	FR-2
Actor	Geospatial Analyst
Priority	High
StoryPoint	As a geospatial analyst, I want to query the OSM data using tools like QGIS and DuckDB, so that I can leverage my existing skills and workflows to extract insights efficiently.
Acceptance Criteria	The data can be queried with: • QGIS • DuckDB Optional: • PostGIS • Python / Jupyter Notebooks • Browser with parquet-wasm

ID	FR-3
Actor	Geospatial Analyst
Priority	High
StoryPoint	As a geospatial analyst, I want to receive regular updates to the D-A-CH-LI OSM dataset, so that my analyses reflect the most current geospatial information available.
Acceptance Criteria	The data must be updated weekly. Optionally, the data is updated daily.

ID	FR-4
Actor	Geospatial Analyst
Priority	Low
StoryPoint	As a geospatial analyst, I want access to usage examples, release information, and documentation, so that I can quickly understand how to work with the GeoParquet files in my tools.
Acceptance Criteria	Usage examples, release information, as well as documentation of the schema are provided on a static website.

ID	FR-5
Actor	Data Engineer, Geospatial Analyst
Priority	High
StoryPoint	As a data engineer and geospatial analyst, I want the schema of the data to be compatible with Overture Maps (themes Places and Divisions), so that I don’t have to learn a new schema and can use my existing knowledge.
Acceptance Criteria	The schema is compatible with Overture Maps (themes Places and Divisions).

ID	FR-6
Actor	Data Engineer
Priority	High
StoryPoint	As a data engineer, I want to process OSM data efficiently using Python-based tools, so that I can leverage a familiar ecosystem to build and maintain the Data Processing pipeline.
Acceptance Criteria	The focus of the pipeline is on Python-based tools, with optional other languages, if necessary.

ID	FR-7
Actor	Data Engineer
Priority	Low
StoryPoint	As a data engineer, I want to monitor the processing time and performance of the pipeline, so that I can ensure it meets the sub-24-hour processing goal for D-A-CH-LI.
Acceptance Criteria	The processing time and performance of the pipeline can be monitored.

ID	FR-8
Actor	Data Engineer
Priority	Low
StoryPoint	As a data engineer, I want the pipeline to store each output dataset with a unique version number and allow for easy reversion to an older dataset version if quality checks fail, so that I can maintain a history of the data and revert to previous versions if needed.
Acceptance Criteria	Each output dataset is stored with the date of the release as version number and can be reverted to an older version if quality checks fail.

ID	FR-9
Actor	Data Engineer
Priority	Low
StoryPoint	As a data engineer, I want comprehensive documentation of the pipeline architecture and processes, so that future maintainers can understand and improve the system.
Acceptance Criteria	Comprehensive documentation of the pipeline architecture and processes is provided in the README of the Git-Repository.

3.3. Non Functional Requirements

ID	NFR-1
Title	Processing Time
Requirement	The Data Processing pipeline must process the full D-A-CH-LI OSM dataset into GeoParquet format in under 24 hours, ensuring timely availability of updated data.
Acceptance Criteria	The pipeline processes the full D-A-CH-LI OSM dataset into GeoParquet format in under 24 hours, measured from the start to the end of the pipeline.

ID	NFR-2
Title	Geographic Scalability
Requirement	The pipeline architecture must be designed to scale from Switzerland-sized datasets to planet-wide OSM data with minimal redesign.
Acceptance Criteria	The pipeline demonstrates linear scalability when adding countries and the system design includes parallel processing capabilities to handle potential planet-scale data with appropriate infrastructure scaling.

ID	NFR-3
Title	Query Performance
Requirement	A typical client query like “all restaurants in D-A-CH-LI” must take no longer than 3 minutes.
Acceptance Criteria	A DuckDB query, filtering for all restaurants in the region D-A-CH-LI takes no longer than 3 minutes.

ID	NFR-4
Title	Access Controls and Security
Requirement	Access Controls and security features are included.
Acceptance Criteria	The only access to the data is through the static website and the S3-compatible endpoint. S3 prefixes outside of the release prefix must not be publicly accessible.

4. Design

This chapter outlines the technical design of the Cadence Maps data pipeline, structured into two sections. The first section covers the architecture of the system through established software engineering models, including data flow representations and C4-based architectural diagrams. The second section documents the architectural decisions that shaped the system, presenting the evaluation of technologies and design choices that led to the final architecture. This includes the rationale behind selecting specific tools and patterns to ensure modularity, reproducibility, and performance, based on the project's requirements.

4.1. Architecture

The architecture of the Cadence Maps data pipeline follows a modular and containerized design. It separates concerns into well defined functional units, ensuring scalability and reusability. This section describes the high-level data flow, the system context, and the technical container structure based on the C4 model.

4.1.1. High-Level Data Flow

The data flow diagram (Figure 6) provides an overview of how data traverses through the processing pipeline and which containers contribute to the overall system functionality.

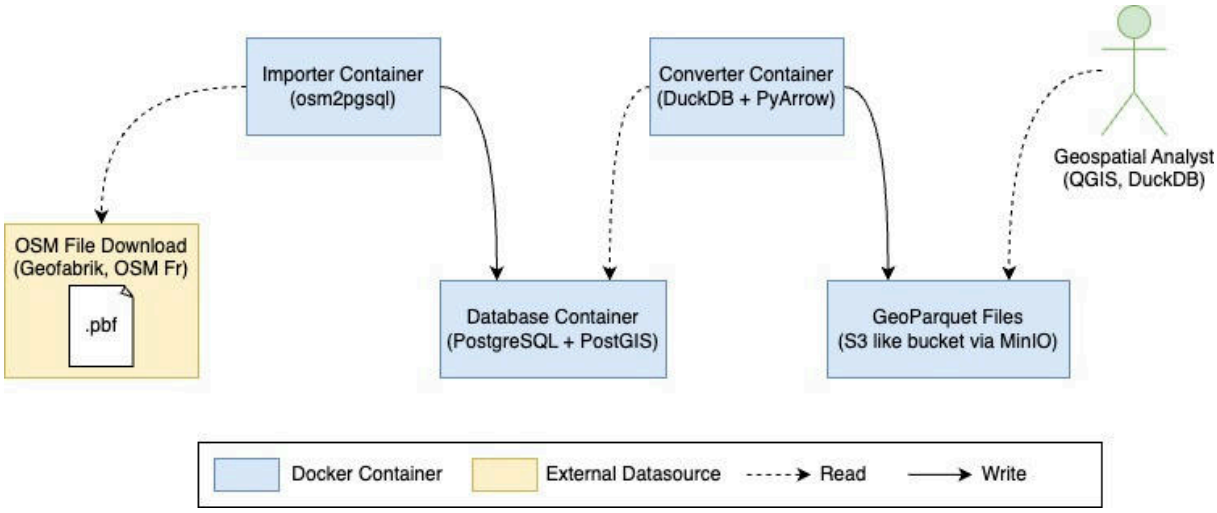


Figure 6: Data Flow Diagram

The pipeline follows the classic ETL pattern:

- **Extract:** OpenStreetMap data in .pbf format is downloaded from Geofabrik or OpenStreetMap France as a fallback.
- **Transform:** The data is parsed and saved into a relational database. From there, a converter module writes it to GeoParquet files.
- **Load:** The transformed GeoParquet files are stored in an object storage system (MinIO) which supports S3-like access.

4.1.2. System Context (C4 Level 1)

The system context diagram (Figure 7) outlines the high-level interaction between core actors, external systems, and the application.

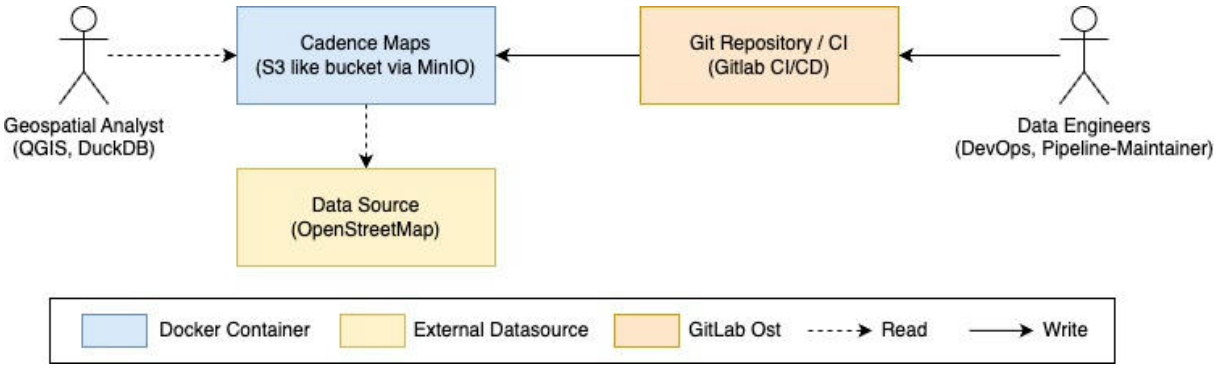


Figure 7: System Context Diagram

The architecture involves the following stakeholders:

- **Geospatial Analysts** access the published GeoParquet datasets via Tools like QGIS or DuckDB SQL.
- **Data Engineers** maintain the system via GitLab CI/CD pipelines and contribute updates to code and configuration

An additional external data source is integrated:

- **OpenStreetMap** acts as the external raw data source

4.1.3. Container Architecture (C4 Level 2)

The container architecture diagram (Figure 8) illustrates the technical structure of the system and the relationships between individual Docker-based components.

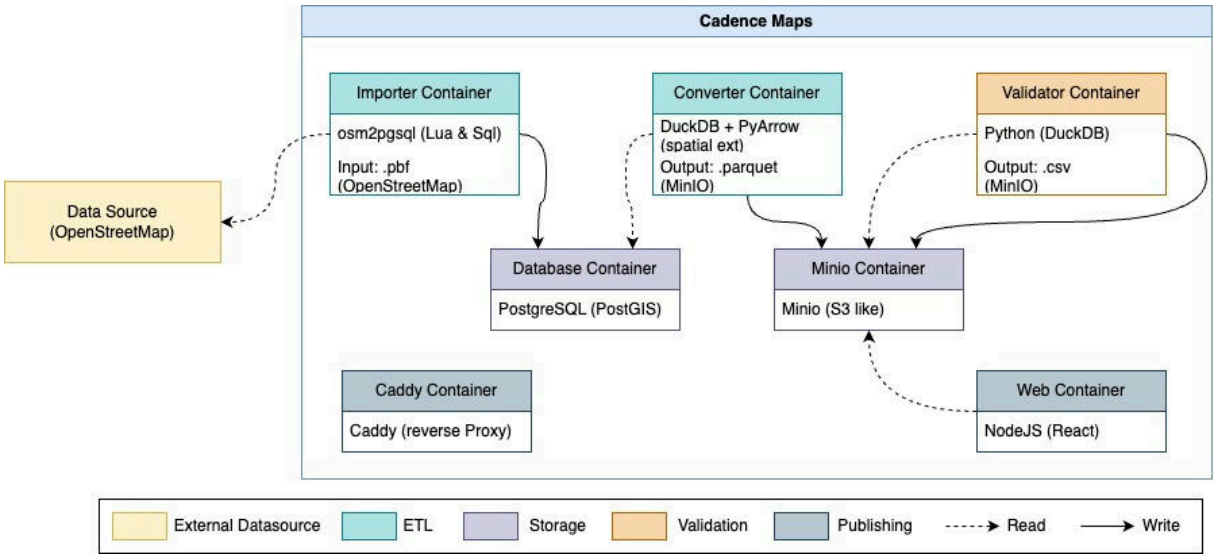


Figure 8: Container Architecture Diagram

To enhance modularity and clarity, containers are grouped by their respective roles within the pipeline:

- Data Processing Stage
  - Importer Container: Uses osm2pgsql with Lua mappings to import .pbf data into PostGIS.
  - Converter Container: Reads from the database and writes partitioned .parquet files using DuckDB and PyArrow.
- Storage & Serving
  - Database Container: A PostgreSQL database extended with PostGIS for data storage between Importer and Converter
  - MinIO Container: Serves as object storage for GeoParquet and CSV outputs.
- Validation
  - Validator Container: Performs quality checks on the parquet data using DuckDB and produces .csv summaries as well as .csv diff files.
- Publishing
  - Web Container: A NodeJS frontend providing basic access and information.
  - Caddy Container: A reverse proxy that routes domain-based HTTP traffic to the respective containers, enabling clean URL mapping and SSL termination.

4.1.4. Reflection on Architecture

The use of Docker for components such as PostgreSQL, MinIO, and Caddy was mainly driven by the requirement that the pipeline must run on the institute's infrastructure. Containerization ensured a reproducible and portable setup across different environments with minimal manual configuration.

For most production scenarios, however, this level of abstraction may be unnecessary. It is recommended to use a locally installed PostgreSQL database and run Caddy or a similar reverse proxy directly on the host system to reduce complexity and simplify debugging.

4.2. Evaluation

The evaluation section provides the rationale behind key decisions made throughout the project. It explains why specific technologies and design strategies were chosen and how these choices align with project requirements. Each subsection addresses a major design dimension and justifies the selected approach through comparison, reasoning, and contextual relevance.

4.2.1. Architecture Decision: Country-Based Processing

From the outset, the project was scoped to support the D-A-CH-LI region (Germany, Austria, Switzerland, Liechtenstein). Therefore, a country-based processing approach was the natural choice. This strategy not only fits the requirement but also enables efficient scaling: each country can be processed independently and in parallel using dedicated container instances.

A global or continental based transformation would have introduced considerable complexity in terms of compute resources, processing time, and partitioning strategies. The country-based approach reduces these concerns and aligns well with a containerized architecture, allowing us to parallelize imports and conversions while keeping processing time predictable and resource usage bounded.

Additionally, this modular approach simplifies debugging and validation, since issues can be traced to a specific region without affecting others.

Table 1 summarizes key advantages and disadvantages of the country-based processing approach:

	Advantages	**Disadvantages**
Country-based	• Enables parallel processing per country • Simplifies debugging by region • Scales well as new countries are added • Lower memory/compute requirements • Users and analysts often want country-specific views or exports	• Fragmented view for global use cases • Disputed borders and territories are a sensitive matter
Planet-based	• Unified global dataset • Easier for global analysis tasks	• Extremely high memory/CPU demands • Longer processing time • Harder to parallelize cleanly • Debugging more complex and risky

Table 1: Comparison of Country-based and Planet-based Processing Approaches

Y - Statement:
In the context of supporting the D-A-CH-LI region, facing requirements for efficient scaling and manageable processing, we decided for a country-based transformation strategy and neglected global or continental based processing, to achieve modularity, parallelization and predictable performance, accepting downside of regional fragmentation and increased orchestration complexity.

4.2.2. Architecture Decision: Containerization

One of the key architectural decisions was the use of Docker containers to structure the pipeline. This choice was partially guided by project constraints: the pipeline is expected to be deployable on the infrastructure of the supervising institute. Containerization ensures portability, reproducibility, and environment isolation, making it significantly easier to run the entire setup across different machines with minimal manual configuration. It also aligns with DevOps best practices and simplifies orchestration for CI/CD pipelines.

Each core function (import, convert, storage and validation) is encapsulated within its own Docker container, enabling modular development and execution. This separation of concerns supports maintainability and deployment flexibility. In particular, the architecture supports horizontal scaling by design: the import and transformation steps for each country are executed in parallel using separate instances of the importer container and converter container. This approach significantly reduces processing time and prepares the system for future geographic expansion.

Y - Statement:
In the context of deploying the pipeline on external institutional infrastructure, facing requirements for portability and reproducibility, we decided for Docker-based containerization and neglected direct host-based or virtual machine–based setups, to achieve isolated, modular, and scalable execution, accepting downside of added orchestration complexity and container overhead.

4.2.3. Comparison of Importer Tools

The transformation of OSM data into the desired format is a complex task, as the OSM schema is not directly compatible with the Overture Maps schema. OSM's primitive data model, which consists solely of nodes, ways, and relations, requires the construction of geometry types (points, lines, polygons, etc.). Additionally, to make it fully compatible with the Overture Maps schema, the OSM tagging system must be transformed to match the Overture Maps schema. This involves mapping OSM's free-form key-value pairs to the structured schema while preserving essential attributes. Current tooling support for this specific transformation remains limited, with only a few specialized solutions available.

Two primary tools were evaluated for processing OSM PBF files, each with distinct architectural approaches:

- `osm2pgsql`
  - Operates in flex mode with custom Lua scripts to define table schemas and data transformations
  - Directly streams OSM data into a PostgreSQL/PostGIS database
  - Provides native support for spatial operations and complex geometries
  - Enables SQL-based querying and post-processing of the imported data
- `PyOsmium`
  - Python interface to the high-performance C++ Osmium library
  - Loads and processes OSM data in-memory using Python data structures
  - Stores results in Pandas DataFrames for further processing

Performance Benchmark

To assess the performance of the two tools, a benchmark was performed using the small PBF extract of Liechtenstein. The task was to transform the PBF data into their respective intermediate formats, implementing an incomplete subset of the Overture Places schema. The results were as follows:

- `osm2pgsql`: ≈ 2 seconds
- `PyOsmium`: ≈ 57 seconds

This test reveals a substantial performance advantage for `osm2pgsql`, even in its single-threaded flex mode. While `PyOsmium` benefits from underlying C++ performance via `libosmium`, the overhead from Python-based transformation and geometry handling dominates already for small datasets.

Summary

Overall, `osm2pgsql` offers a more robust, maintainable, and performant solution for converting OSM data, additionally offering the possibility to use SQL-based post-processing workflows. While `PyOsmium` provides a Python-based alternative that seems suitable for small tasks, its performance limitations make it less suitable for large-scale production use.

The tool chosen for production use was `osm2pgsql` with Lua flex mode, due to its superior performance, spatial operation capabilities, and compatibility with SQL-based post-processing workflows. It is well known in the geospatial community and therefore shouldn't be a barrier for a data engineer to maintain the pipeline.

4.2.4. Converter Tooling

The second architectural pillar of the pipeline is the Converter, which is responsible for extracting spatial data from PostgreSQL (populated by the importer) and exporting it as partitioned GeoParquet files.

A core design consideration for the converter was the partitioning strategy. Partitioning is used to split large datasets into smaller, manageable files, such that spatial queries using bounding boxes (BBOX) touch as few files as possible. Since GeoParquet files are typically written once and queried often, the initial partitioning must be efficient, scalable, and maximize spatial locality. The choice of tools, libraries, and techniques in the converter was therefore heavily influenced by the partitioning strategy adopted.

Evaluated Partitioning Methods

This evaluation examines partitioning strategies for dividing spatial data into separate files. These strategies are grouped into three conceptual categories, with representative algorithms implemented for each. A detailed evaluation of these methods, including comprehensive performance metrics and analysis, is available in the appendix, Section 13 (see page 68).

1. **Space-Filling Curves**

   Methods that map multi-dimensional data to a single dimension while preserving spatial locality, ideal for creating a linear ordering of spatial data.

   - **Hilbert Curve**: Implemented natively in DuckDB, this approach maps 2D coordinates to a 1D index using space-filling curves. While offering perfect row distribution, it produces irregular partition shapes unsuitable for efficient spatial queries.

2. **Recursive Space Subdivision**

   Techniques that recursively split the space into smaller, manageable regions, typically using axis-aligned or grid-based approaches.

   - **KD Tree**: Implemented using a DuckDB common table expression (CTE) with recursive median calculations, this method performs alternating axis splits (longitude/latitude). It provides clean rectangular partitions with excellent spatial locality (no bounding box overlap) and near-perfect row distribution, though it requires specifying the number of splits rather than target partition sizes.
   - **KDB Tree**: This adaptive variant of KD Tree, implemented in both DuckDB and Sedona, continues splitting based on row limits. The DuckDB implementation was preferred over Sedona's for its better balance and partition shapes, while both maintain excellent spatial locality with axis-aligned rectangular regions.
   - **QuadTree**: Implemented in Apache Sedona, this method uses recursive quadrant division to create grid-like rectangular regions, offering good spatial locality with minimal bounding box overlap.

3. **Hierarchical Geospatial Tiling** Global tiling systems that provide predefined, hierarchical spatial partitions, particularly well-suited for planet-scale data.

   - **GeoHash**: Implemented using Sedona's native support, this method produces non-contiguous regions when hash-regions are merged, leading to significant bounding box overlap and poor spatial locality.
   - **S2**: Using the s2geometry community extension via DuckDB, this method provides good partitions with spherical quadrilateral tiling. While showing great spatial organization, it can produce small partitions at cell boundaries.
   - **H3**: Implemented using both DuckDB and PostgreSQL's H3 extension, this hexagonal global tiling offers good spatial organization with adaptive resolution, though it may produce small partitions at boundaries.

After initial testing, the following partitioning approaches were eliminated before proceeding to scaling tests:

- **Hilbert Curve**: Produced poor partition shapes unsuitable for spatial queries despite excellent row distribution.
- **KD Tree**: Required specifying the number of splits rather than target partition sizes. Eliminated in favor of KDB Tree.
- **Sedona KDB Tree**: DuckDB's implementation provided better-balanced partitions with faster performance.
- **GeoHash**: Poor partition shapes due to diagonal merges of regions leading to significant bbox overlap and non-contiguous partitions.

Partitioning Methods Scaling Analysis

For the scaling analysis, the research focused on the four methods that demonstrated the best partition quality:

- S2 (DuckDB)
- KDB Tree (DuckDB)
- QuadTree (Sedona)

- H3 (PostgreSQL and DuckDB)

The benchmarking utilized a dataset from Switzerland, with additional scaling tests on larger datasets from Germany and the United States. Results in Figure 9 show that DuckDB-based methods (KDB Tree and S2) consistently outperform alternatives.

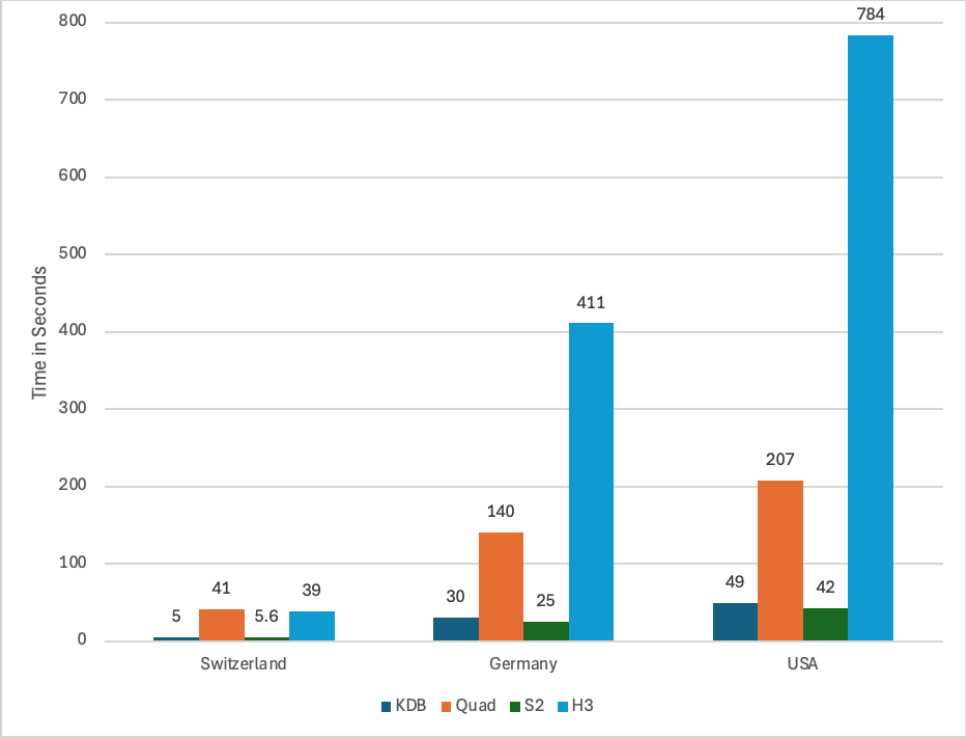


Figure 9: Partitioning Algorithm Scaling Comparison

Partitioning Methods Query Tests

Notably, query performance showed minimal variation between methods, confirming that partitioning strategy primarily impacts preprocessing efficiency rather than runtime query performance.

Summary

The converter's design prioritized efficient spatial data organization in GeoParquet files. The evaluation of seven partitioning methods across three categories (Space-Filling Curves, Recursive Space Subdivision, and Hierarchical Geospatial Tiling) led to the development of a KDB Tree implementation using DuckDB as the primary tool. This solution provides optimal partitioning for country-based datasets and demonstrated superior performance in terms of preprocessing efficiency and partition quality compared to alternative approaches.

The KDB Tree implementation, built on top of DuckDB, establishes a robust foundation for processing OSM data into spatially optimized GeoParquet files. While S2 (also available through DuckDB) remains a strong alternative for global-scale processing, the KDB Tree implementation offers the best balance of performance and partition quality for regional focus.

4.2.5. Converter Implementation Design

This section documents the key design decisions for the OSM GeoParquet converter, focusing on file-internal structure and metadata handling to ensure optimal performance and accessibility.

Sorting within Partitions

While file-level partitioning improves query performance by reducing the number of files that need to be read, it does not address the organization of data within each partition file. To optimize spatial queries further, the data within each partition file must be sorted based on spatial proximity.

This is achieved through the use of space-filling curves, which map 2D spatial data into a 1D space while preserving locality. The two most common approaches are the Hilbert curve and Z-Order curve. Between these, the Hilbert curve is generally preferred for its superior spatial locality and balance, which leads to better query performance. [30]

An alternative approach involves using the Z-Order curve with the bigmin optimization. The bigmin optimization improves Z-Order performance by enabling efficient skipping of irrelevant regions during range queries. It works by calculating the next relevant Z-Order value that falls within the query range, allowing the system to jump over large sections of irrelevant data. However, this approach comes with several drawbacks:

1. It requires additional calculations for Z-Order curve values
2. It needs an external library for the bigmin jump optimization
3. It necessitates an additional column in the GeoParquet file
4. It introduces complexity in the processing pipeline

Given these considerations, the Hilbert curve was selected for its simplicity and effectiveness. The Hilbert curve provides good spatial locality out-of-the-box, and the filtering can be efficiently handled by the row group min/max statistics in the Parquet files. DuckDB's native sorting capabilities are utilized to sort the data within each partition by its Hilbert index, ensuring optimal spatial clustering.

GeoParquet Metadata Handling

The GeoParquet export process was designed to ensure proper handling of spatial metadata. Although DuckDB offers native support for GeoParquet export, it currently lacks the capability to include certain metadata attributes that are considered essential for effective spatial data usage and optimization. Specifically, the following metadata elements were identified as important:

- Coordinate Reference System (CRS) Information: This attribute encodes the spatial reference system associated with the dataset, enabling accurate interpretation of spatial coordinates and facilitating transformations between different projections.
- Covering Metadata: This attribute references the bounding box columns corresponding to the geometry. Its inclusion supports efficient spatial queries by enabling spatial indexing and filtering.

While the GeoParquet specification does not require these metadata attributes, their inclusion enhances the utility, correctness, and performance of spatial datasets.

The initial implementation involved exporting the dataset using DuckDB's native Parquet functionality, followed by a post-processing step with PyArrow to inject the necessary metadata. However, this two-step process introduced additional complexity and increased input/output operations.

To address these limitations, the export workflow was ultimately restructured to utilize PyArrow exclusively for the entire file-writing process. This approach simplifies the export pipeline, reduces I/O overhead, and ensures the inclusion of relevant metadata.

STAC Consideration

The SpatioTemporal Asset Catalog (STAC) standard was evaluated as a potential solution for cataloging the generated GeoParquet files. STAC provides comprehensive metadata management and discovery capabilities for geospatial assets through a standardized specification. However, after careful consideration, it was determined that implementing STAC would introduce unnecessary complexity relative to the project's requirements and scope.

Key considerations that led to this decision include:

- The provided data already has a well-defined, consistent organization with clear naming conventions
- The additional metadata management overhead of STAC would not provide sufficient value given the project's scale
- The cloud-optimized GeoParquet format already includes self-describing metadata and efficient query capabilities

While STAC offers robust cataloging features, the project's current needs are sufficiently met by the existing GeoParquet metadata and file organization.

4.2.6. Validation

One of the initial project requirements was to establish a mechanism ensuring data integrity and protection against potential vandalism within the dataset. During a design consultation with the academic advisor, the feasibility of implementing such validation mechanisms was thoroughly evaluated. It was concluded that comprehensive vandalism detection and mitigation would necessitate advanced logic or artificial intelligence based approaches, which would have exceeded the defined scope of this project and introduced considerable maintenance complexity.

An external service, Clearance[1], was considered for this purpose. Clearance is an open-source validation tool tailored for OpenStreetMap data, utilizing rule-based and statistical anomaly detection techniques to identify suspicious edits. Despite its capabilities, the integration and sustained operation of such a system would require administrative resources and operational oversight that could not be guaranteed within the institutional constraints.

As a practical compromise, a lightweight validation prototype was implemented. This mechanism performs consistency checks on key administrative names such as "Rapperswil" or "St. Gallen" across subsequent dataset versions. Given that such names are typically stable over time, any unexpected variation is interpreted as a potential indicator of vandalism.

If discrepancies are detected, the affected dataset is not released immediately. Instead, it is retained within a staging directory on the MinIO object storage system. A manual review and approval process—executed via a dedicated GitLab pipeline—must then be conducted before the data is promoted to the release state. This hybrid approach balances operational simplicity with essential quality assurance for critical regional data.

4.2.7. OSM ID Handling and GERS ID Decision

OpenStreetMap's nodes, ways, and relations share the same ID space, meaning a node, way, and relation can all have the same numeric ID, which prevents their use as globally unique identifiers. [31] The Global Entity Reference System (GERS) by the Overture Maps Foundation

[1] https://clearance.teritorio.xyz/

was considered, which provides 128-bit identifiers that remain stable across data releases and updates [32].

While GERS offers robust ID stability, its implementation would require maintaining a comprehensive history of object versions and complex conflation logic—requirements that exceed the scope of our stateless, extract-based processing model.

Instead, the following three practical strategies based on recommendations from giswiki.ch were considered [33]:

1. Prefixed IDs Encode object type using a prefix:
   - N123456 for node
   - W789012 for way
   - R345678 for relation
2. Custom osm_id with separate type field
   Store the numeric OSM ID in a dedicated osm_id column and provide an additional field (e.g., osm_type) to indicate the entity type. This improves filtering and indexing performance in SQL-based systems.
3. Bit-Shifted IDs
   Encode both type and ID in a single numeric value:
   - node_id = osm_id * 10
   - way_id = osm_id * 10 + 1
   - relation_id = osm_id * 10 + 4

The prefixed ID format was selected for its optimal balance of human readability, technical simplicity, and compatibility with existing OSM tooling, while avoiding the infrastructure overhead of GERS.

5. Implementation

This chapter outlines the most relevant implementation decisions made during the development of the system. Rather than documenting every technical detail, it focuses on the key design trade-offs, configurations, and decisions that influenced the pipeline's structure and behavior.

5.1. Pipeline Architecture and Workflow

The system is designed as a modular, cloud-optimized data processing pipeline. Its primary objective is to convert raw OpenStreetMap (OSM) data into analysis-ready, partitioned GeoParquet files that conform to the schemas defined by OvertureMaps. The process is broken down into distinct stages, each responsible for one aspect of the transformation, and coordinated via a GitLab-based CI/CD, started by a schedule job once weekly.

5.1.1. Import Stage

Each pipeline execution starts with importing .osm.pbf extracts from Geofabrik and as a fallback OSM.FR, using the tool osm2pgsql. This tool parses OSM's complex graph-based data model (nodes, ways, and relations) into structured PostgreSQL tables using a custom Lua-based mapping configuration. Two thematic datasets, places and divisions, are created during this step, each aligned with the target schema.

Each country is processed in a separate CI/CD job. While osm2pgsql itself does not support parallel execution in flex mode, the pipeline achieves parallelism by spawning one import job per country, each targeting a separate PostgreSQL database.

5.1.2. Transformation and Conversion Stage

After import, a dedicated converter component (implemented in Python) connects to the PostgreSQL/PostGIS database. It extracts the normalized spatial data into DuckDB, applies spatial partitioning using a K-D-B tree (optimized via Hilbert ordering), and writes the result as GeoParquet files using PyArrow.

The partitioning is optimized for spatial querying and cloud storage access patterns. The resulting files are uploaded to a MinIO bucket under the staging/ prefix, using a path structure that includes the processing date and country code (e.g., staging/2025-06-06/country=CH/theme=places/…). This structure supports temporal versioning and downstream validation workflows.

5.1.3. Validation and Release Management

To ensure data quality, a validation pipeline compares the freshly generated dataset with the latest released version by executing a set of predefined SQL queries. If no differences are found, the dataset is automatically promoted from staging/ to release/. If differences exist, they are saved as CSV diffs and made available for manual review.

This mechanism supports DataOps best practices and protects the integrity of the released dataset. Once validated, the datasets can also be promoted by manually triggering a release job via the CI/CD interface, offering controlled publication for reviewed outputs.

5.2. Importer

The importer is implemented using osm2pgsql. As defined in the thesis scope, the implementation of two thematic datasets were required: places (e.g., cities, shops) and divisions (e.g., administrative boundaries). These two themes are aligned with the target schemas defined by Overture.

To achieve this, the mapping logic was carefully designed to retain compatibility with the defined schema while ensuring the output is consistent across different countries.

5.2.1. Places Category Mapping

To fulfill the Places theme requirements, a dynamic mapping logic was implemented to assign structured categories to OSM features based on their tags. This logic is designed to align with the Overture Places schema, which organizes entities in a hierarchical, multi-level taxonomy.

The implementation loads two CSV files:

- one defining the category hierarchy,
  e.g., eat_and_drink > restaurant > italian_restaurant,
- and one mapping OSM tag combinations to category keys,
  e.g., amenity=restaurant, cuisine=italian → italian_restaurant.

The function `getCategory()` matches OSM features against these mappings by checking if all required tag-value pairs are present in the object. When a match is found, the associated taxonomy path is returned as a list of category levels, ordered from most specific to general.

Final Schema of categories

Each matched feature includes a categories field of the following structure:

```
{
    "primary": "italian_restaurant",
    "alternate": ["restaurant", "eat_and_drink"]
}
```

The primary category represents the most specific match (leaf of the taxonomy), while the alternate categories represent higher-level categories.

Design Critique and Recommendation

Although the current implementation aligns with the Overture schema by separating the matched categories into a primary and an alternate field, we identified this as a suboptimal design for practical usage. The schema only allows a single category match, even though OSM features can often logically fit into multiple categories. This restriction forces a prioritization that may not reflect the user's intent or use case.

Instead, we recommend a simpler and cleaner structure: a single list of ordered categories, from most specific to general. For example:

```
{
    "categories": ["italian_restaurant", "restaurant", "eat_and_drink"]
}
```

This format eliminates ambiguity, avoids artificial prioritization, and provides a deterministic, consistent path for querying and analysis. We have communicated this proposal[2] as feedback to the Overture project for potential consideration in future schema revisions.

[2] https://github.com/orgs/OvertureMaps/discussions/360

5.2.2. Division mapping subtypes

The Overture Divisions theme is divided into three main types: `division`, `division_area`, and `division_boundary`. Each type serves a specific purpose in representing administrative boundaries and their associated geometries.

Setting the subtype of an administrative division is of fundamental importance, as it determines how the `"hierarchies"`, `"parent_division_id"`, `"capital_division_ids"` and `"region"` properties of the division type is filled.

A mapping logic was implemented to assign subtypes to OSM administrative boundaries based on their tags. This logic is designed to align with the Overture Divisions schema, which organizes administrative entities into a hierarchy of subtypes and classes.

Each type in the divisions theme has specific subtypes or classes and they are described in the following sections.

Type Division

This type represents the administrative divisions themselves, such as countries, states, or regions. Geometries are saved as points, which are typically used to represent the centroid or a point on surface of the division. The `division` type is used to provide a high-level overview of administrative entities without detailed geometries.

The official Wiki page for OSM admin_levels [34] was used as a reference and adapted to align in the best possible way with the actual hierarchy and levels of real-world administrative boundaries for Switzerland.

Most ways, relations and points that describe a division, defined in the OSM data, have a tag `"admin_level"` with a value between 2 and 11. The subtype is solely determined by this tag. Switzerland uses levels 2-10 but the value 11 is also considered as a valid subtype because some places were tagged as such in the OSM data.

```
{

  "2": "country",
  "3": "dependency",
  "4": "region",
  "5": "county",
  "6": "localadmin",
  "7": "locality",
  "8": "macrohood",
  "9": "neighborhood",
  "10": "microhood",
  "11": "microhood"

}
```

Type Division Area

This type represents the actual geometries of administrative divisions, such as polygons or multipolygons. It is used to provide detailed spatial information about the administrative areas.

OSM tags can have multiple values since it is a flexible public tagging system. The following values of the tag `"boundary"` are used to select which OSM features to include in the divisions theme:

```
[
  "administrative",
  "local_authority",
  "political",
  "special_economic_zone",
  "state",
  "province",
  "region",
  "district"
]
```

Type Division Boundary

This type represents the boundaries between administrative division areas, the geometries are of type line or multiline that define the edges of administrative regions.

Only the boundaries of division areas with a subtype of `"country"`, `"region"` or `"county"` are considered for the `division_boundary` type.

This ensures that only significant administrative boundaries are included, while smaller or less relevant boundaries are excluded from this type.

OSM features with missing admin_level

In some cases, OSM features may not have an `"admin_level"` tag, which would lead to missing entries in the divisions theme. To address this, we implemented a fallback mechanism that assigns an `"admin_level"` based on the `"place"` tag. If the `"admin_level"` is missing, the `"admin_level"` is determined by the `"place"` tag value, which can be one of the following:

```
{
  "country": "2",
  "county": "5",
  "megacity": "6",
  "city": "8",
  "municipality": "8",
  "town": "9",
  "village": "10",
  "quarter": "10",
  "suburb": "10",
  "borough": "10",
  "hamlet": "11",
  "city_block": "11"
}
```

Class Field

In addition to the subtype, the Overture Divisions theme defines a `class` field that categorizes the administrative divisions into broader classes. This field is derived from the `place` tag and provides a high-level classification of the division types.

Overture Maps defines the following classes for the division type:

```
["megacity", "city", "town", "village", "hamlet"]
```

Division area and division boundary types can only have `"land"` or `"maritime"` tag.

Class Field Mapping

The class field is derived from the `"place"` tag in OSM data. If the value of the tag is not one of the defined classes, the following mapping is applied:

```
{
  "class": {
    "municipality": "town",
    "county": "megacity",
    "suburb": "village",
    "quarter": "village",
    "city_block": "hamlet",
    "borough": "town"
  }
}
```

If the `"place"` tag is not present but the “`"admin_level"` tag is set, the following mapping is applied:

```
{
  "6": "megacity",
  "8": "city",
  "9": "town",
  "10": "village",
  "11": "hamlet"
}
```

5.3. Converter

The converter is implemented in Python, utilizing DuckDB for spatial SQL queries and PyArrow for writing the final GeoParquet files.

The process starts by reading spatial data from PostgreSQL into DuckDB. If the dataset surpasses a certain size, the data is spatially partitioned using a recursive KDB-tree algorithm implemented in DuckDB SQL. Each partition is then sorted using the Hilbert curve, extracted from DuckDB and written as a GeoParquet file using PyArrow.

5.3.1. Code Structure

The converter is organized into three Python modules to ensure clarity and maintainability:

- `converter.py`: This is the main entry point of the converter. It handles the initialization of the DuckDB connection, loads required extensions, attaches the PostgreSQL database, and orchestrates the conversion process by iterating over the configured datasets.
- `converter_config.py`: Contains the ConverterConfig class, which encapsulates all configuration parameters for each dataset, including SQL queries, partition sizes, and output paths. This file also defines the list of all dataset configurations and shared type definitions.
- `converter_logic.py`: Implements the core logic for converting, partitioning and writing the data to GeoParquet files.

This modular structure separates configuration, orchestration, and data processing logic, making the codebase easier to extend and maintain.

5.3.2. GeoParquet Parameter Optimization

The optimization of GeoParquet output files requires tuning of two parameters: the partition size (rows per file) and the row group size (rows per internal Parquet unit). As of the time of writing, the geospatial community has not established formal best practices for these parameters, research and experimentation to determine optimal values is still ongoing.

To determine the optimal parameter configuration, a series of benchmarking experiments were conducted, focusing on query performance across varying file and row group sizes. The objective was to balance file size, row count, and read efficiency.

The testing process included:

1. Estimating row sizes through analysis of representative data samples
2. Measuring query performance across datasets ranging from 10′000 to 8′000′000 rows
3. Evaluating row group sizes between 50′000 and 200′000 rows
4. Recording and comparing average query execution times

Based on the results, the following configurations were established (Table 2):

Theme/Type	Max Rows/Partition	Row Group Size
`places/place` `divisions/division`	10′000′000	100′000
`divisions/division_area`	100′000	1′500
`divisions/` `division_boundary`	1′000′000	10′000

Table 2: GeoParquet Parameter Optimization

These parameters address significant variations in memory usage between different data types. For instance, point geometries in the `places/place` type demonstrate substantially lower memory requirements compared to the complex polygon geometries in `divisions/division_area`. Consequently, partition sizes were adjusted to maintain individual file sizes within an optimal range of approximately 500 MB to 1 GB, ensuring efficient data processing while accommodating the inherent characteristics of each dataset.

5.4. Validation Logic

To address the requirement of ensuring data consistency and detecting potential vandalism in administrative names, a lightweight validation prototype was developed. The validation logic is implemented in Python using DuckDB for querying and MinIO for dataset access and diff storage.

The system compares the current staging dataset against the most recent released dataset by executing a predefined set of SQL queries (e.g., selecting primary names for cities, towns, or villages). Each query template includes placeholders for both the dataset states (staging or release) and the processing date. These placeholders are dynamically resolved at runtime.

For each query:

- The results from the current and previous dataset are fetched and sorted.
- A row-level diff is calculated using pandas to detect any discrepancies.
- If differences are found, a CSV diff file is created and uploaded to MinIO under `validation/{date}/diffs/`.

Additionally, a summary.csv file is generated containing the mismatch count for each query. If no differences are detected across all queries, the staging dataset is automatically promoted to the release directory. Otherwise, the files remain in staging, and manual approval via CI/CD is required to proceed.

This approach allows targeted validation of high-confidence attributes such as administrative names.primary, which are stable over time and likely to expose unintended changes. The system can be extended with more queries or adapted to check other themes in the future.

5.5. Performance

5.5.1. Database configuration

The Database is minimally configured optimized for the pipeline's requirements. While the osm2pgsql V2 documentation provides recommendations for PostgreSQL server tuning [35], initial testing revealed stability issues when processing larger datasets such as Germany or France. The recommended settings resulted in premature connection termination during osm2pgsql operations, leading to incomplete data imports.

The deployment environment consists of a server equipped with 8 CPU cores and 128 GB of RAM, enabling substantial memory allocation to PostgreSQL. The following configuration was implemented to balance performance and stability:

```
shared_buffers=32GB
work_mem=32GB
temp_buffers=2GB
```

5.5.2. Parallelism via Pipeline Jobs

A significant limitation of osm2pgsql in flex mode is it's lack of multi-threaded or multi-processor execution, which leads to performance bottlenecks when importing large .pbf datasets. To address this constraint, parallelism was introduced at the orchestration level rather than within the tool itself.

The GitLab CI/CD pipeline leverages a matrix job setup, where each country is processed independently in a separate job. Each job is configured with its own .pbf download URL, country code, and database name.

Instead of deploying separate PostgreSQL instances, a single PostgreSQL server is reused, and a dedicated database is created for each country (e.g., Germany, Switzerland, Austria, etc.). This ensures isolation between runs while keeping the infrastructure lightweight and manageable.

This setup enables the system to process multiple countries in parallel, reducing the overall runtime of the import stage significantly. While introducing some complexity in terms of managing multiple databases, it offers an effective and scalable solution to the single-threaded limitation of osm2pgsql.

5.5.3. Postprocessing in pgsql instead of osm2pgsql

Osm2pgsql processes OSM data in a specific sequence: first nodes, then ways, and finally relations [36]. This sequential processing means that when nodes and ways are being handled, their potential membership in relations remains unknown. For features requiring relation context (e.g., `'hierarchies'`, `'parent_division_id'`, `'region'`, etc.), osm2pgsql flex mode offers a

“reprocessing” step. During this step, the Lua function `select_relation_members()` is invoked for each relation, allowing the specification of which nodes or ways require reprocessing with relation context. The implementation can utilize built-in helpers like `osm2pgsql.way_member_ids()` or custom selection logic as needed.

The initial implementation of relation member selection revealed substantial performance limitations, particularly when reprocessing extensive datasets such as those for Germany and France.

To address these performance constraints, the postprocessing logic was moved into a dedicated SQL script that executes after the osm2pgsql import operation. This script uses PostGIS spatial functions, specifically `ST_Contains` and `ST_PointOnSurface` to determine the relationships between features. This modification presents itself as significantly faster and more efficient than the osm2pgsql reprocessing step, as it leverages PostgreSQL’s optimized spatial indexing and query execution capabilities.

5.6. Security

To ensure the security and data integrity of the pipeline, several measures were implemented as described below.

Pipeline variables

The pipeline uses GitLab CI/CD variables to store sensitive information such as database credentials, MinIO access keys, and other configuration parameters. These variables are encrypted and only accessible to the pipeline jobs that require them. This approach prevents hardcoding sensitive information in the codebase and prevents credential exposure in logs or source control.

MinIO Bucket Access Controls

The MinIO object storage system provides multiple mechanisms for configuring bucket access controls. The pipeline uses the MinIO Client (mc) utility[3] to set up and manage bucket configurations and access control policies.

The data is only accessible through MinIO’s S3-compatible endpoint.

The following policy is applied to the bucket `cadencemaps`, enforcing strict access restrictions. This policy permits anonymous read-only access exclusively to objects within the `release/` directory, while maintaining restricted access to the `staging/` and `validation/` directories. The policy adheres to the JSON-based Amazon IAM policy format [37] and is applied during the pipeline’s setup step.

```
1   "Version": "2012-10-17",
2   "Statement": [
3     {
4       "Effect": "Allow",
5       "Action": ["s3:List*", "s3:GetObject", "s3:Get*"],
6       "Resource": ["arn:aws:s3:::cadencemaps/release/*"],
7       "Principal": "*"
8     },
9     {
```

[3] https://min.io/docs/minio/linux/reference/minio-mc.html

```
      "Effect": "Allow",
      "Action": ["s3:ListBuckets", "s3:ListBucket","s3:GetBucketLocation"],
      "Resource": ["arn:aws:s3:::cadencemaps"],
      "Principal": "*"
    },
    {
      "Effect": "Deny",
      "Action": ["s3:GetObject", "s3:Get*", "s3:List*", "s3:Put*"],
      "Resource": ["arn:aws:s3:::cadencemaps/staging/*"],
      "Principal": "*"
    },
    {
      "Effect": "Deny",
      "Action": ["s3:GetObject", "s3:Get*", "s3:List*", "s3:Put*"],
      "Resource": ["arn:aws:s3:::cadencemaps/validation/*"],
      "Principal": "*"
    }
  ],
  "Sid": "PublicReadForGetBucketObjects"
```

The pipeline user only has read and write access to the bucket, and no rights to modify the bucket configuration or access controls. The following policy is applied to the pipeline user:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PipelineUserFullBucketAccess",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket",
        "s3:GetBucketLocation",
        "s3:GetBucketVersioning"
      ],
      "Resource": ["arn:aws:s3:::cadencemaps"]
    },
    {
      "Sid": "PipelineUserFullObjectAccess",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:PutObject",
        "s3:DeleteObject",
        "s3:DeleteObjectVersion",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload"
      ],
      "Resource": ["arn:aws:s3:::cadencemaps/*"]
    },
    {
      "Sid": "PipelineUserMultipartUpload",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucketMultipartUploads"
```

```
32        ],
33        "Resource": ["arn:aws:s3:::cadencemaps"]
34      }
35    ]
```

Database

The PostgreSQL database, which temporarily stores imported OSM data during processing, is configured to accept connections exclusively from within the Docker network. This configuration ensures that only authorized pipeline components can access the database, preventing any external connections. Following the completion of the import and conversion processes, the database is automatically removed.

Release Website

The release website does not contain any sensitive information and is designed to be publicly accessible. It provides a user-friendly interface with quick links that point to the various releases of the datasets on the MinIO Bucket.

Server security

The server hosting the MinIO instance needs to be secured and managed by the system administrator. While server administration falls outside this project's scope, it is crucial to ensure that the server is properly configured, updated, and monitored to prevent unauthorized access or data breaches.

5.7. Contribution to the GeoParquet Ecosystem

In alignment with the project's cloud-native design, a pull request (#122[4]) was submitted to the open-source GeoParquet Downloader plugin for QGIS, developed by Chris Holmes. The plugin enables users to directly load GeoParquet datasets into QGIS via DuckDB's remote querying capabilities.

The enhancement enables users to load GeoParquet files directly from self-hosted MinIO buckets by leveraging DuckDB's remote querying capabilities. This aligns closely with the project's architectural design, which stores published datasets in a MinIO-based object store with S3-compatible access.

Although the pull request has not yet been merged, it represents a valuable improvement to the plugin and demonstrates a contribution back to the tools and ecosystem this project builds upon.

[4] https://github.com/cholmes/qgis_plugin_gpq_downloader/pull/122

6. Testing and Verification

6.1. Data Quality Testing

To evaluate the quality of Cadence Maps, selected manual comparisons with Overture Maps were conducted. While the dataset is based on community maintained OpenStreetMap data and cannot be modified directly, it is essential to ensure that the data is suitable for analysis.

6.1.1. Places Zürich, Switzerland

Overture Maps (Red): 28′543 POIs
Cadence Maps (Blue): 30′022 POIs

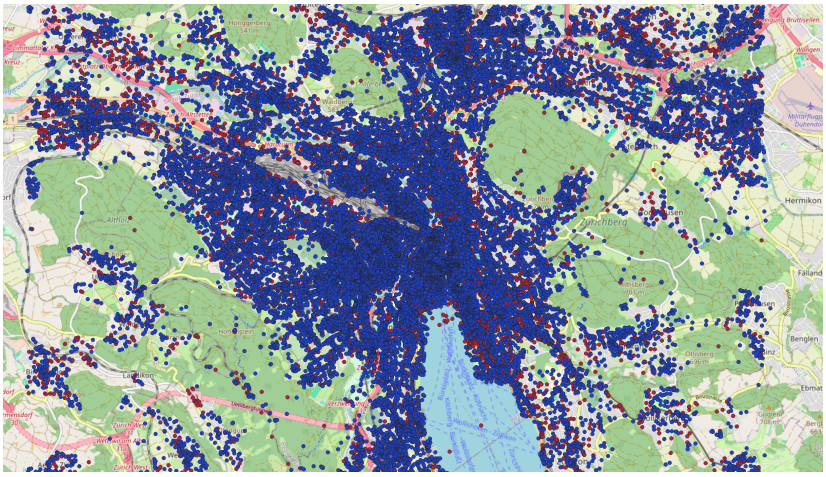


Figure 10: Comparison Overture Maps vs Cadence Maps Places (Zürich, Switzerland)

Result: Cadence Maps contains more POIs and shows a visibly denser and more complete coverage than Overture Maps in the Zürich area (Figure 10).

6.1.2. Places Dortmund, Germany

Overture Maps (Red): 10′582 POIs
Cadence Maps (Blue): 11′529 POIs

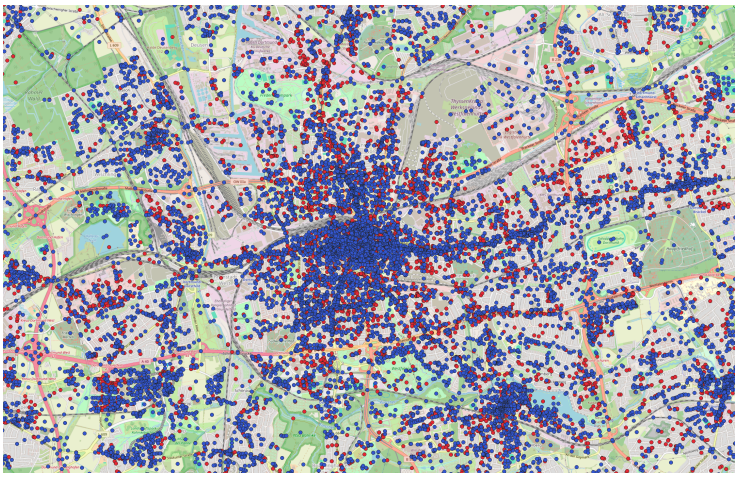


Figure 11: Comparison Overture Maps vs Cadence Maps Places (Dortmund, Germany)

Result: Cadence Maps shows a higher POI count and more complete spatial coverage than Overture Maps in the Dortmund area (Figure 11).

6.1.3. Places Vienna, Austria

Overture Maps (Red): 65′965 POIs
Cadence Maps (Blue): 76′616 POIs

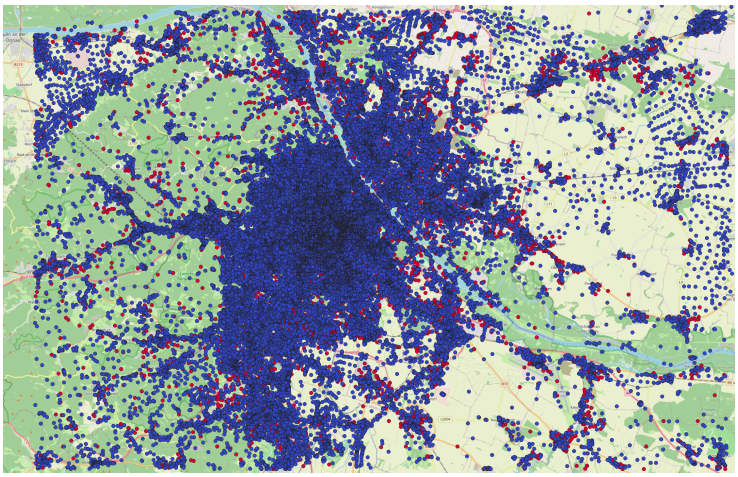


Figure 12: Comparison Overture Maps vs Cadence Maps Places (Vienna, Austria)

Result: Cadence Maps provides significantly more POIs and a denser spatial distribution than Overture Maps in the Vienna region (Figure 12).

6.1.4. Division Area Rapperswil-Jona, Switzerland

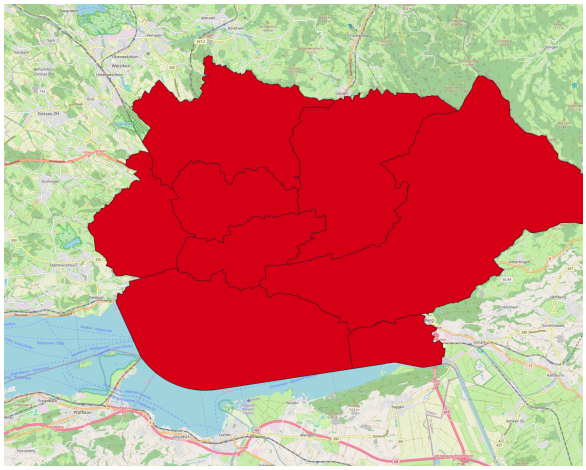
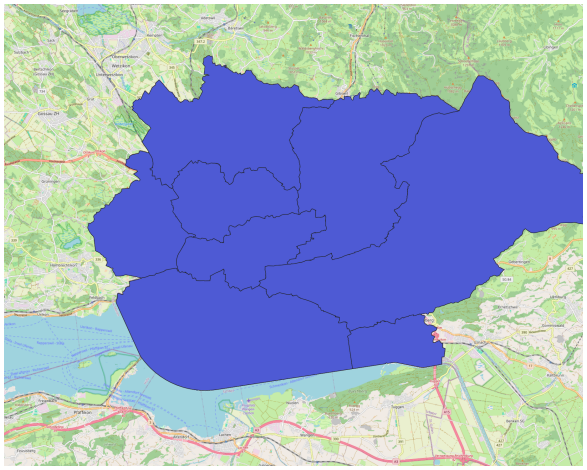
Overture Maps	Cadence Maps
	

Table 3: Comparison Overture Maps vs Cadence Maps Divisions Rapperswil-Jona

Result: Cadence Maps and Overture Maps show identical administrative boundaries in the tested regions, indicating schema alignment (Table 3).

6.1.5. Automatic Vandalism Detection

A prototype for automatic vandalism detection is integrated into the pipeline and runs with every import, serving as an example for future, more advanced detection mechanisms. The Data quality tests are documented in the implementation chapter Section 5.1.3 (see page 39).

6.2. Functional Requirements

ID	Comment
FR-1	Pass, the data is available in the GeoParquet format via the MinIO bucket.
FR-2	Pass, the data can be accessed using DuckDB, QGIS, Python, and Jupyter Notebooks.
FR-3	Pass, a cron job is scheduled to run every Thursday at 1:05 AM.
FR-4	Pass, usage examples, release information and documentation is available on a static website, accessible via https://cadencemaps.infs.ch/.
FR-5	Pass, the schema is compatible with OvertureMaps (themes Places and Divisions), see appendix Section 15.2 (page 92).
FR-6	Pass, while the converter is implemented in Python, the importer uses osm2pgsql and lua scripts to transform the OSM data. This is accepted, because it is optionally allowed to use other languages and osm2pgsql is a well known tool in the geospatial community. The justification for this choice can be found in the design chapter Section 4.2.3 (see page 32).
FR-7	Pass, the processing time and performance of the pipeline can be monitored through the GitLab CI/CD pipeline. Optionally, additional monitoring tools can be used as mentioned in Section 8.3 (see page 59).
FR-8	Pass, the datasets are versioned using their release date as the unique identifier and can be reverted to an older version via the GitLab CI/CD pipeline if quality checks fail.
FR-9	Pass, the documentation is available in the README of the Git-Repository.

Table 4: Verification of Functional Requirements

6.3. Non Functional Requirements

6.3.1. NFR-1 Processing Time

Run	Processing time (min)
1	63
2	67
3	64
Average Time	65

Table 5: Processing Time for D-A-CH-LI Dataset, rounded to minutes

The results in Table 5 show that the time for processing the D-A-CH-LI dataset is around 1 hour and therefore well below the 24 hour limit.

6.3.2. NFR-2: Geographic Scalability

The scalability of the pipeline was tested by comparing processing times between two datasets:

- D-A-CH-LI (Germany, Austria, Switzerland, Liechtenstein)
- D-A-CH-LI-FR-IT (Germany, Austria, Switzerland, Liechtenstein, France, Italy)

The results in Table 6 show that the ratio between data size and sequential processing time is consistent, indicating linear scalability. A more detailed analysis of NFR 2 can be found in the appendix in Section 14 (see page 89).

Metric	D-A-CH-LI	D-A-CH-LI-FR-IT
Total Size	5′542 MB	12′081 MB
Size Ratio	1.0x	2.2x
Parallel Processing Time (avg.)	65 min	74 min
Sequential Processing Time (avg.)	79 min	174 min
Sequential Processing Time Ratio	1.0x	2.2x

Table 6: Processing Time Comparison

Key Findings

1. The pipeline demonstrates linear scalability, with processing time increasing proportionally to dataset size.
2. Parallel processing maintains efficiency, with only a 10-minute overhead when processing the larger dataset.
3. Individual country processing times remain largely consistent between test runs, indicating stable performance.

Global Scalability Projection

The performance analysis indicates a processing throughput of 65 MB/min for the slowest-performing region (France). Extrapolating this rate to a planet-scale dataset of 81 GB yields an estimated processing time of approximately 21 hours.

It is important to note that this projection assumes ideal hardware resource availability and does not account for potential system constraints or resource contention.

6.3.3. NFR-3 Query Performance

This NFR was tested with the following query, measured with the Explain Analyze statement in DuckDB. Tested on a MacBookPro with 32GB RAM and an M2 chip.

Query:

```
INSTALL httpfs;
LOAD httpfs;
INSTALL spatial;
LOAD spatial;
SET s3_endpoint='api.cadencemaps.infs.ch';
SET s3_url_style='path';
SELECT *
FROM read_parquet('s3://cadencemaps/release/2025-06-04/theme=places/type=
place/*/*', hive_partitioning=1)
WHERE
  country IN ('DE', 'CH', 'LI', 'AT')
  AND categories.primary = 'restaurant'
  OR 'restaurant' IN categories.alternate;
```

Run	Execution Time
Run 1	20.17 seconds
Run 2	19.17 seconds
Run 3	19.40 seconds
Run 4	19.68 seconds
Run 5	19.05 seconds

Table 7: NFR-3 Query Performance measurements

Table 7 shows that the query takes around 20 seconds to execute. The requirement of a maximum 3-minute execution time is therefore met.

Note: A test in QGIS revealed that a full download of the theme places and type place dataset for the D-A-CH-LI region until fully displayed took aproximately 1 minute and 30 seconds (500 MB/s download speed).

6.3.4. NFR-4 Access Controls and Security

The S3 endpoint is configured to enforce access restrictions, permitting access exclusively to the `release` directory while denying all other paths.

To validate these access controls, test queries were executed against restricted paths using DuckDB, as demonstrated below:

```
INSTALL httpfs;
LOAD httpfs;
INSTALL spatial;
LOAD spatial;
SET s3_endpoint='api.cadencemaps.infs.ch';
SET s3_url_style='path';

SELECT *
FROM read_parquet('s3://cadencemaps/staging/2025-06-04/theme=places/type=place/country=CH/*', hive_partitioning=1)
WHERE
  categories.primary = 'restaurant'
  OR 'restaurant' IN categories.alternate;


SELECT * FROM read_csv('s3://cadencemaps/validation/2025-06-07/summary.csv');

```

The test queries resulted in the following error responses:

```
HTTP Error:
HTTP GET error on 'https://api.cadencemaps.infs.ch/cadencemaps/staging/2025-06-04/theme%3Dplaces/type%3Dplace/country%3DCH/place_0.parquet' (HTTP 403)

HTTP Error:
HTTP GET error on 'https://api.cadencemaps.infs.ch/cadencemaps/validation/2025-06-07/summary.csv' (HTTP 403)

```

These responses confirm that access to both the `staging` and `validation` directories is properly restricted, as evidenced by the HTTP 403 Forbidden status code.

7. Results

The outcome of this Bachelor's thesis is the development of Cadence Maps, a modular, cloud-native data pipeline for geospatial data processing, accessible directly from cloud storage and usable without dedicated backend infrastructure. This system transforms OpenStreetMap (OSM) country extracts into spatially partitioned, schema-aligned GeoParquet files that are fully compatible with the Overture Maps schema ("Places" and "Divisions" themes). The pipeline currently supports Switzerland, Germany, Austria, and Liechtenstein (shown in Figure 13) and is designed for extension to additional regions or even global scale.

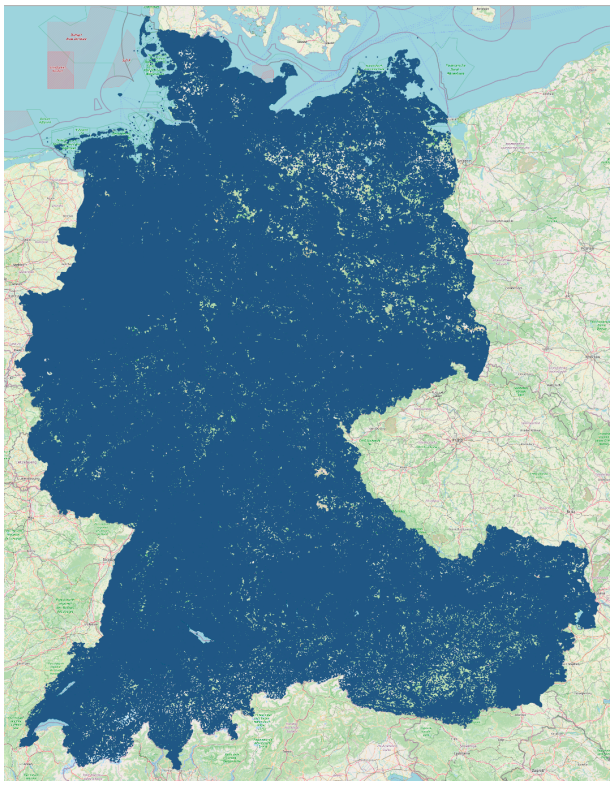


Figure 13: QGIS Screenshot of Cadence Maps' ~5.8M Places in the D-A-CH-LI region

7.1. Parallel Processing and Architecture

A central achievement is the pipeline's parallel processing architecture. By leveraging GitLab CI/CD matrix jobs, the system processes multiple countries simultaneously, substantially reducing processing time compared to sequential execution. Additional performance gains were realized by shifting computationally intensive reprocessing steps from osm2pgsql to PostgreSQL spatial SQL postprocessing.

7.2. Data Quality and Validation

To ensure data integrity, a lightweight validation framework prototype was implemented. This system automatically flags anomalies in critical regional identifiers (e.g., administrative names) and defers publishing in case of suspicious changes. Manual validation can be triggered via CI/CD before the final promotion of data to the release path, ensuring that only validated data is published and maintaining high data quality standards.

7.3. Performance and Scalability

Benchmarking results demonstrate that the pipeline reliably completes processing for the D-A-CH-LI region in slightly over an hour. Extrapolation from current performance metrics suggests

that, with adequate hardware, the pipeline could support global-scale processing within a 24-hour timeframe, given appropriate hardware scaling.

Real-world query benchmarks further validate scalability and responsiveness. For example, queries such as “all restaurants in D-A-CH-LI” using DuckDB consistently execute in around 20 seconds. This performance is achieved through:

- Efficient spatial partitioning using the KDB-tree method via DuckDB
- In-file sorting using Hilbert curves to preserve spatial locality
- Optimized row group sizing tailored to each dataset type

The performance of the pipeline can be monitored through the GitLab CI/CD pipeline. Optionally, additional monitoring tools can be used as mentioned in Section 8.3 (see page 59).

7.4. Compatibility and Integration

The resulting GeoParquet files are:

- Fully compatible with Overture Maps schemas
- Usable in tools such as QGIS, DuckDB, and Python (with DuckDB, GeoPandas, PyArrow)
- Optimized for performance through:
  - Hive-compatible S3 prefixes, enabling query engines to skip irrelevant partitions
  - Min/max metadata values for row groups, enabling query engines to skip irrelevant row groups within GeoParquet files

This compatibility allows analysts familiar with Overture Maps and standard GIS tools to easily incorporate the data into their existing workflows.

7.5. Comparison with Overture Maps

Manual comparisons demonstrate that Cadence Maps offers more comprehensive data coverage than Overture Maps. For instance:

- In Zürich, Cadence Maps contained 30,022 POIs, compared to Overture’s 28,543
- In Dortmund, 11,529 POIs versus 10,582 in Overture Maps
- Similar improvements were observed in Vienna and Rapperswil-Jona

These results underscore the reliability and completeness of OSM data when processed through this pipeline.

7.6. Documentation and Ecosystem Contribution

A static website was developed to provide release information, usage examples, and documentation for end users (Figure 14). Information for usage can also be viewed in the appendix Section 15 (see page 92). The project repository includes documentation in the form of a README file to support future maintainers.

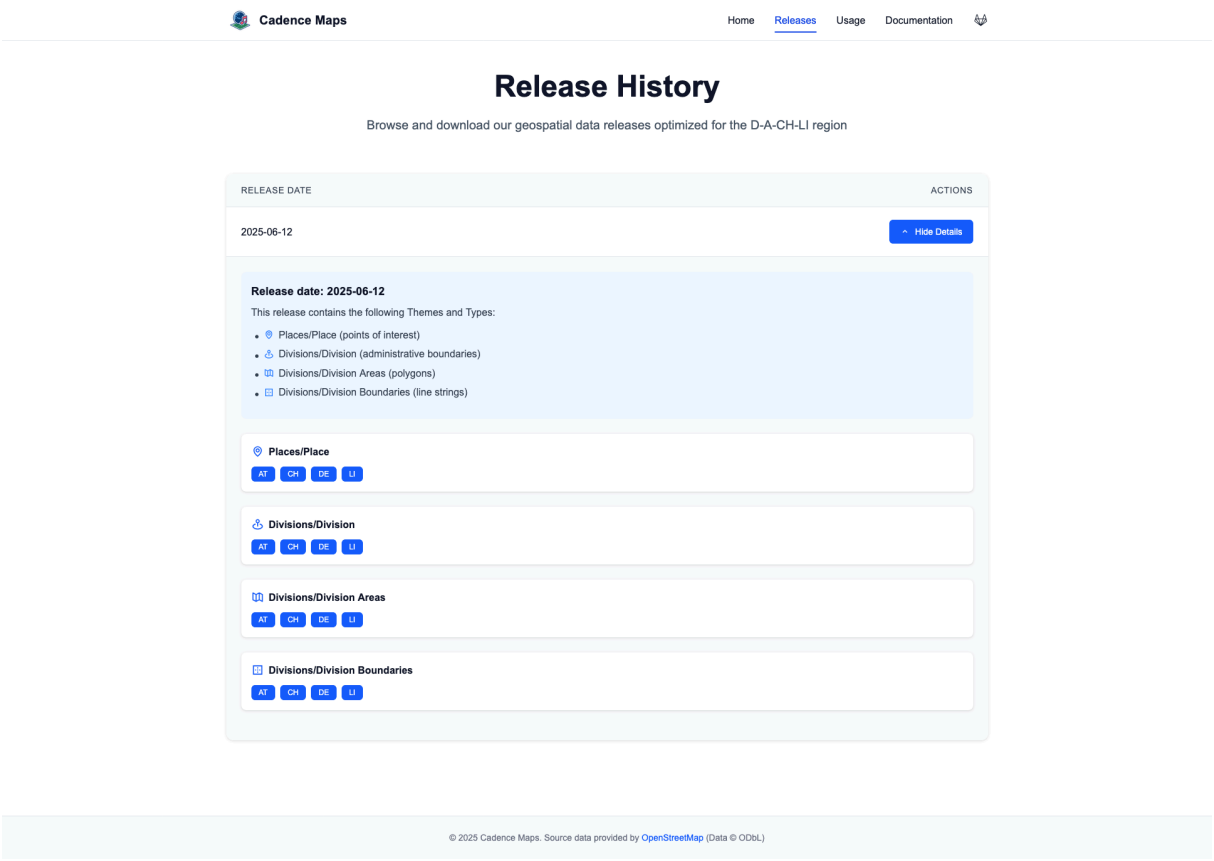


Figure 14: Cadence Maps website displaying the available releases

Additionally, the project contributed to the open-source ecosystem by submitting a pull request to the QGIS GeoParquet Downloader Plugin, enhancing its support for MinIO-based sources and enabling broader adoption of decentralized spatial data services.

8. Outlook

While the current implementation successfully enables scalable transformation of OpenStreetMap (OSM) data into partitioned GeoParquet files for the D-A-CH-LI region, several promising directions exist for future improvements and expansion.

8.1. Global Scaling Strategy

The pipeline design is inherently scalable and technically capable of processing global OSM datasets. A natural extension would be to move from country-based processing toward continent-based or even planet-wide ingestion. One possible approach is to import larger extracts (e.g., entire continents from Geofabrik) into a single PostgreSQL table and enrich them with an additional column containing the S2 Level 0 or Level 1 cell ID. The S2 geometry's hierarchical cell structure provides globally consistent, non-overlapping spatial boundaries that are perfectly suited for distributed processing. This enables efficient parallel processing by allowing the data to be partitioned into manageable chunks, with each thread handling a separate S2 cell. The system can adapt to varying data densities by starting with high-level S2 cells (e.g., level 10) and recursively combining them into larger cells when the point count is below a defined threshold, ensuring each partition contains an optimal number of points. The deterministic nature of S2 cell identifiers ensures consistent partitioning across different processing runs, while maintaining spatial locality and balancing the computational load. This approach would allow the pipeline to apply intelligent partitioning on a per-cell basis, improving spatial locality and distribution while simplifying storage management.

However, this shift requires careful consideration. Moving from country-based to continent-based or global imports would significantly increase hardware requirements. For example, osm2pgsql's memory usage can be several times the dataset size, with Germany's 4.3 GB dataset requiring up to 29 GB of RAM at peak usage.

Additionally, since users typically expect data organized by country or region, a global import would require either additional post-processing to split the data by country boundaries, or if keeping the S2 partitioning, users would need to employ geospatial functions to filter the data by country boundaries when querying, which can be significantly more time-consuming.

8.2. Improved Validation Logic

The current validation approach is functional but minimal. Future versions should introduce more comprehensive quality checks to verify structural consistency, schema conformance, and semantic integrity across datasets. Automated checks for unexpected changes (e.g., deleted or moved administrative boundaries, name mismatches, or suspicious feature density changes) would help ensure higher trust in the published GeoParquet files.

Integrating machine learning-based anomaly detection or external tools like Clearance could further enhance vandalism detection and data reliability. As usage scales, trust in data quality becomes increasingly important especially for downstream users in academia, urban planning, and infrastructure monitoring.

8.3. Infrastructure and Monitoring Recommendations

While the use of Docker throughout the project ensured portability and reproducibility, we recommend decoupling PostgreSQL, the release website and the reverse proxy (Caddy) from containerization in future deployments. Running these components natively on the host

system would reduce complexity, improve performance, and simplify debugging in production environments.

Pipeline Monitoring Tool

For the Monitoring of the pipeline performance and health, we recommend implementing a dedicated monitoring tool like Sentry[5] or Prometheus[6]. These tools can provide real-time insights into the pipeline's operational status, error rates, and performance metrics. This would enable proactive identification of issues, optimization of resource usage, and improved overall reliability.

8.4. Iceberg Integration for Long-Term Data Management

To further enhance scalability, traceability, and compatibility with big data ecosystems, future iterations of the pipeline could integrate Apache Iceberg. As of version 3, Iceberg supports native geospatial data types, making it a strong candidate for managing large volumes of GeoParquet files. Iceberg would enable:

- ACID-compliant transactions and atomic file management
- Time travel support for historical OSM snapshots
- Efficient data pruning and partition evolution
- Compatibility with distributed query engines like Apache Spark or Trino

These features would be particularly beneficial for global-scale datasets or scenarios involving frequent updates and historical analysis, such as monitoring urban development or land use change over time.

[5] https://sentry.io/
[6] https://prometheus.io/

Appendix

9. Bibliography

[1] OpenStreetMap contributors, “About OpenStreetMap.” [Online]. Available: https://wiki.openstreetmap.org/wiki/About_OpenStreetMap

[2] J.-F. Girres and G. Touya, “Quality assessment of the French OpenStreetMap dataset,” *Transactions in GIS*, vol. 14, no. 4, pp. 435–459, 2010, doi: 10.1111/j.1467-9671.2010.01203.x.

[3] C. Barrington-Leigh and A. Millard-Ball, “The world’s user-generated road map is more than 80% complete,” *PLOS ONE*, vol. 12, no. 8, Aug. 2017, doi: 10.1371/journal.pone.0180698.

[4] C. Barron, P. Neis, and A. Zipf, “A comprehensive framework for intrinsic OpenStreetMap quality analysis,” *Transactions in GIS*, vol. 18, no. 6, pp. 877–895, 2014, doi: 10.1111/tgis.12073.

[5] O. M. Foundation, “Overture Maps Foundation.” [Online]. Available: https://overture.org/

[6] O. M. Foundation, “Attribution and Licensing.” [Online]. Available: https://docs.overturemaps.org/attribution/

[7] O. M. Foundation, “Frequently Asked Questions.” [Online]. Available: https://overturemaps.org/about/faq/

[8] O. M. Foundation, “Overture Maps 2025-02-19.0 Release Notes.” [Online]. Available: https://docs.overturemaps.org/release/2025-02-19.0/

[9] osm2pgsql Developers, “osm2pgsql: Import OpenStreetMap data into PostgreSQL/PostGIS.” [Online]. Available: https://osm2pgsql.org/

[10] T. osm2pgsql Developers, “osm2pgsql Manual.” [Online]. Available: https://osm2pgsql.org/doc/manual.html

[11] P. G. D. Group, “PostgreSQL Documentation.” [Online]. Available: https://www.postgresql.org/docs/

[12] PostGIS Project Steering Committee, “PostGIS 3.3 Documentation.” [Online]. Available: https://postgis.net/documentation/

[13] D. Team, “Why DuckDB.” [Online]. Available: https://duckdb.org/why_duckdb.html

[14] D. Team, “Spatial Extension Documentation.” [Online]. Available: https://duckdb.org/2023/04/28/spatial.html

[15] J. Topf and others, “Libosmium: Fast and flexible C++ library for working with OpenStreetMap data.” [Online]. Available: https://osmcode.org/libosmium/

[16] A. S. Foundation, “Apache Sedona: A cluster computing framework for processing large-scale spatial data.” [Online]. Available: https://sedona.apache.org/

[17] A. S. Foundation, “Apache Parquet.” [Online]. Available: https://github.com/apache/parquet-format

[18] G. Community, “GeoParquet Specification 1.1.0,” 2025. [Online]. Available: https://geoparquet.org/releases/v1.1.0/

[19] S. G. Butler, Howard, Schaub, and others, “GeoJSON Format.” [Online]. Available: https://tools.ietf.org/html/rfc7946

[20] ESRI, “ESRI Shapefile Technical Description,” 1998. [Online]. Available: https://www.esri.com/content/dam/esrisites/sitecore-archive/Files/Pdfs/library/whitepapers/pdfs/shapefile.pdf

[21] R. E. Foundation and Contributors, “SpatioTemporal Asset Catalog (STAC) Specification.” [Online]. Available: https://stacspec.org/

[22] Q. Community, “QGIS GeoParquet Downloader Plugin.” 2024.

[23] H. Mühleisen and M. Raasveldt, “Efficient Querying of Parquet Files in DuckDB.” [Online]. Available: https://duckdb.org/2021/06/25/querying-parquet.html

[24] G. contributors, “GDAL: ogr2ogr utility documentation.” [Online]. Available: https://gdal.org/programs/ogr2ogr.html

[25] G. contributors, “GDAL: Parquet/GeoParquet driver documentation.” [Online]. Available: https://gdal.org/drivers/vector/parquet.html

[26] O. Contributors, “Overpass Turbo.” [Online]. Available: https://overpass-turbo.eu/

[27] E. Trimaille, “QuickOSM is a QGIS plugin to download data from Overpass server..” 2022.

[28] K. Raczycki, “OvertureMaestro.” [Online]. Available: https://github.com/kraina-ai/overturemaestro

[29] R. AI, “parquet-wasm: WebAssembly support for Parquet.” 2024.

[30] H. K. Dai and H. C. Su, “Clustering Analyses of Two-Dimensional Space-Filling Curves: Hilbert and z-Order Curves,” *SN Computer Science*, vol. 4, no. 1, p. 8, 2022, doi: 10.1007/s42979-022-01320-9.

[31] OpenStreetMap contributors, “Elements - OpenStreetMap Wiki.” [Online]. Available: https://wiki.openstreetmap.org/wiki/Elements

[32] O. M. Foundation, “Global Entity Reference System (GERS).” [Online]. Available: https://docs.overturemaps.org/gers/

[33] GIS Wiki contributors, “Permanent ID for OSM.” [Online]. Available: https://www.giswiki.ch/Permanent_ID_for_OSM

[34] O. contributors, “Tag:boundary=administrative - OpenStreetMap Wiki.” [Online]. Available: https://wiki.openstreetmap.org/wiki/Tag:boundary%3Dadministrative#Country_specific_values_%E2%80%8B%E2%80%8Bof_the_key_admin_level=*

[35] The osm2pgsql Developers, “osm2pgsql Manual: Tuning the PostgreSQL Server.” 2024. [Online]. Available: https://osm2pgsql.org/doc/manual.html#tuning-the-postgresql-server

[36] O. Contributors, “Osm2pgsql Manual: Stages in Flex Output.” [Online]. Available: https://osm2pgsql.org/doc/manual.html#stages

[37] A. W. Services, “AWS Identity and Access Management (IAM).” [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

10. List of Figures and Tables

11. Used Tools

Task	Tools
Project Organization	Confluence, Jira, Microsoft Teams, Gitlab, Outlook
Documentation	Visual Studio Code, Windsurf, Typst, Excel, Figma, Draw.io, ChatGPT, DeepL, Github Copilot
Code	Visual Studio Code, Windsurf, Grok, ChatGPT, Github Copilot

12. Used Libraries

Library	License
osm2pgsql	GPL-2.0
duckdb	MIT
pyarrow	Apache License 2.0
pandas	BSD 3-Clause
boto3	Apache License 2.0
heroicons	MIT
next	MIT
react	MIT

13. Partitioning

Spatial partitioning is a key element of the OSM GeoParquet Data Service pipeline. The goal is to split large point datasets (e.g., millions of OSM places) into smaller, manageable GeoParquet files, such that spatial queries using bounding boxes (BBox) touch as few files as possible. Crucially, partitioning is static—performed once during the pipeline and not adjusted at runtime, making efficient initial partitioning essential.

Each partitioning method is evaluated for how well it preserves spatial locality, balances partition sizes, and supports scaling. We use a `max_rows_per_partition` parameter to ensure output files remain within optimal size and row count limits.

13.1. Categorization of Partitioning Methods

The spatial partitioning methods under evaluation can be categorized into three conceptual groups based on their underlying geometric and algorithmic principles.

13.1.1. Space-Filling Curves

These methods map multi-dimensional geographic coordinates into a single-dimensional order while preserving spatial locality. Though not partitioning methods per se, they enable highly efficient chunking after sorting.

- Representative: Hilbert Curve

13.1.2. Recursive Space Subdivision

These algorithms split the space recursively—either along axes (KD, KDB) or using a fixed grid structure (QuadTree). While primarily designed for planar space, they work well on projected data.

- Representatives: KD Tree, KDB Tree, QuadTree

13.1.3. Hierarchical Geospatial Tiling

These systems tile the globe using predefined cell structures—rectangular (GeoHash), spherical-quadrilateral (S2), or hexagonal (H3). The tiling is hierarchical and deterministic, making them ideal for parallel and distributed workflows.

- Representatives: GeoHash, S2, H3

13.2. Partitioning Method Benchmarks

With this categorization in mind, an implementation of each method was tested on the Swiss dataset. The tests were performed with consistent partition size limits and their performance was measured.

Dataset:

- Original `.pbf` Filesize: ca. 500MB
- Places count: ca. 500K
- Places conversion: ca. 3m
- `max_rows_per_partition = 100'000`

The following metrics are recorded:

- **Setup:** Environment initialization and dependency loading
- **Reading and Transformation:** Reading from PostgresSQL and converting to internal structures
- **Partitioning:** Applying the spatial partitioning algorithm
- **Writing:** Exporting the partitions as GeoParquet

- **Total Time Reading to Writing:** The total time it took from reading the partitions from PostgreSQL to writing the partitions as GeoParquet
- **Total Time Script:** The total time it took to run the Python command, measured in the entry point bash script.

13.2.1. Hilbert Curve Partitioning

Mechanism: Maps 2D coordinates to a 1D Hilbert index, then sorts the dataset by this index and partitions using the `max_rows_per_partition` variable.

Implementation: Using DuckDB's native Hilbert Curve partitioning with the spatial extension.

Benchmarks:

Measurement	1	2	3	4	5	Average
Setup	1.84	1.73	1.94	1.62	1.68	1.762
Reading and transformation	2.66	2.71	2.63	2.77	2.74	2.702
Partitioning	0	0	0	0	0	0
Writing to GeoParquet	2.76	2.81	2.77	2.81	2.79	2.788
Total time reading to writing	5.42	5.52	5.4	5.58	5.53	5.49
Total time script	8	7	7	8	7	7.4

Table 8: DuckDB Hilbert Curve Partitioning Benchmarks (measured in seconds)

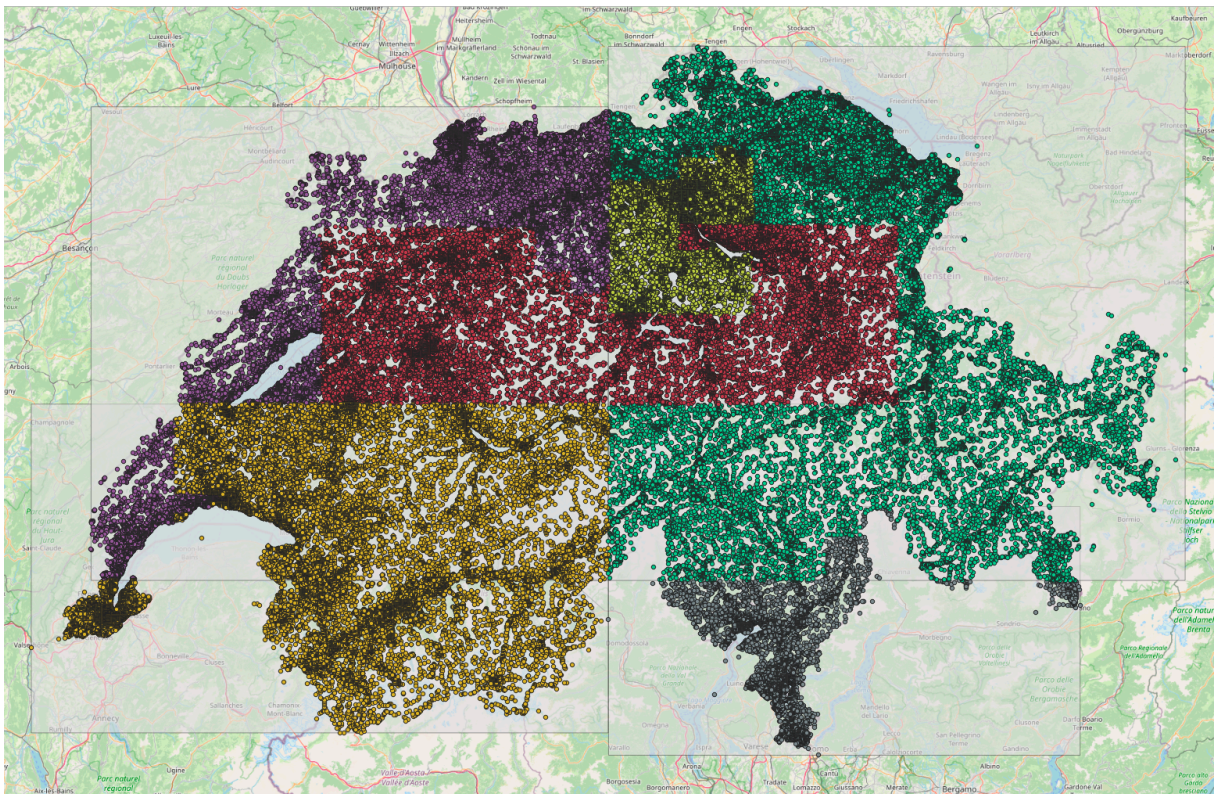


Figure 15: DuckDB Hilbert Curve partitions

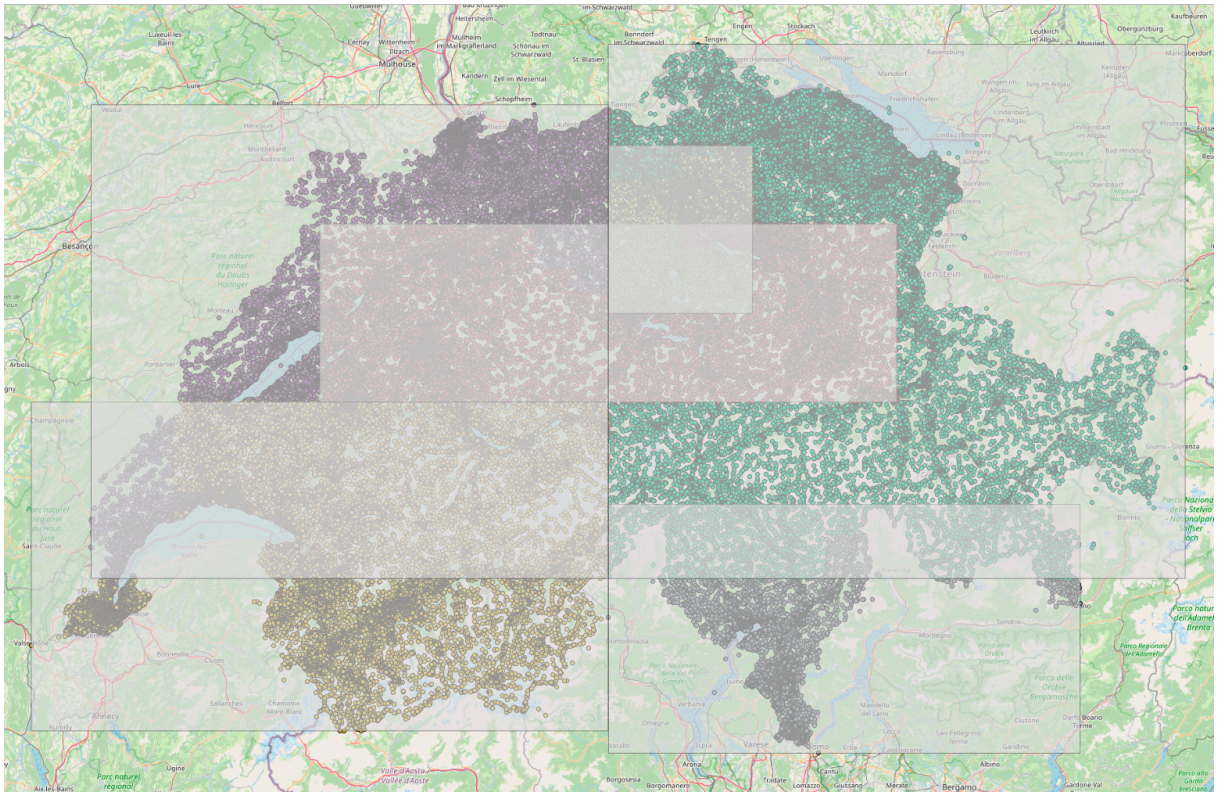


Figure 16: DuckDB Hilbert Curve partition BBoxes

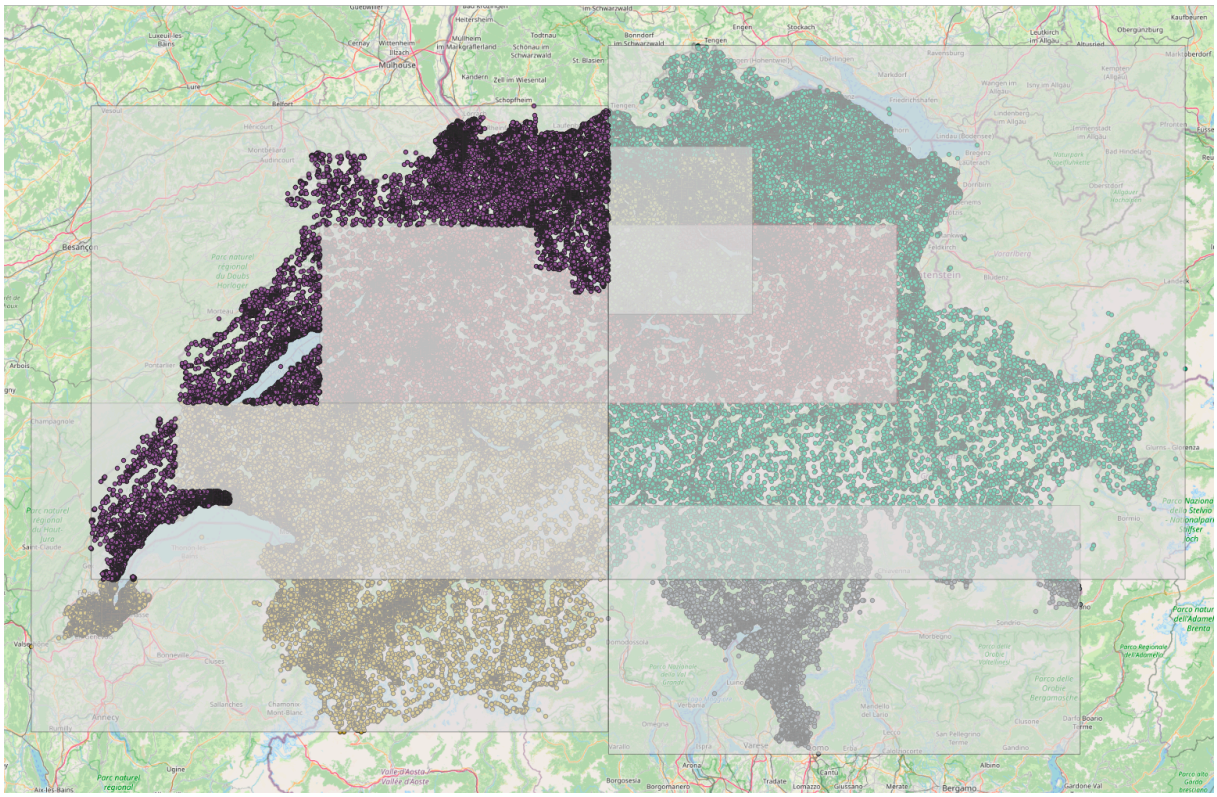


Figure 17: DuckDB Hilbert Curve partition skewed shape

Results:

- Partition shape / clustering: Elongated, snake-like partitions that follow the Hilbert curve's space-filling path.
- Bounding box overlap: Significant, because the partitions are of irregular shape.

- Amount of rows per partition: Perfect, exact amount of rows per partition, except for the last partition which contains the remaining rows.

13.2.2. KD Tree Partitioning

Mechanism: Recursively splits the dataset by alternating axis (longitude/latitude), yielding axis-aligned rectangles.

Implementation: Using DuckDB queries to calculate the median of the dataset and split it into two partitions. This is repeated recursively until the defined number of splits is met.

Benchmarks:

Measurement	1	2	3	4	5	Average
Setup	1.32	1.27	1.29	1.28	1.26	1.284
Reading and transformation	1.83	1.8	1.83	1.88	1.73	1.814
Partitioning	0.27	0.27	0.27	0.26	0.27	0.268
Writing to GeoParquet	1.11	1.1	1.1	1.14	1.08	1.106
Total time reading to writing	3.21	3.17	3.2	3.28	3.08	3.188
Total time script	5	5	5	5	5	5

Table 9: DuckDB KD Tree Partitioning Benchmarks (measured in seconds)

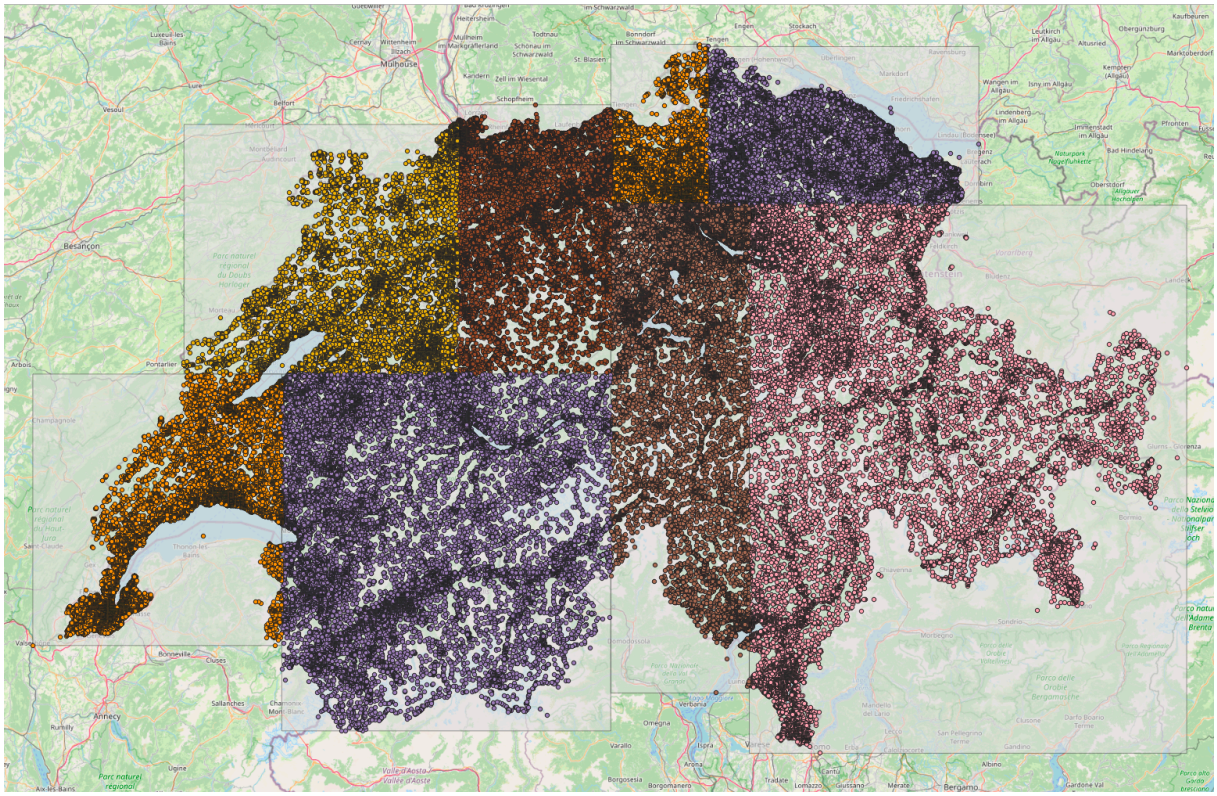


Figure 18: DuckDB KD Tree partitions

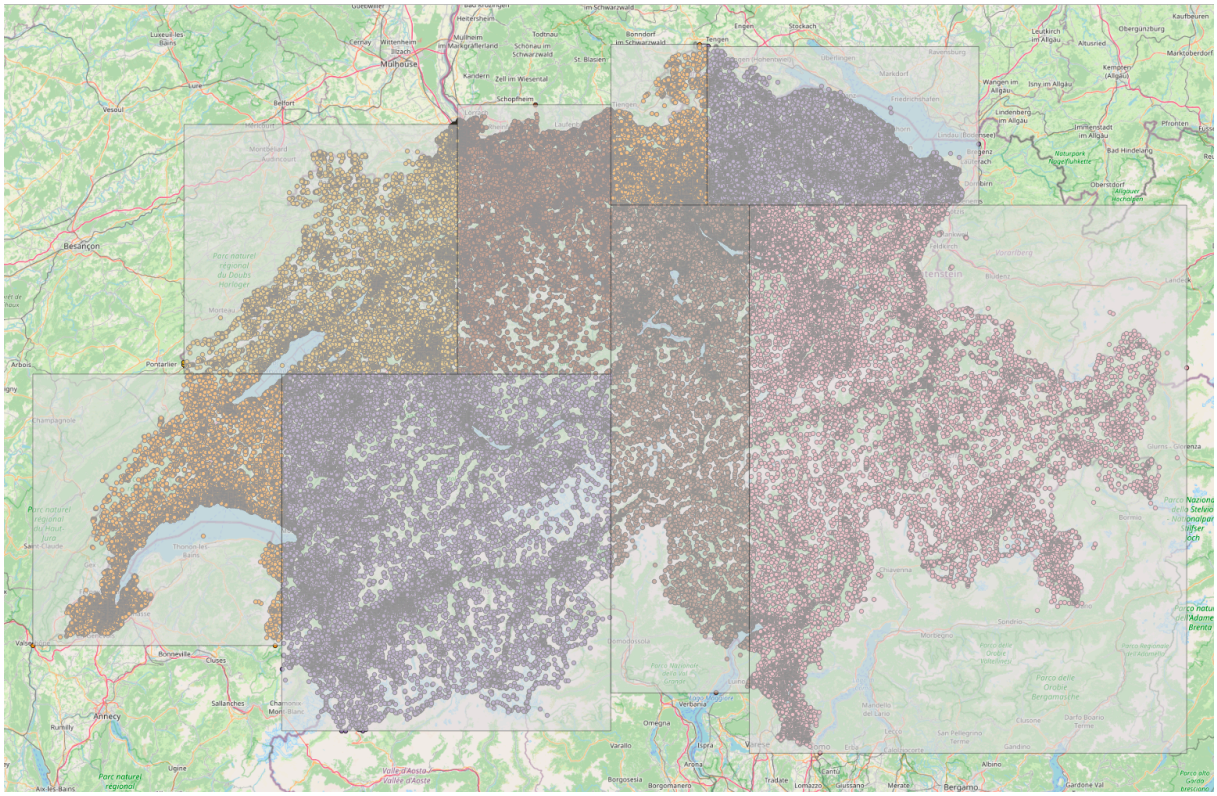


Figure 19: DuckDB KD Tree partition BBoxes

Results:

- Partition shape / clustering: Clean, rectangular partitions.
- Bounding box overlap: None, because the partitions are axis-aligned.
- Amount of rows per partition: Great, almost the same amount of rows in each partition.

13.2.3. KDB Tree Partitioning

Mechanism: Data-aware axis-aligned binary partitioning. Splits the largest partitions until row limits are met.

Implementation DuckDB: Using DuckDB queries to calculate the median of the dataset and split it into two partitions. This is repeated recursively until the number of rows per partition is below the defined limit.

Benchmarks:

Measurement	1	2	3	4	5	Average
Setup	1.23	1.26	1.2	1.25	1.29	1.246
Reading and transformation	1.81	1.84	1.85	1.81	1.78	1.818
Partitioning	0.39	0.39	0.38	0.38	0.39	0.386
Writing to GeoParquet	1.32	1.34	1.27	1.37	1.27	1.314
Total time reading to writing	3.52	3.57	3.5	3.56	3.44	3.518
Total time script	5	5	5	5	5	5

Table 10: DuckDB KDB Tree Partitioning Benchmarks (measured in seconds)

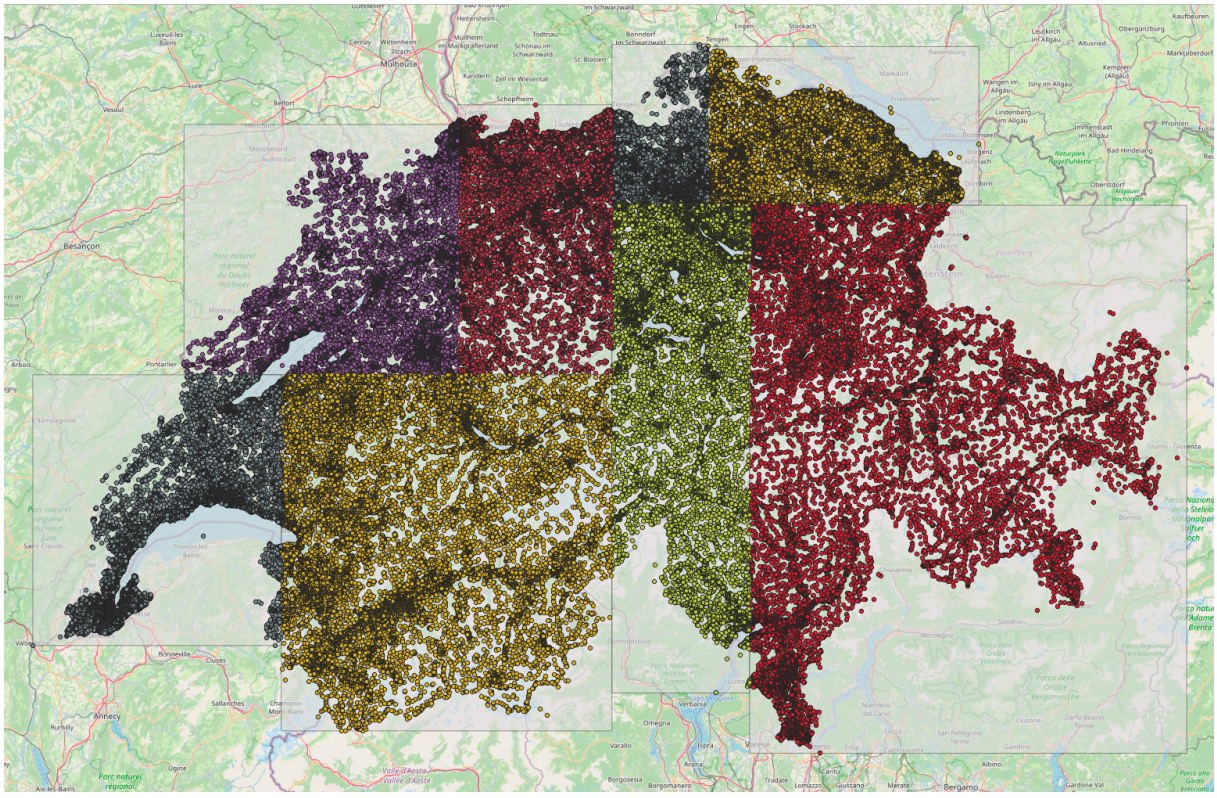


Figure 20: DuckDB KDB Tree partitions

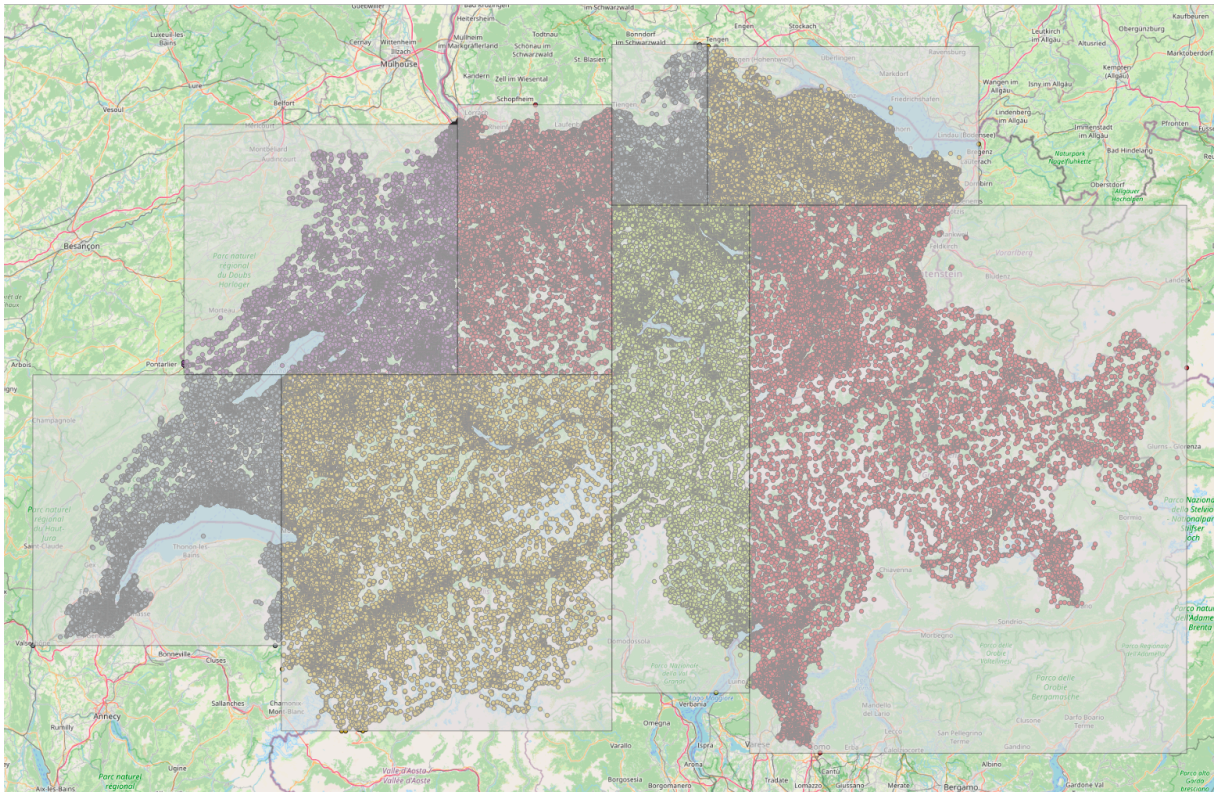


Figure 21: DuckDB KDB Tree partition BBoxes

Implementation Sedona: Using Sedona's native KDB Tree partitioning.

Benchmarks:

Measurement	1	2	3	4	5	Average
Setup	28.5	27.39	26.32	25.92	26.72	26.97
Reading and transformation	0.29	0.29	0.29	0.3	0.29	0.292
Partitioning	10.04	10.06	10.05	10.94	10.12	10.242
Writing to GeoParquet	3.1	2.98	3.06	2.93	2.88	2.99
Total time reading to writing	13.43	13.33	13.4	14.17	13.29	13.524
Total time script	43	41	41	41	41	41.4

Table 11: Sedona/Spark KDB Tree Partitioning Benchmarks (measured in seconds)

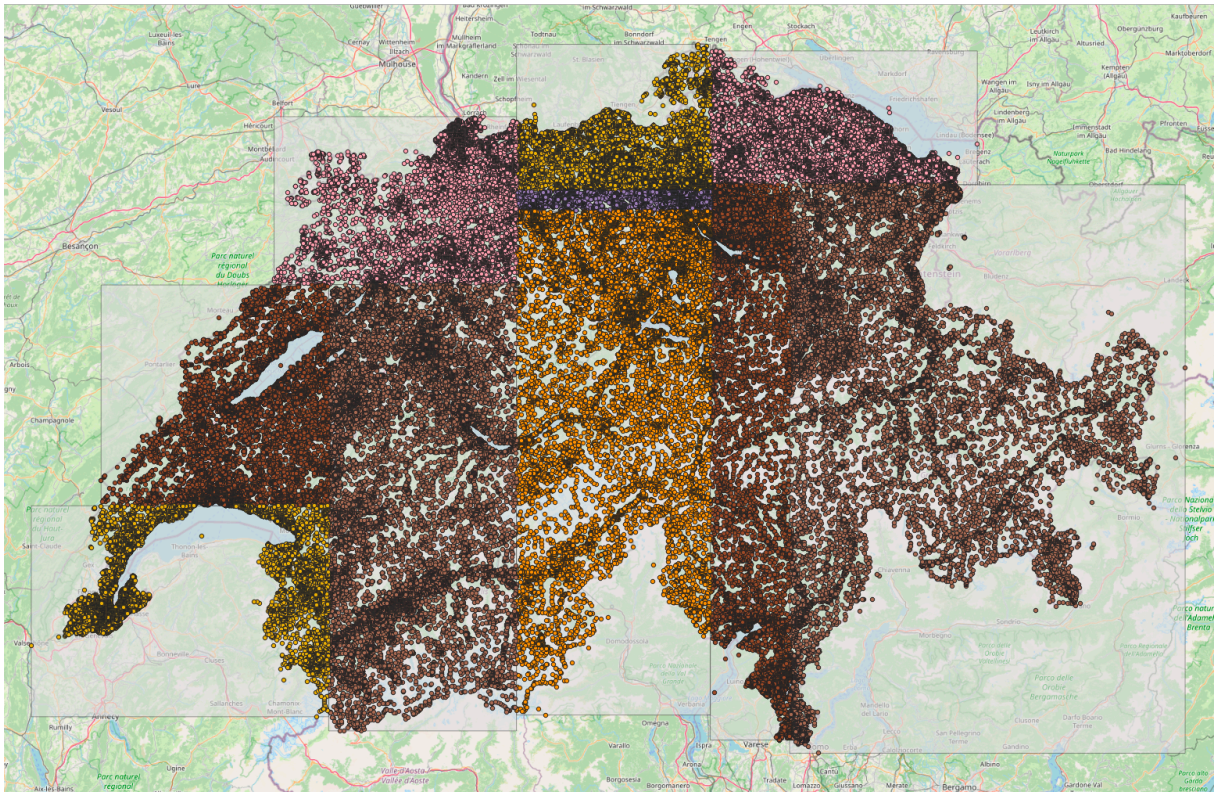


Figure 22: Sedona/Spark K-D-B Tree partitions

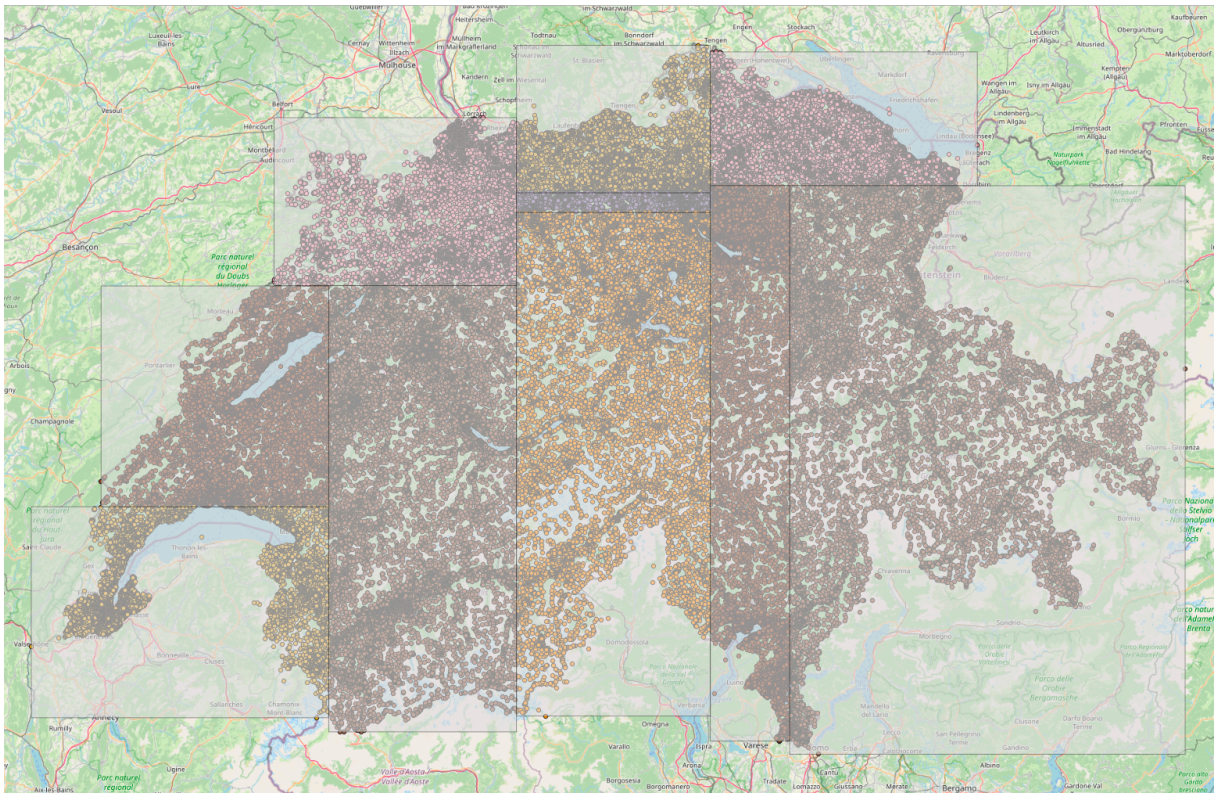


Figure 23: Sedona/Spark K-D-B Tree partition BBoxes

Results:

- Partition shape / clustering: Clean, rectangular partitions.
- Bounding box overlap: None, because the partitions are axis-aligned.

- Amount of rows per partition: Great, almost the same amount of rows in each partition. The DuckDB implementation balances better.

13.2.4. QuadTree Partitioning

Mechanism: Recursively divides space into quadrants until partitions are below the point limit.

Implementation: Using Sedona's native QuadTree partitioning.

Benchmarks:

Measurement	1	2	3	4	5	Average
Setup	25.68	26.67	26.6	25.57	27.57	26.418
Reading and transformation	0.3	0.3	0.3	0.3	0.29	0.298
Partitioning	10.17	10.35	10.62	11.22	10.52	10.576
Writing to GeoParquet	3.07	3.05	3.05	3.12	3.09	3.076
Total time reading to writing	13.54	13.7	13.97	14.64	13.9	13.95
Total time script	40	41	41	41	42	41

Table 12: Sedona/Spark QuadTree Partitioning Benchmarks (measured in seconds)

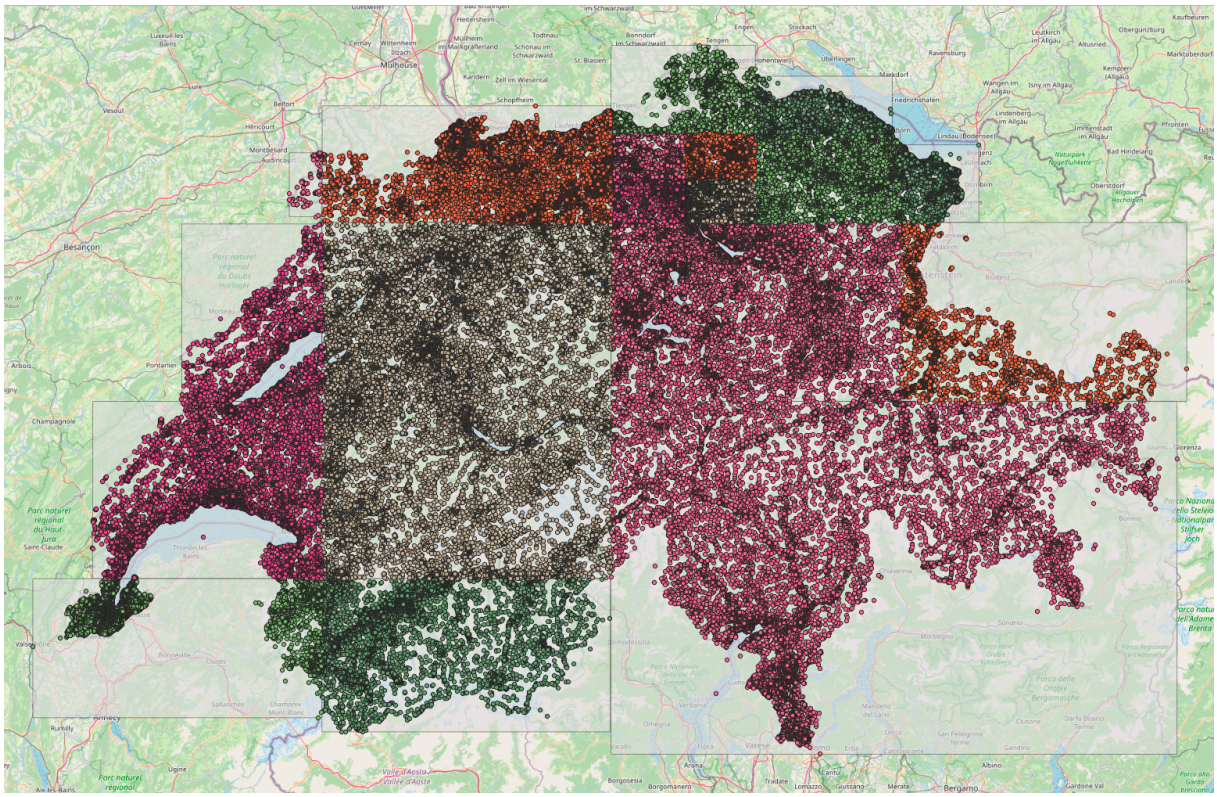


Figure 24: Sedona/Spark QuadTree partitions

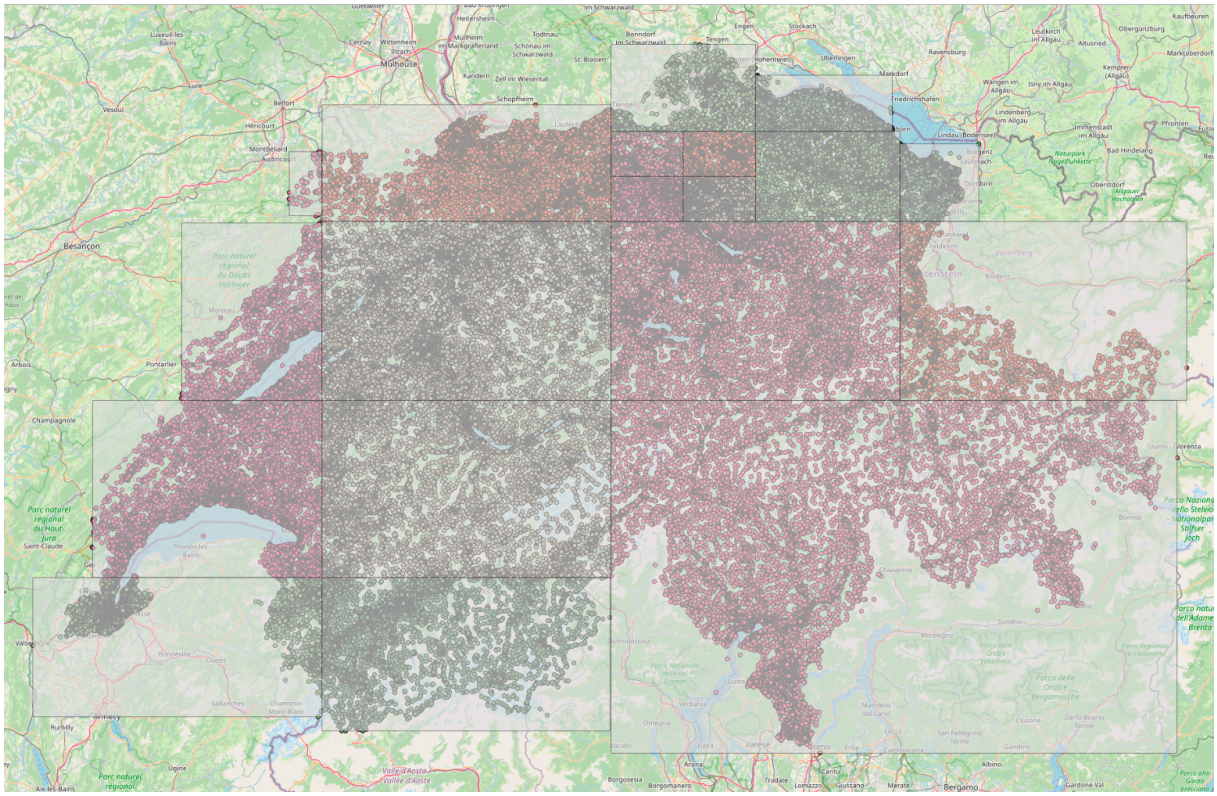


Figure 25: Sedona/Spark QuadTree partition BBoxes

Results:

- Partition shape / clustering: Clean, rectangular partitions ideal for BBOX queries.
- Bounding box overlap: None, because the partitions are axis-aligned.
- Amount of rows per partition: Okay, some variation from file to file.

13.2.5. GeoHash Partitioning

Mechanism: Converts coordinates into geohash strings, optionally varying the geohash length to fit target partition size.

Implementation: Using Sedona's native GeoHash partitioning.

Benchmarks:

Measurement	1	2	3	4	5	Average
Setup	25.55	25.56	26.69	25.17	27.18	26.03
Reading and transformation	0.3	0.3	0.3	0.3	0.3	0.3
Partitioning	9.5	9.91	9.65	9.99	9.65	9.74
Writing to GeoParquet	3.54	3.43	3.31	3.39	3.32	3.398
Total time reading to writing	13.34	13.64	13.26	13.68	13.27	13.438
Total time script	39	40	41	40	41	40.2

Table 13: Sedona/Spark GeoHash Partitioning Benchmarks (measured in seconds)

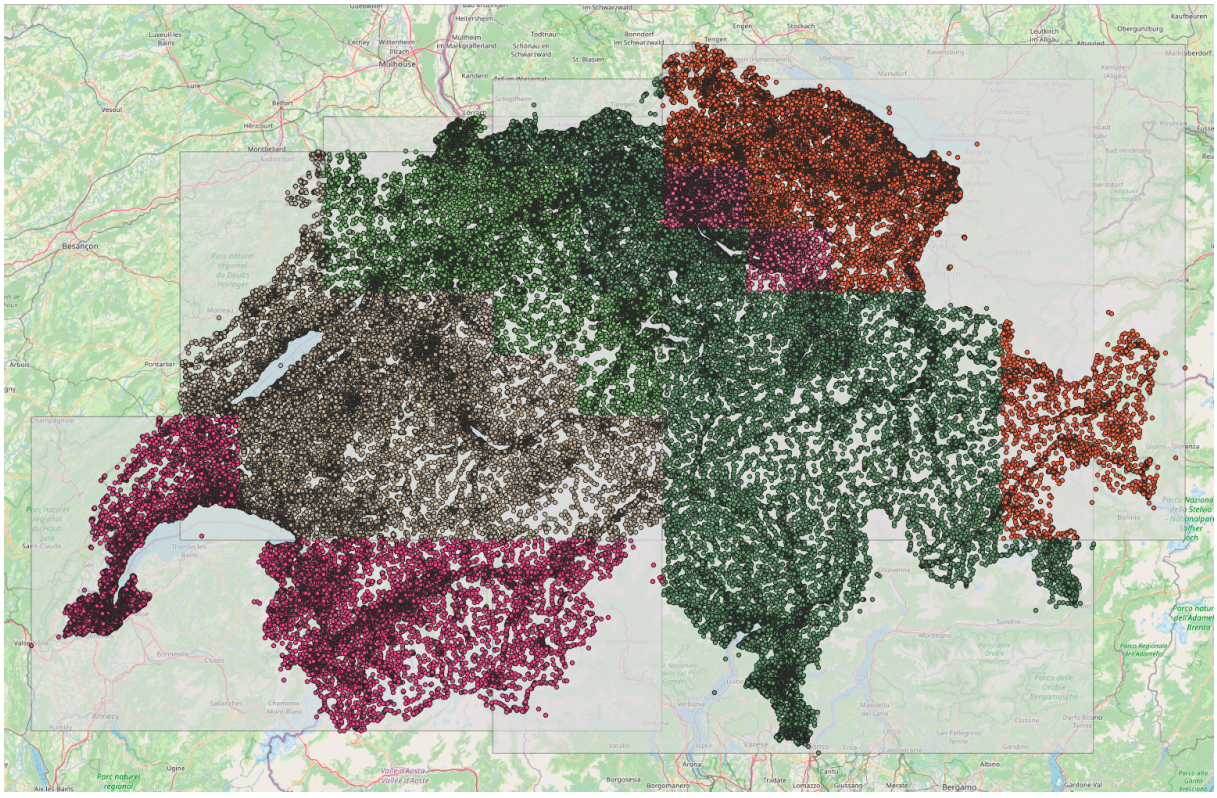


Figure 26: Sedona/Spark GeoHash partitions

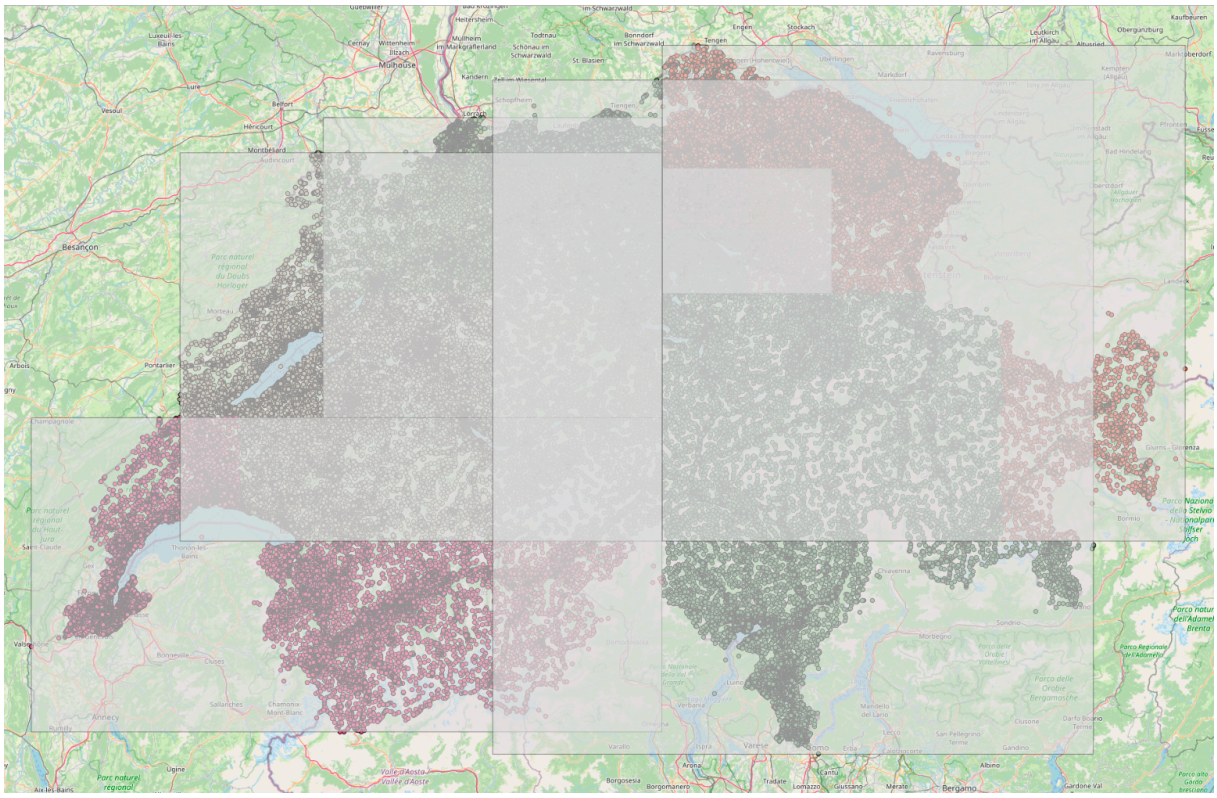


Figure 27: Sedona/Spark GeoHash partition BBoxes

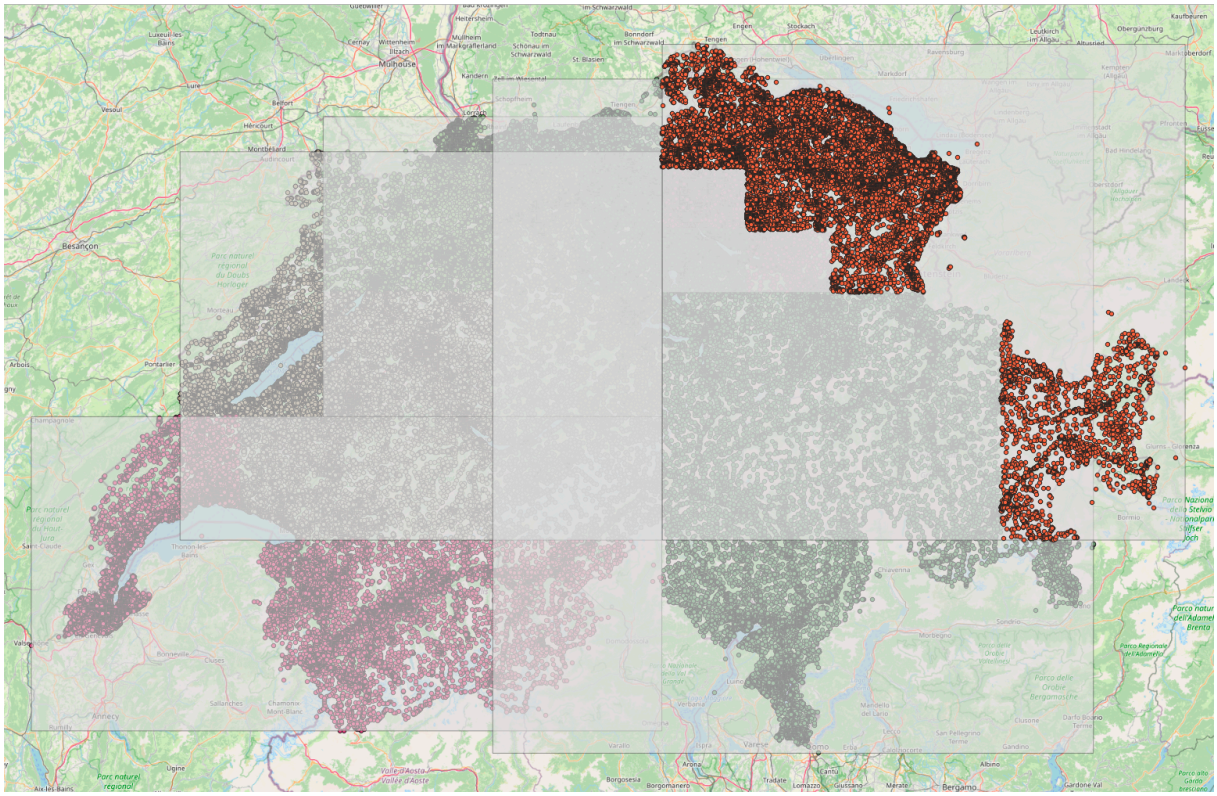


Figure 28: Sedona/Spark GeoHash fragmented partition

Results:

- Partition shape / clustering: Some geohash rectangles get merged. Since they're in Z-Order, they get merged diagonally.
- Bounding box overlap: Significant, large bounding boxes because of the diagonal merge.
- Amount of rows per partition: Good, some variation, but not by much.

13.2.6. S2 Partitioning

Mechanism: Assigns points to S2 cells and refines cell level until each is under the point limit. Uses spherical geometry and Hilbert curves internally to sort partitions, developed and used by Google Maps.

Implementation: Using DuckDB's community extension `s2geometry`, the adaptive spatial partitioning recursively groups points into S2 cells, starting from the maximum level (most granular) and aggregating to coarser levels until each partition contains fewer than the specified maximum points or reaches the minimum level.

Benchmarks:

Measurement	1	2	3	4	5	Average
Setup	1.7	1.71	1.67	1.74	1.8	1.724
Reading and transformation	2.1	1.83	1.78	1.77	1.83	1.862
Partitioning	0.02	0.02	0.02	0.02	0.02	0.02
Writing to GeoParquet	1.83	1.83	1.83	1.82	1.83	1.828
Total time reading to writing	3.95	3.68	3.63	3.61	3.68	3.71
Total time script	5	6	6	6	5	5.6

Table 14: DuckDB S2 Partitioning Benchmarks (measured in seconds)

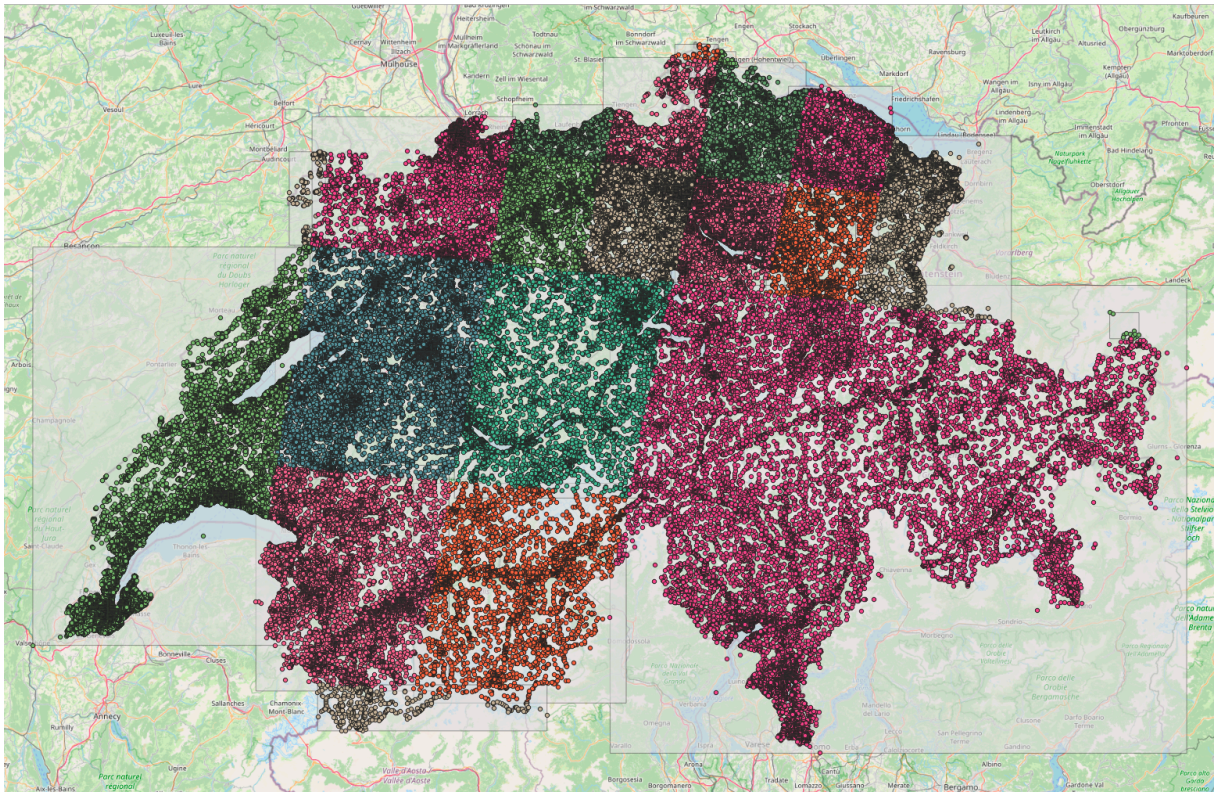


Figure 29: DuckDB S2 partitions

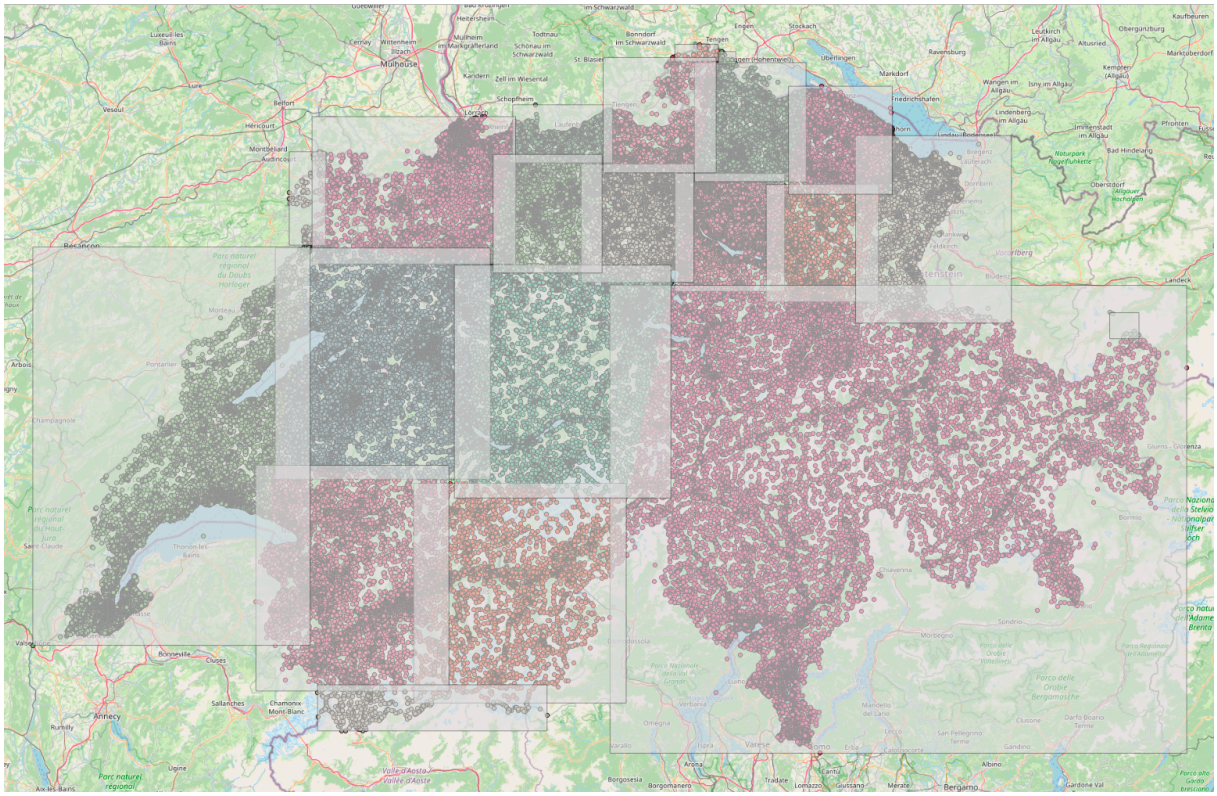


Figure 30: DuckDB S2 partition BBoxes

Results:

- Partition shape / clustering: Great, partition shapes are spherical geodesic.
- Bounding box overlap: Some overlap, because of the spherical geometry.
- Amount of rows per partition: Not so balanced, has some very small partitions, likely because of the shape of the country borders.

13.2.7. H3 Partitioning

Mechanism: Hexagon-based grid system developed by Uber. Assigns points to H3 cells at a fixed resolution. Optionally, adaptively subdivide overloaded cells.

Implementation: Using PostgreSQL's h3 extension to compute adaptive hexagon cells, starting from a fine resolution and aggregating to coarser resolutions until each cell contains fewer than the specified maximum points.

Benchmarks:

Measurement	1	2	3	4	5	Average
Setup	1.24	1.24	1.2	1.17	1.2	1.21
Reading and transformation	2.92	3.52	2.45	2.3	2.41	2.72
Partitioning	32.2	33.31	31.55	34	32.63	32.738
Writing to GeoParquet	1.77	1.78	1.75	1.74	1.72	1.752
Total time reading to writing	36.89	38.61	35.75	38.04	36.76	37.21
Total time script	38	40	38	40	39	39

Table 15: PostgreSQL/DuckDB H3 Partitioning Benchmarks (measured in seconds)

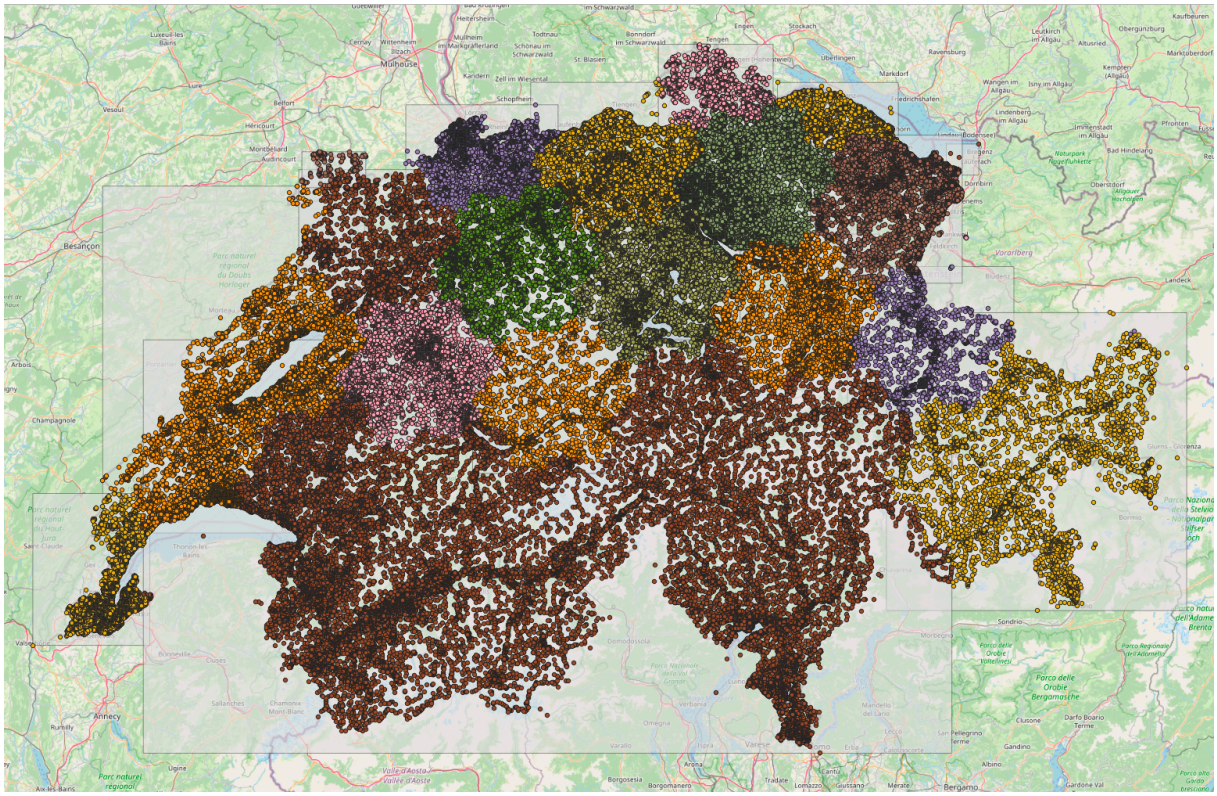


Figure 31: PostgreSQL/DuckDB H3 partitions

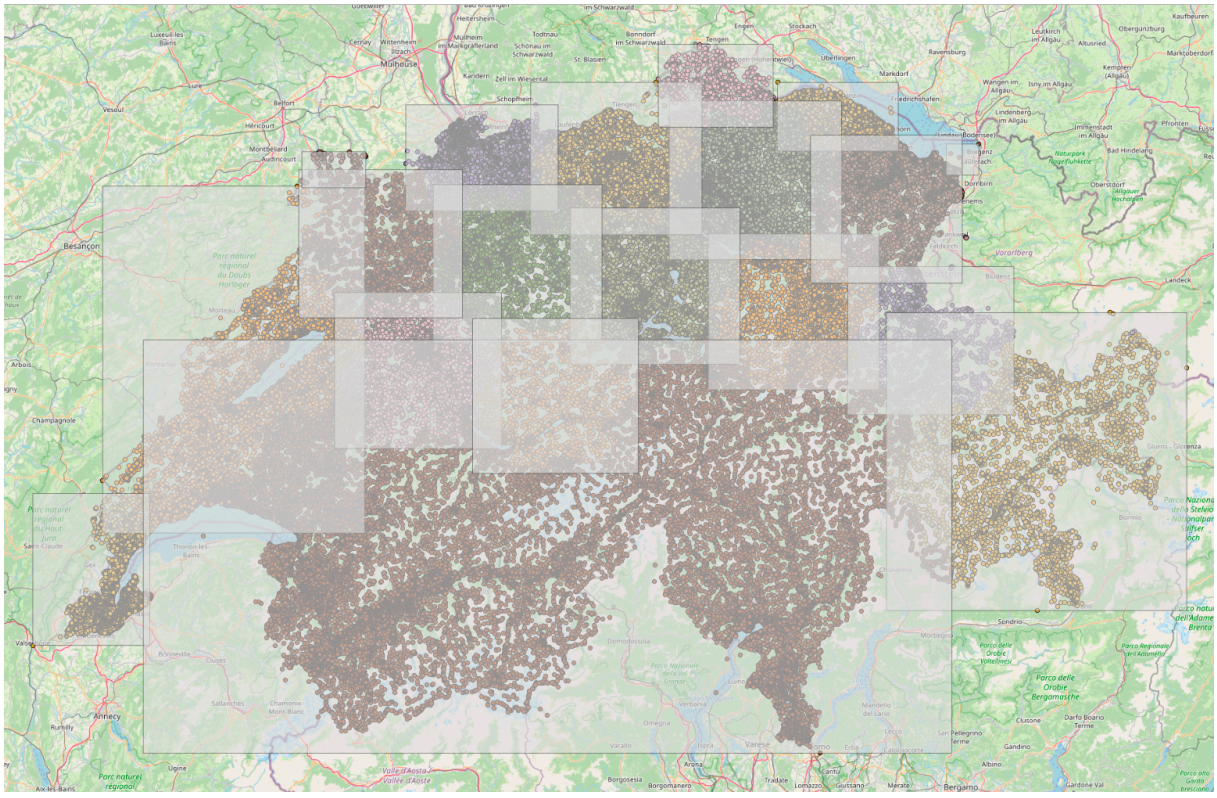


Figure 32: PostgreSQL/DuckDB H3 partition BBoxes

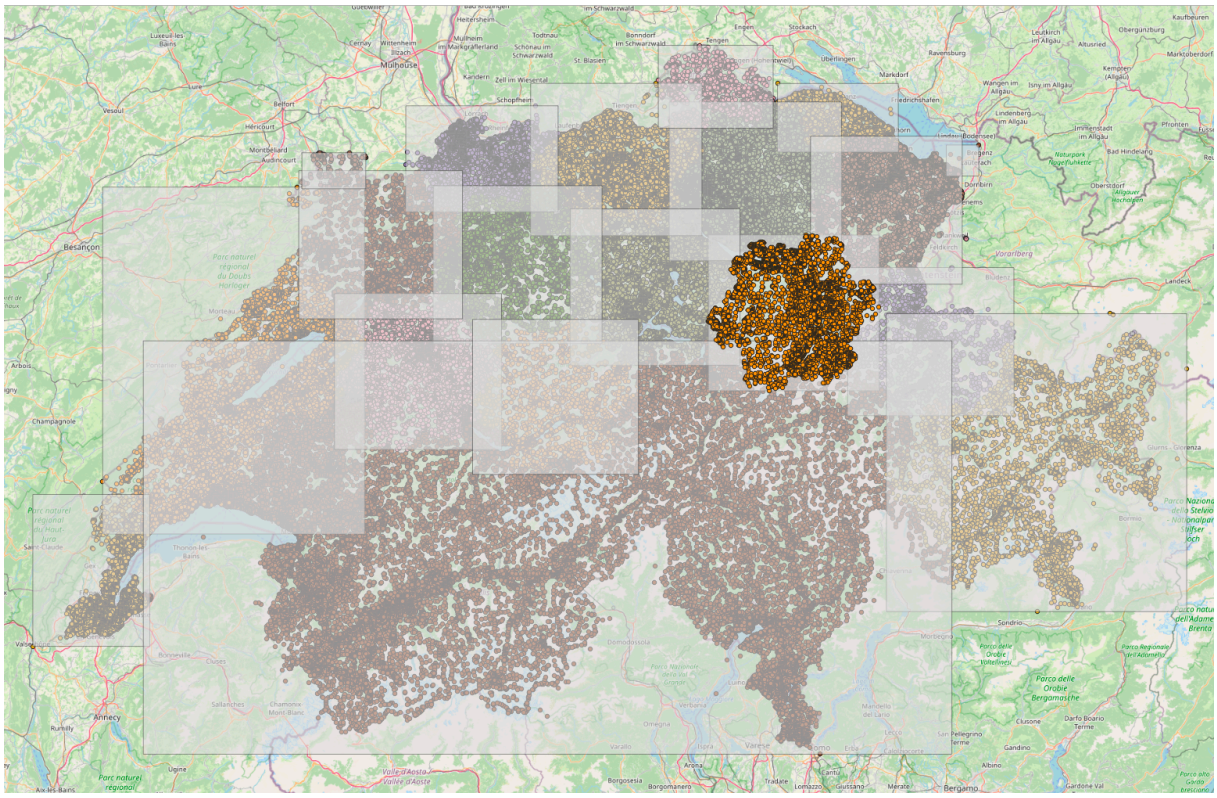


Figure 33: PostgreSQL/DuckDB H3 recognizable hexagon shape

Results:

- Partition shape / clustering: Great, partition shapes are hexagonal.
- Bounding box overlap: Some overlap, because of the hexagonal geometry.

- Amount of rows per partition: Not so balanced, has some very small partitions, likely because of the shape of the country borders.

13.2.8. Summary and Observations

Algorithm	Setup	Reading and Trans-formation	Partitioning	Writing	Total Time Reading to Writing	Total Time Script
DuckDB Hilbert Curve	1.762	2.702	0	2.788	5.49	7.4
Sedona GeoHash	26.03	0.3	9.74	3.398	13.438	40.2
Sedona KDB Tree	26.97	0.292	10.242	2.99	13.524	41.4
DuckDB KDB Tree	1.246	1.818	0.386	1.314	3.518	5
DuckDB KD Tree	1.284	1.814	0.268	1.106	3.188	5
Sedona QuadTree	26.418	0.298	10.576	3.076	13.95	41
DuckDB S2	1.724	1.862	0.02	1.828	3.71	5.6
PostgreSQL/DuckDB H3	1.21	2.72	32.738	1.752	37.21	39

Table 16: Partitioning Algorithm Comparison (measured in seconds)

Based on our evaluation, we made the following observations and decisions:
- Hilbert: Elongated, snake-like partitions that follow the Hilbert curve's space-filling path make it unsuitable for partitioning.
- KDB Tree (Sedona/DuckDB): DuckDB resulted in more balanced partitions, while partitions in sedona were more elongated. Sedona KDB Tree will be eliminated in favor of DuckDB KDB Tree.
- DuckDB KDB and KD Tree: The resulting partitions were approximately the same. The KDB Tree has the advantage of setting partition sizes, which is a clear advantage for this use case and eliminates the KD Tree.
- GeoHash: Sedona GeoHash partitions are non-rectangular because some areas get merged diagonally, which makes it unsuitable for partitioning.
- QuadTree, S2, H3: Generally partition well, however, they can produce partitions with only few points due to the border shape of the country.

This narrows down the selection to the following partitioning methods for further analysis:
- KDB Tree
- QuadTree
- S2
- H3

13.3. Scaling Experiments

To evaluate the scalability of the partitioning methods, the four preferred algorithms are tested using larger countries (Germany and the United States) with a partition size of 500,000 points per file.

13.3.1. Germany

Dataset:

- Original `.pbf` Filesize: ca. 4.52 GB
- Places count: ca. 4.5M
- Places conversion: ca. 31m
- `max_rows_per_partition = 500'000`

	KDB-Tree	QuadTree	S2	H3
Setup	1.4	27.09	1.76	1.27
Reading and transformation	15	0.35	14.96	29.87
Partitioning	7.87	94.03	0.07	372.54
Writing to GeoParquet	6.06	17.63	8.02	7.04
Total time reading to writing	28.93	112.01	23.05	409.45
Total time script	30	140	25	411

Table 17: Germany Partitioning Algorithm Comparison (measured in seconds)

13.3.2. USA

Dataset:

- Original `.pbf` Filesize: ca. 11.1 GB
- Places count: ca. 7.96M
- Places conversion: ca. 1h 9m
- `max_rows_per_partition = 500'000`

	KDB-Tree	QuadTree	S2	H3
Setup	1.21	26.39	1.75	1.26
Reading and transformation	24.07	0.35	23.33	74.26
Partitioning	14.02	151.86	0.12	691.91
Writing to GeoParquet	9.1	27.19	16.23	16.37
Total time reading to writing	47.19	179.4	39.68	782.54
Total time script	49	207	42	784

Table 18: USA Partitioning Algorithm Comparison (measured in seconds)

The results in Figure 34 show that DuckDB-based methods (KDB Tree and S2) consistently outperform alternatives:

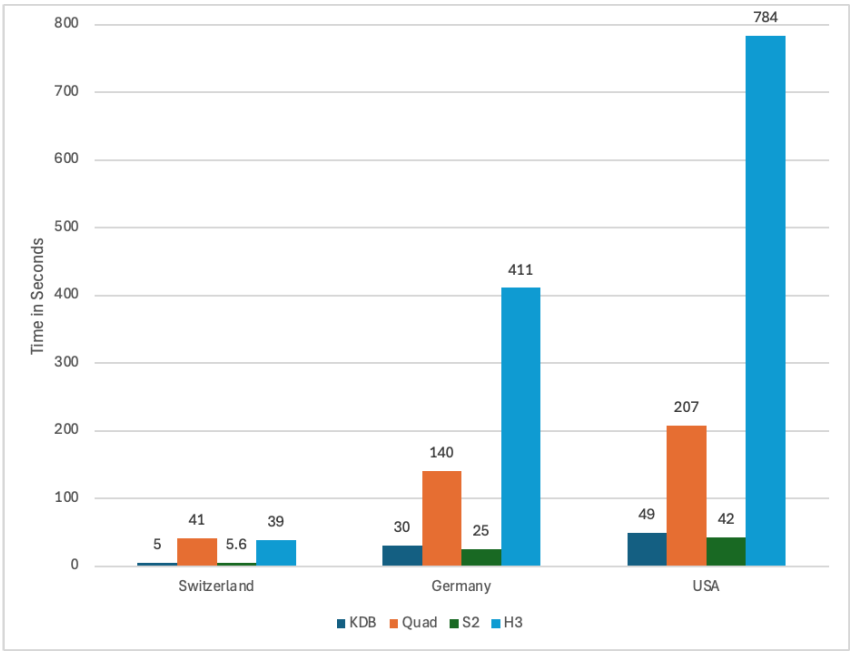


Figure 34: Partitioning Algorithm Comparison

13.4. Query Testing on Germany

To further investigate the performance of the partitioning methods, we execute queries on the resulting GeoParquet files using DuckDB. We measure the query execution time using the `EXPLAIN ANALYZE` command to measure the time.

The queries used for this test are as follows:

Query “sushi_restaurant” (for BBox see Figure 35)

```
SELECT names, categories, websites, emails, phones, addresses
FROM read_parquet('{path}')
WHERE categories.primary = 'sushi_restaurant'
AND bbox.xmin BETWEEN 8.000000 AND 14.000000
AND bbox.ymin BETWEEN 51.000000 AND 53.000000;
```

Figure 35: Sushi Restaurant Query BBox

Query “restaurant” (for BBox see Figure 36)

```
SELECT names, categories, websites, emails, phones, addresses
FROM read_parquet('{path}')
WHERE categories.primary = 'restaurant'
OR array_contains(categories.alternate, 'restaurant')
AND bbox.xmin BETWEEN 9.500000 AND 10.000000
AND bbox.ymin BETWEEN 47.600000 AND 54.000000
```

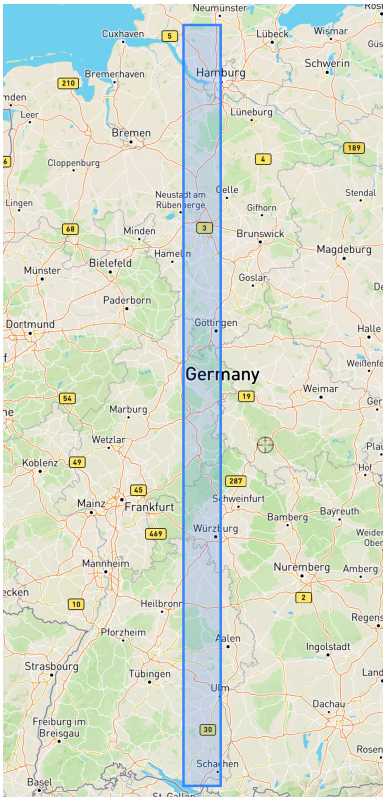


Figure 36: Restaurant Query BBox

Query “warehouse” (for BBox see Figure 37)

```sql
SELECT geometry, bbox, names, categories, websites, emails, phones, addresses
FROM read_parquet('{path}')
WHERE categories.primary = 'warehouse'
OR array_contains(categories.alternate, 'warehouse')
AND bbox.xmin BETWEEN 8.000000 AND 12.000000
AND bbox.ymin BETWEEN 48.000000 AND 54.000000
```

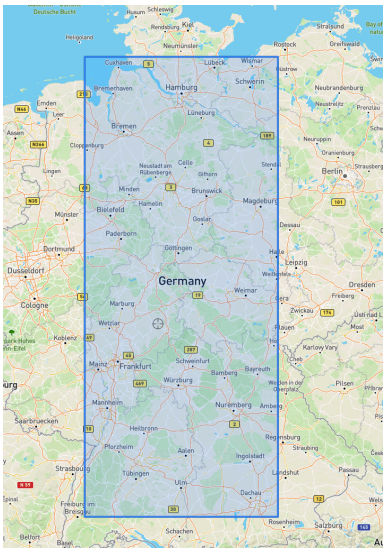


Figure 37: Warehouse Query BBox

Query Results:

Algorithm	Query	Run 1	Run 2	Run 3	Run 4	Run 5	Average
KDB-Tree	restaurant	0.231	0.227	0.224	0.224	0.226	0.226
KDB-Tree	sushi_restaurant	0.115	0.0814	0.0825	0.085	0.082	0.089
KDB-Tree	warehouse	0.279	0.28	0.28	0.252	0.252	0.269
S2	restaurant	0.24	0.228	0.241	0.258	0.247	0.243
S2	sushi_restaurant	0.0775	0.0765	0.0793	0.0752	0.078	0.077
S2	warehouse	0.255	0.255	0.279	0.258	0.276	0.265
H3	restaurant	0.246	0.266	0.252	0.241	0.247	0.25
H3	sushi_restaurant	0.101	0.0642	0.0672	0.067	0.0674	0.073
H3	warehouse	0.263	0.263	0.264	0.286	0.268	0.269
QuadTree	restaurant	0.253	0.244	0.251	0.254	0.243	0.249
QuadTree	sushi_restaurant	0.116	0.115	0.115	0.124	0.113	0.117
QuadTree	warehouse	0.271	0.271	0.296	0.297	0.276	0.282

Table 19: Query Results (measured in seconds)

Query Averages:

Query	H3	KDB-Tree	QuadTree	S2
restaurant	0.25	0.226	0.249	0.243
sushi_restaurant	0.073	0.089	0.117	0.077
warehouse	0.269	0.269	0.282	0.265

Table 20: Query Averages (measured in seconds)

The results from Table 19 and Table 20 demonstrate that the partitioning method has negligible impact on spatial query performance.

13.5. Conclusion

Both S2 and KDB Tree demonstrate excellent performance in our tests. S2 excels in global-scale scenarios with its clean spatial boundaries, adaptive resolution, and disjoint partitions, which enable efficient parallel processing. The true power of these tiling systems lies in their ability to partition different world regions independently without coordination, yet still produce consistent, interoperable results.

While S2 is ideal for worldwide datasets, KDB Tree proves more than sufficient for country-level datasets, offering comparable performance for regional processing.

14. NFR 2 - Performance and Scalability Analysis

14.1. Dataset Comparison

The scalability of the pipeline was tested by comparing processing times between two datasets:
1. D-A-CH-LI (Germany, Austria, Switzerland, Liechtenstein)
2. D-A-CH-LI-FR-IT (Germany, Austria, Switzerland, Liechtenstein, France, Italy)

Country	File Size (MB)
Germany (DE)	4′328
Austria (AT)	737
Switzerland (CH)	474
Liechtenstein (LI)	3
Total	**5′542**

Table 21: D-A-CH-LI Dataset Size

Country	File Size (MB)
Germany (DE)	4′328
Austria (AT)	737
Switzerland (CH)	474
Liechtenstein (LI)	3
France (FR)	4′569
Italy (IT)	1′970
Total	**12′081**

Table 22: Extended Dataset Size

The scalability factor can be calculated by comparing the total dataset sizes from Table 21 and Table 22:

$$\text{Dataset Size Ratio} = \frac{12081 \text{ MB}}{5542 \text{ MB}} \approx 2.2$$

The extended dataset, which adds France and Italy, represents a 2.2-fold increase in size compared to the original D-A-CH-LI dataset.

14.2. Processing Time Analysis

14.2.1. Initial Benchmark

Run	Processing Time (min)
1	74
2	75
3	74
Average	**74**

Table 23: Processing Time for D-A-CH-LI-FR-IT Dataset, rounded to minutes

The experimental results in Table 23 demonstrate the effectiveness of the parallel processing implementation. Despite the 2.2-fold increase in dataset size, the processing time for the extended D-A-CH-LI-FR-IT dataset increased by only 10 minutes compared to the original D-A-CH-LI dataset (as documented in Table 5, see page 52).

14.2.2. Detailed Performance Metrics

In order to prove scalability, we have to look at the processing time of the countries in Table 24 and Table 25.

Country	Run 1	Run 2	Run 3	Average Time
DE	58 minutes 34 seconds	62 minutes 29 seconds	59 minutes 8 seconds	60 minutes 4 seconds
AT	9 minutes 41 seconds	9 minutes 59 seconds	10 minutes 1 seconds	9 minutes 54 seconds
CH	7 minutes 2 seconds	6 minutes 53 seconds	7 minutes 14 seconds	7 minutes 3 seconds
LI	1 minutes 18 seconds	1 minutes 26 seconds	2 minutes 3 seconds	1 minute 36 seconds
Sum	76 minutes 35 seconds	80 minutes 47 seconds	78 minutes 26 seconds	78 minutes 36 seconds

Table 24: Processing Time for D-A-CH-LI Dataset

Country	Run 1	Run 2	Run 3	Average Time
DE	58 minutes 48 seconds	58 minutes 36 seconds	60 minutes 47 seconds	59 minutes 24 seconds
AT	8 minutes 52 seconds	9 minutes 42 seconds	9 minutes 42 seconds	9 minutes 25 seconds
CH	6 minutes 39 seconds	6 minutes 38 seconds	6 minutes 41 seconds	6 minutes 39 seconds
LI	1 minutes 32 seconds	1 minutes 55 seconds	2 minutes 48 seconds	2 minutes 5 seconds
FR	69 minutes 33 seconds	70 minutes 10 seconds	69 minutes 27 seconds	69 minutes 43 seconds
IT	26 minutes 46 seconds	25 minutes 59 seconds	27 minutes 24 seconds	26 minutes 43 seconds
Sum	172 minutes 10 seconds	173 minutes 0 seconds	176 minutes 49 seconds	174 minutes

Table 25: Processing Time for D-A-CH-LI-FR-IT Dataset

The scalability factor can be calculated by comparing the total processing times:

$$\text{Processing Time Ratio} = \frac{174 \text{ minutes}}{78 \text{ minutes and } 36 \text{ seconds}} = \frac{10440 \text{ seconds}}{4716 \text{ seconds}} \approx 2.2$$

The proportional increase in processing time relative to the dataset size confirms a linear scaling relationship.

14.3. Global Scalability Projection

In order to project the processing time for a global-scale dataset, the processing time per country is calculated in MB/minute.

Country	Size in MB	Processing Time rounded to minutes	Processing Time in MB/minute
DE	4328	59	73
AT	737	9	82
CH	474	7	68
FR	4569	70	65
IT	1970	27	73

Table 26: Processing Throughput by Country

To evaluate the system's scalability to a global dataset, a projection was calculated using the most conservative throughput measurement of 65 MB/minute observed for the France dataset (see Table 26). For a global dataset of 81 GB (81,000 MB), the estimated processing time is:

$$\text{Global Processing Time} = \frac{81{,}000 \text{ MB}}{65 \text{ MB/min}}! \approx 1,240 \text{ minutes} \approx 21 \text{ hours}$$

This projection indicates that the system can process a global-scale dataset in approximately 21 hours when operating at the lowest observed throughput rate.

15. Usage

What follows is usage information of the data provided by the pipeline.

15.1. Available Data

The data is available in two themes:

- `places`
- `divisions`

Each theme has one or multiple types:

- `places`
  - `place`
- `divisions`
  - `division`
  - `division_area`
  - `division_boundary`

15.2. File Schemas

The file schemas are as follows:

15.2.1. Places Theme

Place

```
root
 |-- id: string (nullable = true)
 |-- geometry: binary (nullable = true)
 |-- bbox: struct (nullable = true)
 |    |-- xmin: float (nullable = true)
 |    |-- xmax: float (nullable = true)
 |    |-- ymin: float (nullable = true)
 |    |-- ymax: float (nullable = true)
 |-- type: string (nullable = true)
 |-- version: integer (nullable = true)
 |-- sources: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- property: string (nullable = true)
 |    |    |-- dataset: string (nullable = true)
 |    |    |-- record_id: string (nullable = true)
 |    |    |-- update_time: string (nullable = true)
 |    |    |-- confidence: double (nullable = true)
 |-- names: struct (nullable = true)
 |    |-- primary: string (nullable = true)
 |    |-- common: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: string (valueContainsNull = true)
 |    |-- rules: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- variant: string (nullable = true)
 |    |    |    |-- language: string (nullable = true)
 |    |    |    |-- value: string (nullable = true)
 |    |    |    |-- between: array (nullable = true)
 |    |    |    |    |-- element: double (containsNull = true)
```

```
|    |    |    |-- side: string (nullable = true)
|-- categories: struct (nullable = true)
|    |-- primary: string (nullable = true)
|    |-- alternate: array (nullable = true)
|    |    |-- element: string (containsNull = true)
|-- confidence: double (nullable = true)
|-- websites: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- socials: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- emails: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- phones: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- brand: struct (nullable = true)
|    |-- wikidata: string (nullable = true)
|    |-- names: struct (nullable = true)
|    |    |-- primary: string (nullable = true)
|    |    |-- common: map (nullable = true)
|    |    |    |-- key: string
|    |    |    |-- value: string (valueContainsNull = true)
|    |    |-- rules: array (nullable = true)
|    |    |    |-- element: struct (containsNull = true)
|    |    |    |    |-- variant: string (nullable = true)
|    |    |    |    |-- language: string (nullable = true)
|    |    |    |    |-- value: string (nullable = true)
|    |    |    |    |-- between: array (nullable = true)
|    |    |    |    |    |-- element: double (containsNull = true)
|    |    |    |    |-- side: string (nullable = true)
|-- addresses: array (nullable = true)
|    |-- element: struct (containsNull = true)
|    |    |-- freeform: string (nullable = true)
|    |    |-- locality: string (nullable = true)
|    |    |-- postcode: string (nullable = true)
|    |    |-- region: string (nullable = true)
|    |    |-- country: string (nullable = true)
|-- theme: string (nullable = true)
|-- ext_tags: map (nullable = true)
|    |-- key: string
|    |-- value: string (valueContainsNull = true)
```

15.2.2. Divisions Theme
divisions/division

```
root
 |-- id: string (nullable = true)
 |-- geometry: binary (nullable = true)
 |-- bbox: struct (nullable = true)
 |    |-- xmin: float (nullable = true)
 |    |-- xmax: float (nullable = true)
 |    |-- ymin: float (nullable = true)
 |    |-- ymax: float (nullable = true)
 |-- country: string (nullable = true)
 |-- version: integer (nullable = true)
```

```
 |-- sources: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- property: string (nullable = true)
 |    |    |-- dataset: string (nullable = true)
 |    |    |-- record_id: string (nullable = true)
 |    |    |-- update_time: string (nullable = true)
 |    |    |-- confidence: double (nullable = true)
 |-- subtype: string (nullable = true)
 |-- class: string (nullable = true)
 |-- names: struct (nullable = true)
 |    |-- primary: string (nullable = true)
 |    |-- common: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: string (valueContainsNull = true)
 |    |-- rules: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- variant: string (nullable = true)
 |    |    |    |-- language: string (nullable = true)
 |    |    |    |-- value: string (nullable = true)
 |    |    |    |-- between: array (nullable = true)
 |    |    |    |    |-- element: double (containsNull = true)
 |    |    |    |-- side: string (nullable = true)
 |-- wikidata: string (nullable = true)
 |-- region: string (nullable = true)
 |-- perspectives: struct (nullable = true)
 |    |-- mode: string (nullable = true)
 |    |-- countries: array (nullable = true)
 |    |    |-- element: string (containsNull = true)
 |-- local_type: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
 |-- hierarchies: array (nullable = true)
 |    |-- element: array (containsNull = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- division_id: string (nullable = true)
 |    |    |    |-- subtype: string (nullable = true)
 |    |    |    |-- name: string (nullable = true)
 |-- parent_division_id: string (nullable = true)
 |-- norms: struct (nullable = true)
 |    |-- driving_side: string (nullable = true)
 |-- population: integer (nullable = true)
 |-- capital_division_ids: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- capital_of_divisions: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- division_id: string (nullable = true)
 |    |    |-- subtype: string (nullable = true)
 |-- theme: string (nullable = true)
 |-- type: string (nullable = true)
```

divisions/division_area

```
root
 |-- id: string (nullable = true)
```

```
 |-- geometry: binary (nullable = true)
 |-- bbox: struct (nullable = true)
 |    |-- xmin: float (nullable = true)
 |    |-- xmax: float (nullable = true)
 |    |-- ymin: float (nullable = true)
 |    |-- ymax: float (nullable = true)
 |-- country: string (nullable = true)
 |-- version: integer (nullable = true)
 |-- sources: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- property: string (nullable = true)
 |    |    |-- dataset: string (nullable = true)
 |    |    |-- record_id: string (nullable = true)
 |    |    |-- update_time: string (nullable = true)
 |    |    |-- confidence: double (nullable = true)
 |-- subtype: string (nullable = true)
 |-- class: string (nullable = true)
 |-- names: struct (nullable = true)
 |    |-- primary: string (nullable = true)
 |    |-- common: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: string (valueContainsNull = true)
 |    |-- rules: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- variant: string (nullable = true)
 |    |    |    |-- language: string (nullable = true)
 |    |    |    |-- value: string (nullable = true)
 |    |    |    |-- between: array (nullable = true)
 |    |    |    |    |-- element: double (containsNull = true)
 |    |    |    |-- side: string (nullable = true)
 |-- is_land: boolean (nullable = true)
 |-- is_territorial: boolean (nullable = true)
 |-- region: string (nullable = true)
 |-- division_id: string (nullable = true)
 |-- theme: string (nullable = true)
 |-- type: string (nullable = true)
```

divisions/division_boundary

```
root
 |-- id: string (nullable = true)
 |-- geometry: binary (nullable = true)
 |-- bbox: struct (nullable = true)
 |    |-- xmin: float (nullable = true)
 |    |-- xmax: float (nullable = true)
 |    |-- ymin: float (nullable = true)
 |    |-- ymax: float (nullable = true)
 |-- country: string (nullable = true)
 |-- version: integer (nullable = true)
 |-- sources: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- property: string (nullable = true)
 |    |    |-- dataset: string (nullable = true)
 |    |    |-- record_id: string (nullable = true)
```

```
|    |    |-- update_time: string (nullable = true)
|    |    |-- confidence: double (nullable = true)
|-- subtype: string (nullable = true)
|-- class: string (nullable = true)
|-- is_land: boolean (nullable = true)
|-- is_territorial: boolean (nullable = true)
|-- division_ids: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- region: string (nullable = true)
|-- is_disputed: boolean (nullable = true)
|-- perspectives: struct (nullable = true)
|    |-- mode: string (nullable = true)
|    |-- countries: array (nullable = true)
|    |    |-- element: string (containsNull = true)
|-- theme: string (nullable = true)
|-- type: string (nullable = true)
```

15.3. URL Structure

Let's break down the components in that path:

```
api.cadencemaps.infs.ch/cadencemaps/release/[date]/theme=[theme]/type=[type]/
country=[country]/*
```

- [date]: Releases follow a date-based versioning scheme in the format YYYY-MM-DD
- [theme]: One of the two data themes: `places` or `divisions`
- [type]: A feature type within a theme, e.g. `place` or `division_area`
- [country]: A country code, e.g. `CH` for Switzerland
- /*: The file type Overture uses to store and deliver the data, the * indicates you want all of the Parquet files in a particular directory

If you'd like to get the data of all countries, replace `country=[country]` with `*`. The URL would then look like this:

```
api.cadencemaps.infs.ch/cadencemaps/release/[date]/theme=[theme]/type=[type]/*/
*
```

15.4. Examples

The following examples show how to query the data using different tools.

15.4.1. DuckDB

When using DuckDB with the data, you need to install the `httpfs` and `spatial` extension and configure the S3 endpoint:

```
INSTALL httpfs;
LOAD httpfs;
INSTALL spatial;
LOAD spatial;
SET s3_endpoint='api.cadencemaps.infs.ch';
SET s3_url_style='path';
```

With this out of the way, you can query the data as you would with any other Parquet file.

```
SELECT names, categories, websites, emails, phones, addresses
FROM read_parquet('s3://cadencemaps/release/2025-06-04/theme=places/type=place/country=CH/*', hive_partitioning=1)
WHERE
  categories.primary = 'restaurant'
  OR 'restaurant' IN categories.alternate;
```

Through the use of hive partitioning, multiple countries can be selected in a single query. Note the double star in the path:

```
SELECT *
FROM read_parquet('s3://cadencemaps/release/2025-06-04/theme=places/type=place/*/*', hive_partitioning=1)
WHERE
  country IN ('DE', 'CH', 'LI', 'AT')
  AND categories.primary = 'restaurant'
  OR 'restaurant' IN categories.alternate;
```

15.4.2. Python

DuckDB Python package

To use the data in Python via DuckDB, you need to install the `duckdb` package.

Command Line:

```
pip install duckdb
```

Jupyter Notebook:

```
!pip install duckdb
```

Then, install the required DuckDB extensions and configure the S3 endpoint. It's possible to convert the query result into different formats, such as a Pandas DataFrame, NumPy array, or Arrow Table.

```
import duckdb

con = duckdb.connect()

con.execute("INSTALL httpfs; LOAD httpfs;")
con.execute("INSTALL spatial; LOAD spatial;")
con.execute("SET s3_endpoint='api.cadencemaps.infs.ch';")
con.execute("SET s3_url_style='path';")

query = """
    SELECT *
    FROM read_parquet('s3://cadencemaps/release/2025-06-04/theme=places/type=place/country=CH/*', hive_partitioning=1)
    WHERE
        categories.primary = 'restaurant'
        OR 'restaurant' IN categories.alternate limit 10;
"""

# Option 1: Fetch all rows as a list of tuples
```

```
result = con.execute(query).fetchall()

# Option 2: Fetch as a Pandas DataFrame
result = con.execute(query).fetchdf()

# Option 3: Fetch as a NumPy array
result = con.execute(query).fetchnumpy()

# Option 4: Fetch as an Arrow Table
result = con.execute(query).fetch_arrow_table()

# Option 5: Fetch as a Polars DataFrame
result = con.execute(query).pl()

print("10 Restaurants from Switzerland:")
print(result)
```

PyArrow and GeoPandas

To use the data in Python via PyArrow and GeoPandas, you need to install the `pyarrow`, `geopandas`, `s3fs`, and `shapely` packages.

```
import geopandas as gpd
import pyarrow.parquet as pq
import s3fs
from shapely import wkb

fs = s3fs.S3FileSystem(anon=True, endpoint_url='https://api.cadencemaps.infs.ch')

parquet_path = 'cadencemaps/release/2025-06-07/theme=places/type=place/country=CH/'

dataset = pq.ParquetDataset(parquet_path, filesystem=fs)
df = dataset.read().to_pandas()

# Convert WKB bytes to Shapely geometry
if 'geometry' in df.columns:
    df['geometry'] = df['geometry'].apply(wkb.loads)

# Convert to GeoDataFrame if geometry is present
gdf = gpd.GeoDataFrame(df, geometry='geometry')


# Filter rows where primary category is 'restaurant' OR 'restaurant' in alternate categories
def is_restaurant(row):
    try:
        if row['categories']['primary'] == 'restaurant':
            return True
        if 'restaurant' in row['categories'].get('alternate', []):
            return True
    except Exception:
        return False
    return False
```

```
31
32 filtered_gdf = gdf[gdf.apply(is_restaurant, axis=1)]
33
34 # Select relevant columns
35 result = filtered_gdf[['names', 'categories', 'websites', 'emails', 'phones',
   'addresses']]
36
37 # Display or use as needed
38 print(result.head())
```

15.4.3. QGIS

To load the data into QGIS, you need the addon “GeoParquet Downloader” plugin[7] by Chris Holmes. If our pull request has not yet been merged, be sure to install the plugin from the source code of the pull request[8].

After installing the plugin, you can load the data by following these steps:

1. Activate the OpenStreetMap layer in the sidebar (Figure 38)
2. Zoom to the needed extent (Figure 38)
3. Click on the “GeoParquet Downloader” plugin in the menu bar (Figure 38)
4. Select “Custom URL” (Figure 39)
5. Enter the URL of the GeoParquet data you want to load. You can find an up-to-date link on https://cadencemaps.infs.ch/ (Figure 39)
   - Important: The URL must start with `minio://`, not with `s3://`.

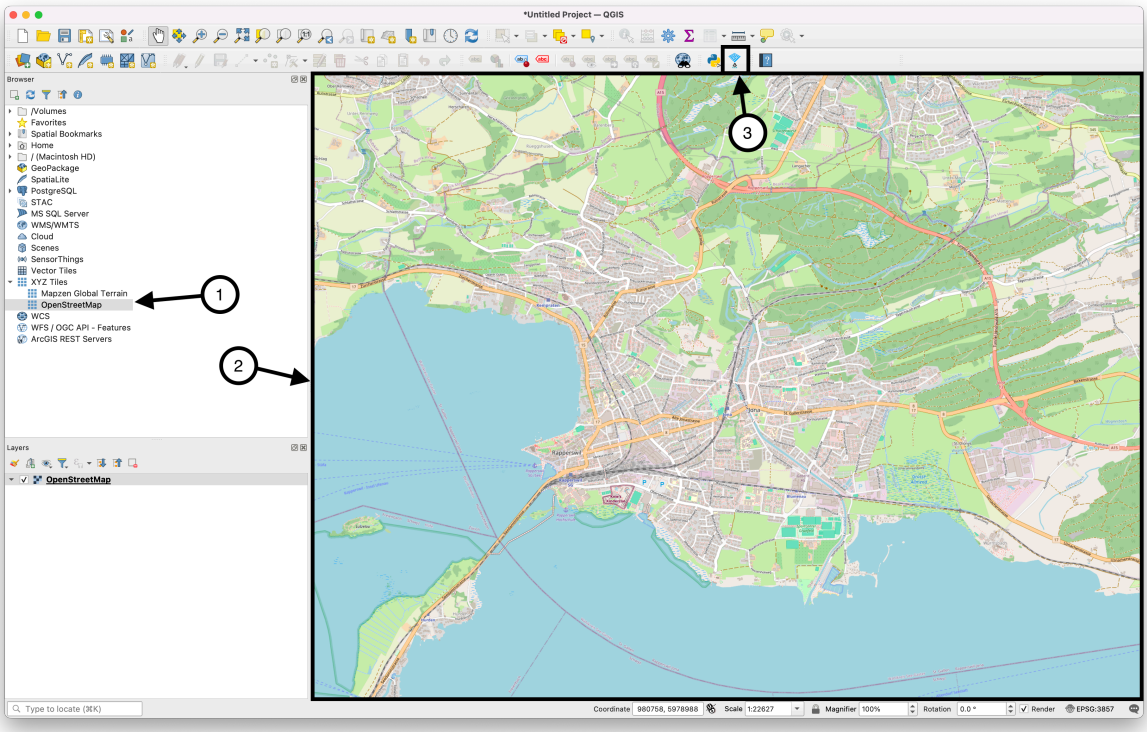


Figure 38: QGIS Guide - Steps to load data

[7] https://github.com/cholmes/qgis_plugin_gpq_downloader
[8] https://github.com/cholmes/qgis_plugin_gpq_downloader/pull/122

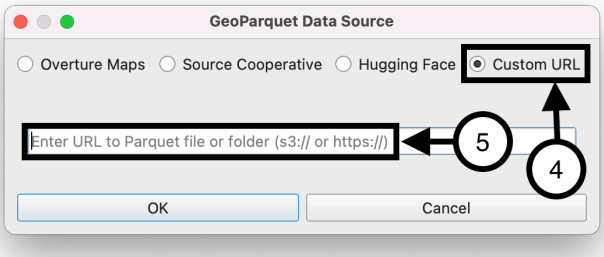


Figure 39: QGIS Guide - Popup to enter data source URL