Doctoral theses at NTNU, 2021:17

Atle Frenvik Sveen

An Event-Based Pipeline for Geospatial Vector Data Management

Doctoral thesis

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of Philosophiae Doctor
Faculty of Engineering
Department of Civil and Environmental Engineering

NTNU
Norwegian University of Science and Technology

Atle Frenvik Sveen

An Event-Based Pipeline for Geospatial Vector Data Management

Thesis for the Degree of Philosophiae Doctor

Trondheim, January 2021

Norwegian University of Science and Technology
Faculty of Engineering
Department of Civil and Environmental Engineering

NTNU

Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Engineering
Department of Civil and Environmental Engineering

ISBN 978-82-326-4816-0 (printed ver.)
ISBN 978-82-326-4817-7 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

Doctoral theses at NTNU, 2021:17

Printed by NTNU Grafisk senter

“Map-making had never been a precise art on the Discworld. People tended to start off with good intentions and then get so carried away with the spouting whales, monsters, waves and other twiddly bits of cartographic furniture that they often forgot to put the boring mountains and rivers in at all.”

— Terry Pratchett

Table of Contents

Preface

Ever since we started exploring the world, information about where things are and how to get there has been valuable and sought-after. The hand-drawn, and later printed, map provided an efficient mechanism for storing and communicating this information. While the digital revolution did not render maps outdated, it changed the landscape. At the core of this revolution lies two important changes to how we think about maps.

First, the digital revolution established a clear boundary between the physical map and map data. While a printed map traditionally was the only representation of map data, it is now one of many representations. Digital map data, or geospatial data, is a core component in search engines, navigational services, and recommendation engines, and is extensively used in planning processes, urban development, retail, and real estate. In addition, geospatial data plays a major role in handling climate challenges, and current events have demonstrated its importance in handling a pandemic.

Second, the digital revolution democratized the map. Surveying and cartography used to be complex and labour-intensive tasks, and the state usually took the role as a provider and maintainer of maps. While the state still values, produces, and maintains maps, this monopoly is a relic of the past. Private corporations provide a plethora of maps and location-based services and numerous businesses provide value-added services on top of governmental, private, and even personal map data. The rise of crowdsourced encyclopaedias paved the way for the crowdsourced map, where volunteers contribute their time and skills to map the world. Thus, geospatial data, which used to be scarce, is now ubiquitous and plentiful in most parts of the developed world.

A common denominator for much of this geospatial data is an open license. The creators provide the data to everyone to use, explore, enhance, and monetize. Why? The reasons are as varied as the actors, but a common theme is a combination of civic duty, personal convictions, moral sense, and cold calculation.

Against this backdrop of abundant and freely available geospatial data, a series of interesting research challenges can be outlined. Namely, how do we process, store, and manage such vast amounts of data? And, how do we deal with issues of privacy, accuracy, and accountability?

These are not just interesting research questions. They are also highly relevant issues. The geospatial industry is currently looking for solutions to handle geospatial data in a more efficient manner. This in turn drives innovation and digitization, which opens new possibilities. While spatial may not be *that* special, geospatial data is extremely relevant in a wide range of solutions. Enabling efficient use, re-use, and enhancement of existing data repositories is key for rapid product development. The winners in this race will be the ones who successfully consume, process, store, and manage the flow of heterogenous geospatial data from disparate sources, so that they can add value to the data.

These challenges are the starting point of this thesis. We describe how an *event-based pipeline for geospatial vector data management* can be created and present a solid foundation for implementation. This pipeline will enable efficient updating and versioning of open geospatial datasets and allow access to both current and historical data, while enabling a storage layout that is able to scale horizontally. The individual components of the pipeline are either based on novel research or described using work

from both the geospatial industry and academia. This combination of research and re-use ensures both a running start and avoids re-inventing the wheel.

Acknowledgements

Most PhD candidates are fortunate if they have one good supervisor. I've had three. Terje Midtbø took on the role as my main supervisor after having spent years trying to convince me that doing a PhD was the right thing to do. Alexander Salveson Nossum took on the role as co-supervisor, representing Norkart AS. Thank you both for nudging me in the right direction, collaborating, providing feedback, coping with my rants, care packages, and for inspiring conversations over a beer or three.

In addition to my official supervisors, I am indebted to my third, unofficial supervisor, my dear Birgitte. From the start you supported the idea, provided feedback and mental support, and you've spent countless hours proofreading and improving my texts with questions and helpful advice. In addition, we've built a house and a family, travelled the world, and enjoyed life together. I truly love you! And Hallvard, who in the final months detracted me from work and taught me that the path from thesis to faeces is short.

Norkart AS has supported this project both financially and intellectually. Having the ability to work on day-to-day tasks alongside this project, while anchoring my ideas in real-life uses-cases, has worked wonders for my sanity. Thanks to all my colleagues, especially at the Trondheim office, for your ideas, discussions, and most of all your patience. The group at NTNU Geomatics is a small, but fine, group, which made my days spent at Gløshaugen both interesting and worthwhile. Thank you!

Thanks to my family and extended family for providing the right amount of interest, while allowing me to think about other things. And, of course, thank you to the neckbeards at #flod. Virtual (and physical) rants and beers with you guys kept me going!

Introduction

In principle, geographical information (GI) does not need to be handled by computers. A paper map is the most common example of an analogue representation of GI. However, a digital representation of GI is for all intents and purposes a model created and stored using modern digital computer technology (1).

Thus, the advent of computer technology transformed the idea of geographical information. Digital geographical information could be rendered as an on-demand printed map, tailored to the use-case, rather than as a general-purpose map. Further advances meant that the map did not need to be printed at all, it could be transmitted using the Internet and rendered on-screen as needed (2). Rendering maps, both analogue and digital, is but one possibility using digital geographical information. The separation between geographical information and the physical map enabled services such as route calculation, finding nearby points of interest, automation of zoning and planning tasks, retail optimization, collaboration on development projects, and monitoring of environmental factors, to name some examples.

The digitization of maps also brought changes to how map data is created and maintained. The traditionally high cost of surveying meant that map production was the domain of the state and large corporations. Their practice of selling physical maps to the end-user carried over to the digital realm.

In an effort to “democratize the map” (3) OpenStreetMap (OSM) was created. This is an open-licenced, world-spanning map database, created and maintained by volunteers, operating using the crowdsourcing principle made popular by Wikipedia (4). Much of the

OSM data is created by its volunteers through means such as GPS logs and tracing of aerial and satellite imagery (5). This is what Goodchild termed Volunteered Geographical Information (VGI) (6).

In parallel with this development, states and governmental institutions changed their practices. While geospatial data surveyed and maintained by the state traditionally was sold to third-parties, the concept of Open Data challenged and changed this practice. Open Data is data that can be "freely used, modified, and shared by anyone for any purpose" (7). An early example of open geospatial data from governments can be found in Denmark. In 2013, the Danish government made almost all their governmental geospatial data available as Open Data, expecting "a positive effect on the national economy, and that it will create growth and a more efficient public sector" (8). The US practice of keeping governmental data free and open is rooted in the Public Domain law (9), but still ensures that everyone can access and use geospatial data free of charge.

Thus, due to sources such as VGI and Open Data, geospatial data is no longer a scarce resource. An abundance of data has replaced data scarcity. The challenge is no longer to get hold of data, but to handle an ever-increasing flow of both new datasets and revisions to existing datasets.

Open geospatial distributors are usually using what Worboys (10) refers to as a "stage one" spatiotemporal system. This means that each release of a dataset represents a temporal snapshot of the data. New snapshots are released in bulk at regular intervals, even though only a fraction of the data may have changed. This problem was noted nearly 20 years ago by Cooper and Peeled (11), who in their report to the ICA Working Group on Incremental Updating and Versioning concluded that «there should be no need to

redistribute an entire data set to its users to propagate changes that are only minor or few in number». This issue is still relevant today, as more geospatial data is made available, and the use-cases have become more varied.

Event sourcing (12) is an interesting approach to handle this continuous flow of data. In a true-to-the book event-sourced system, all changes to a dataset would be recorded as events and distributed to consumers. This is akin to what Worboys (10) refers to as a "stage three" spatiotemporal system. By transforming "stage one" snapshots into "stage three" events as shown in Figure 1, an event-based approach to geospatial vector data management can be applied without being dependent on third parties having to change their infrastructure.

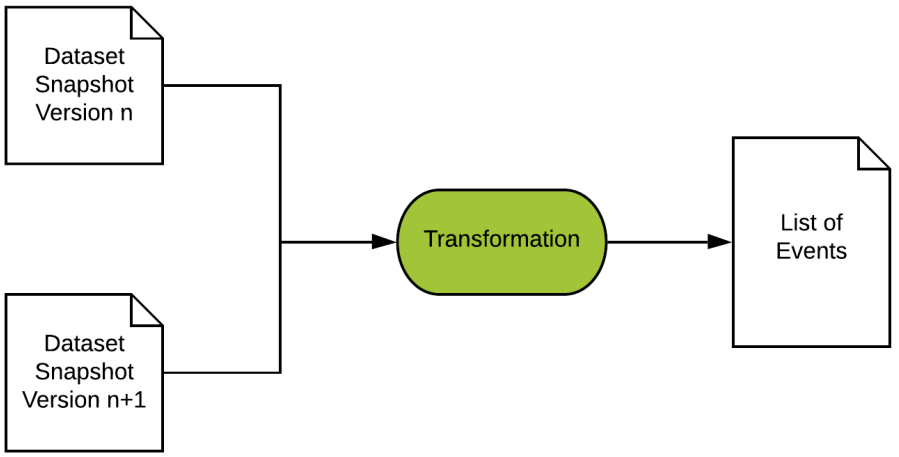

***Figure 1**: Illustration showing how two temporal snapshots of a dataset can be transformed into a list of events.*

Another challenge of data abundance is the inevitable issue of identifying the "best" data. What data is the "best" is of course dependant on the task at hand, but usually the best data is a combination of data from multiple sources. With geospatial data available as

Open Data from national mapping agencies and other governmental sources, as proprietary data from commercial mapping companies, and as crowdsourced data or VGI (6), an important task is data consolidation or conflation. While advances in artificial intelligence (AI) and machine learning (ML) offers a promising trajectory, some degree of manual oversight is often required (13,14).

An automated process for converting a large number of bulk-updated geospatial datasets into event sourced datasets is the main idea of this thesis. In addition, we examine possible applications of event sourced geospatial datasets, aimed both at replacing existing solutions and creating new ones. This process is referred to as *an event-based pipeline for geospatial vector data management*. Such a pipeline should be easy to set up, customize, and should handle heterogenous geospatial data from a wide selection of sources. Some of the pipeline components will be described and examined in detail, while others will be covered more in brief. Creation of the pipeline itself is not the main topic of this thesis, but discussion of how this can be achieved is provided where relevant.

Structure of the thesis

The rest of this thesis is structured as follows. First, the individual research papers that constitute this thesis are presented. Their motivation and scope, as well as the candidate's contributions are detailed. The methodology section provides a high-level overview of the methodology applied throughout the work, without repeating the content of each individual research paper. Following this, the background section expands on the introduction and provides an overview of the envisioned pipeline and its individual components. Relevant challenges, opportunities, and possible solutions for each component are presented and discussed. Some considerations about an actual implementation of such a pipeline are also presented. The main findings of the research papers are presented in the results section, which also presents an example implementation of the described pipeline in order to discuss each of the components. The work is concluded with some final remarks, pointing to the current state of development and possible future trajectories. The full research papers of this thesis are included at the end of this work.

Presentation of research papers

This section presents the individual papers of this thesis individually and positions them in the overarching scope of the project. The candidate's contributions to each paper are also presented.

Paper I

A. F. Sveen, 'The Open Geospatial Data Ecosystem', *Kart og plan*, vol. 77, pp. 108–120, 2017.

This review article was researched and written as part of an effort to obtain an overview of existing research and literature on open geospatial data. The article surveys the main sources of open geospatial data and examines the reasons for publishing such data. The related concepts of Volunteered Geographic Information (VGI), Participatory GIS (PGIS), and crowdsourcing are examined and placed into context. The term Open Governmental Geospatial Data (OGGD) is suggested as a description of geospatial datasets released by governmental agencies. Motivations for data release, formats, and limitations, as well as interactions with VGI data repositories such as OpenStreetMap (OSM) are examined. All aspects of the paper, including motivation, research, and preparation, where carried out by the candidate.

Paper II

A. F. Sveen, A. S. S. Erichsen, and T. Midtbø, 'Micro-tasking as a method for human assessment and quality control in a geospatial data import', *Cartography and Geographic Information Science*, vol. 47, no. 2, pp. 141–152, 2019, doi: 10.1080/15230406.2019.1659187.

The underlying motivation behind this paper was to investigate the relationship between OGGD and VGI, and to gain an understanding of how crowdsourcing principles such as micro-tasking can facilitate integration of data from disparate sources. Based on previous research, we found that OSM utilizes micro-tasking principles in order to import OGGD. However, no clear guidelines or recommendations on how these principles should be applied exists. Creating an online-experiment where volunteers carried out micro-tasking tasks related to a simulated building-data OSM import was pitched to Anne Sofie Strand Erichsen, who made this the topic of her master's thesis (15). The results of the online experiment, with 164 participants was presented at a poster-session at the 2017 FOSS4G Conference in Boston (16).

The results of the online experiment were further analysed and put into context in this paper, which concludes that while geospatial micro-tasking is still in its early stages it is an interesting approach for quality control and assessment when geospatial datasets are merged or consolidated.

The idea and motivation behind the work was conceived by the candidate, who also acted as a supervisor on the master's thesis. Data analysis and major parts of manuscript preparation was carried out by the candidate.

Paper III

A. F. Sveen, 'Efficient storage of heterogeneous geospatial data in spatial databases', *Journal of Big Data*, vol. 6, no. 1, pp. 1–14, Nov. 2019, doi: 10.1186/s40537-019-0262-8.

The rationale behind this paper was to investigate if a practice of using NoSQL principles to store OGGD from heterogenous data sources (17) was the best solution to the problem.

In order to investigate this, the Heterogenous Open Geodata Storage (HOGS) system was implemented and used to compare the performance of a traditional, table-based data storage structure and a NoSQL approach, using the PostgreSQL/PostGIS database in both cases

The paper covers the implementation and test of the Heterogenous Open Geodata Storage (HOGS) system. This system is designed as an efficient and resilient solution for importing geospatial datasets from heterogenous data sources. The design of HOGS allowed for a comparison between a traditional, table-based data storage structure and a NoSQL approach, using the PostgreSQL/PostGIS database in both cases. Implementation of the HOGS system, benchmark execution, and analysis and manuscript preparation were all carried out by the candidate.

Paper IV

A. F. Sveen, 'GeomDiff — an algorithm for differential geospatial vector data comparison', *Open Geospatial Data, Software and Standards*, vol. 5, no. 1, pp. 1–11, Jul. 2020, doi: 10.1186/s40965-020-00076-4.

While the HOGS system did include mechanisms for handling dataset updates, this was a rudimentary "temporal snapshot"-approach (10). In order to support more fine-grained and efficient versioning, we started looking into the use of diffing algorithms for geospatial vector data. While our research indicated that diffing algorithms are used in this manner, we found no implementations of algorithms specifically tailored to geospatial vector data. This was the rationale behind the implementation of GeomDiff. GeomDiff is an algorithm and C# implementation of diff creation, application, and reversion for geospatial vector data, as well as a binary storage format for such diffs.

The GeomDiff implementation was tested by comparing its performance with three general-purpose diffing algorithms adopted to geospatial vector data using pre- and post-processing steps. The algorithms were tested using about 2.5 million OpenStreetMap geometries in a public cloud computing environment. While the GeomDiff algorithm failed to meet expectations for linestring and polygon geometries, this paper shows promising results that encourages further development. All aspects of the paper, including design, implementation, statistical analysis, and manuscript preparation, were carried out by the candidate.

Methodology

The methodology used throughout this thesis is for the most part related to developing software and examining the performance of this software. Thus, in addition to the scientific method, principles of software engineering have been a cornerstone in this work. For details related to the specific methodology employed in each paper, we refer to the individual manuscripts. In the following section we will show how this methodology was applied to complete this thesis.

Paper I is a literature review, Paper II is an online experiment with human participants, Paper III is a benchmark run on a single computer, and Paper IV is a benchmark run in parallel in a cloud computing environment. Each paper required its own approach, design, setup, and analysis. In the following, the different approaches taken, and considerations made are presented and discussed.

Literature reviews can be carried out using either a systematic approach (18), or a more conventional, open-ended approach. Paper I have a clear mission; establish how the concept of Open Data relates to geospatial data. Early on, it became evident that while research on Open Geospatial Data is limited, the related terms Volunteered Geographic Information (VGI) and Participatory GIS (PGIS) or Public Participation GIS (PPGIS) is covered in great detail. This meant that these topics were naturally included in the review. Likewise, as we saw many connections between the free and open geospatial software (foss4g) movement and open geospatial data, we found it relevant to include in the overview.

Paper II examined the performance of human workers in an online experiment. Experiments including human participants require careful planning, execution, and post-processing. The system under test should be quality-checked before a full scale online experiment is launched. In Paper II this was handled by conducting a pilot test with a small number of participants. The System Usability Scale questionnaire (19) was used in order to standardize responses. Based on this pilot, improvements were made before conducting the experiment. Another issue is participant recruiting. It is imperative to have a representative selection of participants to gain relevant insights from the experiment. Using social media and mailing lists available to the authors may skew the results towards participants with existing experience or interest in the subject. However, these are the same channels that would be used in order to recruit participants to an actual task.

Papers III and IV are both focused on examining the performance of a computer program. The premise of a computer program or an algorithm is that it provides the same result given identical input. This process should theoretically take the same amount of time on each repetition. However, there are several factors that can influence running time. Hardware capabilities such as processor speed is an obvious factor. But even repeated measurements of the same task running on identical hardware may yield different results. System interrupts, processor time allocation, available memory, and network speed and latency are all factors that may influence running time.

One approach to minimize these effects is to run the experiment locally on a single machine with as many other applications as possible deactivated. If the measurements are run multiple times, and the results averaged, a reasonable approximation can be found. However, the absolute values obtained from running on a single machine will not be

applicable in other configurations. Thus, it is important to focus on relative differences between several approaches tested on the same configuration. This approach was chosen for testing the two storage layouts used by the HOGS system in Paper III.

Another approach for testing the performance of a computer system is to run the experiment in a public cloud computing environment. A cloud computing environment provides customizable and configurable access to a powerful computing environment with the ability for both vertical and horizontal scaling (20). This means that measurements will be more in line with expected real-life performance, and that we can run multiple measurements in parallel. However, we are not guaranteed that all executions will happen in isolation (21).

The experiments in Paper IV, which compared the performance of the GeomDiff algorithm to three alternatives, was carried out in the Microsoft Azure cloud computing environment. This allowed us to handle a large volume (2.5 million geometry pairs in four algorithms) of operations. This experimental setup required substantial preparation but ensured that the experiment was run in a controllable and repeatable environment. The pay per use model also meant that we had no large up-front hardware investments. Another benefit was that the actual measurements recorded reflects typical performance on a public cloud platform. However, as hardware capabilities constantly increase, these results will quickly be outdated, and the relative difference between the different algorithms are still what is important.

Papers II, III, and IV all include computer code written by the authors. In addition, several scripts and other supporting applications have been written during this work. The process of writing software for a scientific experiment or implementing algorithms that are the

topic of a scientific work, deserve some discussion. Throughout the work on this thesis, we have sought to follow existing best practices on software development. Well-structured, readable code, with clear separation of concerns have been an important goal (22,23). In terms of specific programming languages, we have used the most appropriate language given the task at hand and have tried to avoid religiously favouring one language over another. Prior knowledge is of course an important factor but external dependencies, such as intended platform and available libraries, have also dictated language choice.

One important decision, made early on in the process, was to release all code written as part of this thesis as open source software. This ensures transparency, as peers are free to inspect, validate, and critique the actual code written. In addition, it enables for re-use if the ideas presented are used as basis for further research or commercial implementation. The practice of releasing code as open source software also inspires to write cleaner and more readable code.

Statistical analysis is an important task, no matter how an experiment is conducted. Knowing what kind of analyses to utilize and the requirements of each analysis is important, but an efficient method for running and re-running analyses are just as important. This is especially important when working with a large number of measurements. Approaching this task with the mindset and toolbox of a programmer is useful. We found that the Pandas (24) library for the Python programming language provides a great starting point for statistical analysis of large amounts of data. The library relies on the SciPy (25) and NumPy (26) libraries, which ensures access to a broad spectrum of statistical tools, as well as interoperability with a wide range of tools and extensions. By formalizing all calculations as code modifications can be carried out as

needed. The code used for calculations can also be tracked using source control systems and shared with reviewers and other interested parties.

Background

Geospatial big data

The Internet has changed the way data is distributed and has made map data easier to find than ever. New technologies and techniques for surveying, monitoring, and disseminating geospatial data has created a data abundance. Data from traditional sources such as governmental institutions are increasingly made available free of charge as Open Data, and new sources of data such as VGI has materialized and matured. Thus, geospatial data is no longer the scarce resource it used to be.

The task of handling this increasing stream of geospatial data calls for new methods and techniques. In computer science, this shift is often characterized by the phrase "big data", commonly defined by the three Vs; Volume, Velocity, and Variety (27). These three characteristics can also be applied to geospatial data, in what can be defined as "geospatial big data". Here, by discussing each of the three Vs and providing examples from the geospatial domain, we argue that geospatial data has in fact transitioned to geospatial big data.

The increase in geospatial **data volume** can be illustrated by an OpenStreetMap example. Figure 2 shows two screenshots of OSM in Dublin, Ireland. The left-hand map is the example Goodchild (6) used in 2007, while the right-hand map is a current (2020) map of the same area. Although the cartography is different, the difference in map data is striking. The 2007 map shows several missing street names, no coverage in areas, and a limited set of features mapped. In contrast, the 2020 map is almost too crowded, with features such as building footprints, walkways, and individual businesses mapped.

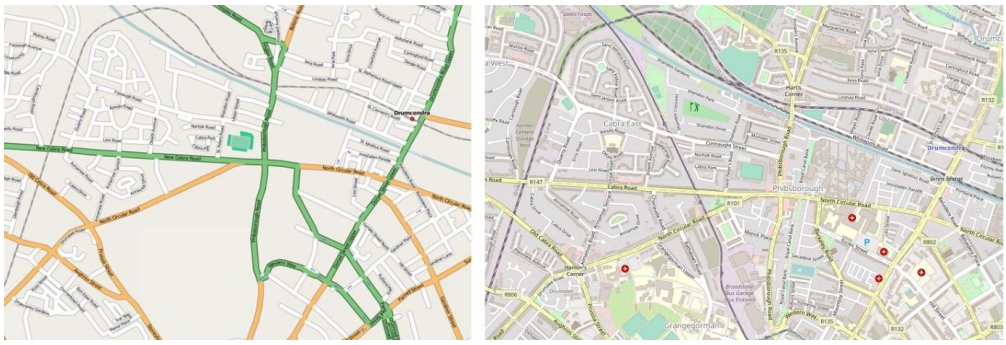

Figure 2: *OpenStreetMap coverage in Dublin, Ireland. Left is a screenshot from Goodchild (6) in 2007, right is from 2020.*

Data velocity refers to the pace at which data is created or updated. Geospatial data has traditionally been slow data. National geospatial datasets maintained by governmental institutions are often released at monthly, quarterly, or even at yearly intervals. On the other hand, technologies such as modern sensors and the Internet of Things (IoT), are capable of delivering new measurements with a geospatial component every second. Another example of continuously updated, high-velocity geospatial data is OSM, which had about 3.5 million edits per day in early 2020 (28).

In terms of **data variety**, geospatial data really *is* special (29). This manifests through storage formats, data schemas, and projections. There is a plethora of formats to choose from, as witnessed by the list of 99 geospatial vector data formats currently supported by the ogr2ogr tool (30). Another example is a survey of Open Geospatial Data, which found that some cities use as many as 12 different formats to distribute their geospatial data (31). In addition to a large variety in formats, there is a multitude of geographic reference systems to utilize, each with its strengths and weaknesses. On top of this, geospatial data is usually accompanied by attribute data. These attributes are either structured using

widely different schemas or eschew a schema altogether and apply a consensus-based tagging system.

In light of this discussion we argue that geospatial data exhibits characteristics of big data, with an increase in both Volume, Velocity, and Variety. Just as big data requires new approaches, we argue that geospatial big data calls for new methodologies for handling this increasing stream of data.

Event sourcing

Event sourcing is a pattern for data storage and management that uses events as a mechanism for storage (32). The pattern can be considered a variation of the commit log pattern used by relational database management systems (RDBMSes), and relies on an event store and a state machine to provide the current (or any previous) incarnation of an entity (33). The event sourcing pattern comes with the promise of simplified scalability and an efficient way to store and query historical records in a dataset.

The event sourcing pattern is well suited for handling a stream of geospatial data with increasing volume and velocity. In addition, it should be capable of handling data variety without sacrificing consistency. Thus, a thorough review of the principles behind event sourcing and how they apply to geospatial data is in order.

A fundamental building block in Event Sourcing is *the event*. As an example, in a system handling ship notifications, an event would be on the form "ship a arrived at port x" or "ship b departed from port y" (12). This approach maps well to an IoT device tracking sheep on pasture, which uses GPS to log sheep location every hour. An event would be "sheep x moved n meters east, m meters north". However, many governmental geospatial datasets are released in bulk as temporal snapshots (10). These snapshots are a complete representation of the mapped features at the current time and does not carry any information about what was changed from the previous snapshot.

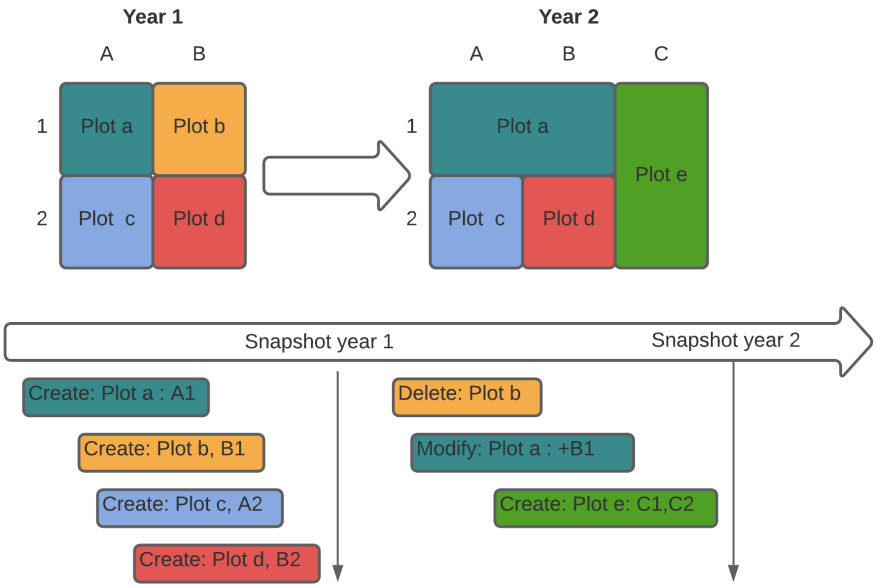

Figure 3: *Two representations of a plot map, showing how five plots change over time. The upper half of the figure shows two temporal snapshots, taken in year 1 and 2, respectively. The lower half is an event log, showing the creation, modification, and deletion of plots as events on a timeline. The timeline also indicates the points where temporal snapshots were taken.*

The simplified plot map in Figure 3 illustrates how temporal snapshots differ from events. Using a temporal snapshot approach, we would get the two separate maps depicted in the upper half of the figure, even though neither Plot c nor Plot d are changed from year 1 to 2. The events depicted in the lower half provides us with information about when the individual plots were created, deleted, and modified. However, in order to realize that Plot a occupies A1 and B1 in year two, we need both the create event and the modify event.

This situation is also illustrated in Table 1 and Table 2. Table 1 shows two temporal snapshots of a collection of linestrings (version n and version $n + 1$), as well as a column

describing what happened to the feature between versions. This table contains redundant information, since feature 1 is unchanged. However, a complete representation of the geometry is repeated for each snapshot. The change to feature 2 consists of changing the x-coordinate of node 2, which means that the $n+1$ version also contains redundant information.

Feature Id	**Geometry, version n**	**Geometry, version $n+1$**	**Action**
1	(1 1, 1 2, 1 3)	(1 1, 1 2, 1 3)	Unchanged
2	(1 1, 2 1, 3 1)	(1 1, 2 1, 4 1)	Modified
3	(4 5, 4,8)	-	Deleted
4	-	(5 4, 8 1, 8 2)	Created

Table 1: *Example of a bulk-updated geospatial dataset in two versions, n and* $n+1$*. Since the features have unique IDs that are consistent across versions, we can deduce whether they are unchanged, changed, created, or deleted from version n to* $n+1$*.*

Table 2 shows the same changes to the same dataset, but this time expressed as an event log. First, all features in version n are created, then the changes made in version $n+1$ are described. In this event log, the concept of a dataset version is redundant. Each feature is updated independent of the other features, and a change to a small subset of features does not implicate a new dataset version. In order to retrieve a given version of a feature, the events related to that feature is played back until the required version is obtained.

Feature Id	Timestamp	Action	Geometry
1	1	Create	(1 1, 1 2, 1 3)
2	2	Create	(1 1, 2 1, 3 1)
3	4	Create	(4 5, 4,8)
2	7	Modify	{2: x+1}
3	8	Delete	(4 5, 4,8)
4	11	Create	(5 4, 8 1, 8 2)

Table 2: *A series of events describing the creation and change of a spatial dataset. Note that the modify events only describe modifications needed to modify the existing geometry.*

An event log, similar to the example in Table 2, is at the heart of an event sourced storage system. When this event log is coupled with an algorithm capable of applying all events to a feature, the system can provide any requested version of a feature.

As noted in Table 2, a modification to a vector feature is only denoted by a record of the delta of each changed node, as opposed to storing a complete representation of the geometry after modification. This is akin to noting “deduction of 1$” in a bank ledger, instead of noting “new balance 100 099$”. This approach allows for more compact storage, as deltas only denote what have changed. In addition, it enables us to keep track of what was actually changed during an event. This concept is known as a diff in the field of computer science, and algorithms for effectively creating diffs for text and source code have been known since the 1970’s (34,35). These diffing algorithms can be used with geospatial vector data when combined with post- and pre-processing steps that convert to and from a textual representation. However, the mathematical nature of geospatial vector data means that faster and more compact diffs can be created using a tailored algorithm. Despite this, we were unable to find any examples of such algorithms in the literature.

This was the rationale behind the development of the GeomDiff algorithm presented in Paper IV.

Read projections

One important consideration of the event sourcing pattern is related to performance. As discussed, an event sourced system builds a feature by applying all events stored in the event store to arrive at the most recent version of a feature. In a system with many events stored for a feature, this process can take considerable time. While a faster algorithm for diffing can improve performance, there will always be a correlation between number of events and running time. This problem can be overcome by applying the practice of rolling snapshots (32), a mechanism which avoids loading and applying all events.

While rolling snapshots may speed up ID-based queries to match the performance of relational databases, they fall short when data needs to be queried in other ways. Geospatial queries can illustrate this. In order to perform a point-in-polygon or intersection query, the actual geometries need to be accessible. While optimizations, such as storing minimal bounding boxes along with the event, could be applied to the event sourcing pattern, there are other more elegant solutions. Using a spatial database as a read projection is one possibility.

The idea behind read projections is rather straightforward. Instead of reading directly from the event source API, a system is set up in front of the API. This system consumes events as they occur and projects the data into a format suitable for reading. In our geospatial example, this format would be a spatial database. This means that users of the data can connect to the database as previously, without having to interact with the event sourced system and adopt to new APIs. One caveat is that the read projection is, as its name implies, a read-only data storage. If clients are used to writing data, this will have to be handled using other mechanisms.

The read-only nature of read projections brings several advantages. Since there is no need to propagate changes, we can easily use a number of different read projections. Data can also be transformed before being stored in the read projection. This means that the concept of a central database that conforms to the needs of all users is not required. As a consequence, we can avoid compromises that degrade performance and ease of use. Each application can use its own read projection, consisting of exactly the data it needs, transformed to match its usage patterns.

While an event sourced system combined with read projections is able to provide data access to traditional geospatial applications such as a desktop GIS, a tile- or WMS-server, and traditional web APIs, it has other applications as well. An event sourced system enables easy access to historical data, since it by design stores all historical versions of a feature. With the rise in machine learning (ML) applications, training data is a valuable commodity. Thus, having the means to build a repository of historical, geospatial vector data is likely a worthwhile effort. Since an event sourced system by design tracks the complete history of a dataset, extraction of historical training data for statistical analysis and machine learning applications should be rather straightforward.

Another application of an event sourced geospatial storage system is utilization of the events themselves. Since each event describes a change to a particular geospatial feature, an event sourced system can be used to monitor changes to an area of interest. One example may be a dataset calculating flooding risks. When an update to this dataset is converted into events, each event can be checked against a list of properties, and owners alerted if there is a change in their flood risk calculation.

These examples show that the inherent properties of event sourcing enable both traditional database storage of geospatial features using a read projection, as well as applications that would be cumbersome to develop using a traditional approach with bulk updated datasets.

Vector data change detection

In the case of automated IoT-sensors the data stream can easily be treated as a series of events. A temperature sensor can trigger an event if the temperature changes above a given threshold. Creating an event sourced system on top of an existing stream of events is by far the simplest approach. However, open governmental geospatial data is rarely distributed as a stream of events. All the 47 nations who provide national map data as open geospatial data utilizes temporal snapshots, according to data from the Global Open Data Index (36). This means that most open geospatial data are updated in bulk at regular intervals. In addition, we have found no indication that institutions responsible for disseminating open geospatial data are considering a transition to an event based delivery mechanism.

While data owners could, in the future, adopt an event-based mechanism for data dissemination, this is uncertain and outside our control as data users. Thus, in order to reap the benefits of event sourcing without having to wait for a third-party, we have to act. We propose to introduce a mechanism for mapping bulk updated geospatial datasets to event-streams. This means that we control the whole process and are not dependant on third-parties adopting new mechanisms.

However, converting bulk-updated temporal snapshots of a dataset into an event log is not straightforward. In the example case illustrated in Table 1 and Table 2, the existence of object identifiers that are consistent across versions allows for a relatively simple conversion. Changes to a feature across dataset versions can be tracked using the object identifier. Unfortunately, not all bulk-updated datasets use object identifiers that are consistent across versions. What is needed is a robust mechanism for change detection.

The term “change detection” in the geospatial domain is usually associated with detecting changed areas in raster imagery (37). However, the term is also used more broadly in computer science to refer to “detecting and representing changes to data” (38) . Here, we adhere to this broader definition and use it to describe the process of detecting changes to bulk-updated geospatial vector datasets. Chawathe et al. (38) lists four key characteristics of a system for change detection in hierarchically structured information:

1. Nested Information
2. Object Identifiers not assumed
3. Old, new version comparison
4. High Performance

If the requirement to handle nested (or hierarchical) information is replaced with a requirement to handle geospatial vector features, these requirements describes what is needed to convert bulk-updated geospatial vector data into an event stream. Such an algorithm should consider both the vector geometry and accompanying attributes of a feature. The development of such an algorithm is outside the scope of this work, but we have identified several algorithms for vector geometries that could serve as the basis for, and constitute the first initial steps toward, such an algorithm (39,40).

Data conflation

So far, our discussion has revolved around how a single dataset can be transformed from temporal snapshots released in bulk to a stream of events using an automated pipeline. We have shown how this pipeline can be enhanced using read projections for accessing current and historical data, and how the events themselves can trigger actions. This means that event sourcing of geospatial datasets offers several new opportunities. Even more opportunities are presented when we start considering options for conflating several datasets. One can argue that this is nothing new. One of the pillars of GIS is the ability to superimpose different geospatial datasets in order to gain new insights.

What is new is that several datasets mapping *the same phenomena* can be combined. Data redundancy is a natural consequence of a growth in competing sources of geospatial information. OpenStreetMap was started as a countermeasure to closed, governmental datasets (4). Many of these datasets are now becoming available as open datasets. Thus, we have multiple sources of geospatial information mapping the same phenomena in the same area. While two different surveys of an area may aim to map the same set of features, they are likely to end up with dissimilar results. Surveying techniques may result in different accuracy and level of detail of the mapped geometries, and the intention and goals of the survey may result in different sets of attributes being collected.

An appropriate example is building footprints. The high-detail Norwegian topological governmental dataset FKB contains 5.4 million building footprints mapped using professional equipment and technology, according to a strict methodology and a formalized schema. OpenStreetMap users have also mapped building footprints in Norway, using satellite and aerial imagery without any strict procedure and using a tag-

based approach (41). This process has resulted in approximately 500.000 digitized building footprints in Norway (as of June 2018). As a general rule of thumb, official datasets are often more accurate, while crowdsourced datasets exhibit a greater variety of data. In this specific case, the FKB data have overall finer resolution and accuracy, while the OSM data contains additional information. Names of businesses, retail categories, and links to home pages and social media are some examples. Thus, for any given application, the ideal building dataset in Norway may be a combination of FKB and OSM data.

However, creating a conflated dataset is not straightforward. Deciding what to keep, what to reject, and what to merge is a decision that is difficult to formalize and encode in an algorithm. This decision is largely dependent on human judgement. This judgement stems from experience and human capabilities that are difficult to replicate using procedural algorithms or even machine learning.

On the other hand, the lure of automation is that it enables us to process data a lot faster than a human is capable of. This is the reasoning behind what is known as “human in the loop computing”, a combined approach where algorithms handle the “easy” cases, while the more difficult cases are handed over to humans. The results of human judgement is then fed back into the system to serve as training data for the algorithm (14).

Micro-tasking is a promising method for handling human interaction in such a system. In a micro-tasking scenario, a problem is partitioned and distributed to a crowd of individuals using the Internet. These tasks should not require extensive training and should be completable in minutes (42,43). Again, OpenStreetMap may serve as an example. Several imports of governmental open geospatial data to OSM has been carried out using a micro-tasking approach (44,45). Juhász and Hochmair (46) show an example

of a “human in the loop”-process for OSM imports, where “one part of the dataset was uploaded automatically, and the other one was set aside for the community to review.”

These examples show that “human in the loop computing” and micro-tasking are promising techniques for dealing with geospatial conflation tasks. However, there are still several unanswered questions regarding this approach. How does one decide what tasks are easy and difficult? How should the tasks be partitioned? The examples we have found on OSM imports used census tracts or similar partitions. While this is an easy solution, we are not certain that this approach is based on anything besides ease of use and assumptions. More research on how micro-tasking can be applied to a geospatial conflation task is thus required.

Implementing an event-based pipeline

Methods for handling geospatial big data, geospatial vector data change detection, geospatial diffing algorithms, and micro-tasking approaches to geospatial conflation tasks are thus important components of an event sourced pipeline for geospatial vector data management. Figure 4 shows an overview of such a pipeline and places the different components in context.

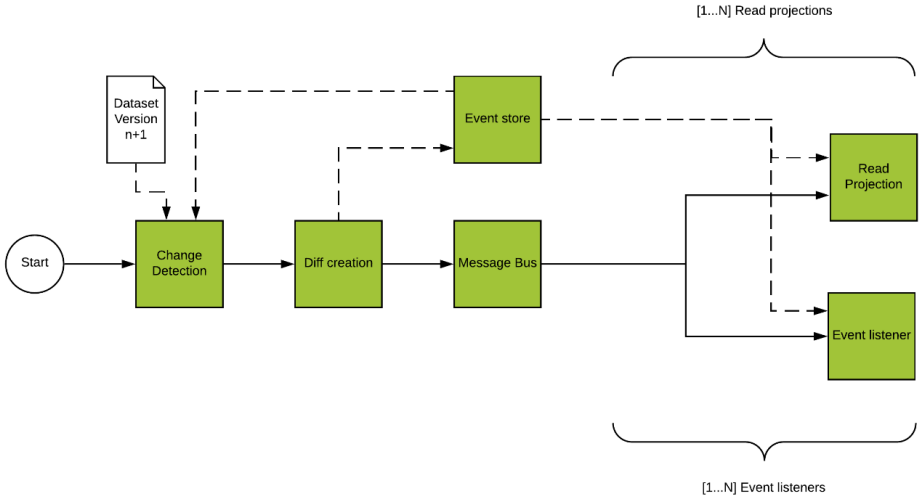

Figure 4: *Overview of an event-sourced pipeline for geospatial vector data.*

The actual implementation of an event-based pipeline is just as important as the individual components it consists of. One important issue is scalability. A pipeline such as this should be able to handle an increase in datasets, with loads in the range of thousands of datasets, each with several hundred thousand features. Fortunately, these datasets are not dependant on each other, so horizontal scale-out is an obvious solution. Another characteristic of such a pipeline is that the workload is not constant. If new versions of a bulk-updated dataset are released at a monthly schedule, the initial mapping to events is

run once a month. In these periods, we need sufficient processing power to handle the load quickly, but once finished this processing power is not required again until the next update.

These characteristics seems indicative that a public cloud computing platform may be a good platform. According to the US National Institute of Standards and Technology (NIST) “Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources […] that can be rapidly provisioned and released with minimal management effort or service provider interaction.” (20). These characteristics closely line up with the needs of such a pipeline. There is no need for an up-front investment in hardware, and the pipeline can be scaled out on demand. Focus on rapid provision and release is also an important factor, as this affects the ability of the system to include new datasets rapidly. Thus, we see the use of a public cloud platform as good match for the envisioned pipeline.

The NIST definition of cloud computing is rather broad, and further classification is often necessary. One option is to group cloud services by how many layers of abstractions they provide (47). In this scheme, Infrastructure as a service (IaaS), where the cloud provider “[…] provides the physical computing resources that are configured by the user to meet variable needs” offers the least amount of abstraction. Platform as a Service (PaaS) offers the users configurable software components. These components are usually databases, web servers, and middleware. Software as a Service (SaaS) is the next step up the abstraction-ladder and offers fully configured software available on-line.

A further abstraction is known by the oxymoron “serverless computing”, or Functions as a Service (FaaS) (48). The idea is to provide the user with a way to run pre-defined

functions which does not depend on state mutation. The actual server running the functions are abstracted out, and all the user is left with is a function that acts on its input to produce a result. The functions in a FaaS architecture can receive input and return results through mechanisms such as HTTP requests, databases, message queues, and file storage repositories.

While all these abstractions come with their strengths and weaknesses, we find that a high level of abstraction may be a good starting point. While a high level of abstraction usually means a limitation of capabilities it also allows for more rapid development. Since the high level of abstraction hides complexity, it allows the developer to focus on the task at hand, rather than supporting functionality. If a solution created using a high level of abstraction turns out to be less performant or scalable than required, it is always an option to re-create it using a lower level of abstraction. The FaaS paradigm also leans heavily on the concept of events (48), which is a good fit for the envisioned pipeline.

Since all the components of the outlined pipeline seems feasible to implement using FaaS-level abstractions, this seems like a good starting point. All the “big four” cloud platforms provide a FaaS offering (AWS Lambda, IBM Cloud Functions, Google Cloud Functions, and Azure Functions) (49). In principle, these technologies offer the same functionality, but there are differences in platforms, communication mechanisms, available programming languages, orchestration options, performance, and cost models.

Results

This thesis argues that an event-based pipeline is a viable solution for management of large amounts of bulk-updated geospatial vector data from heterogenous sources. Figure 5 shows an illustration of the envisioned pipeline.

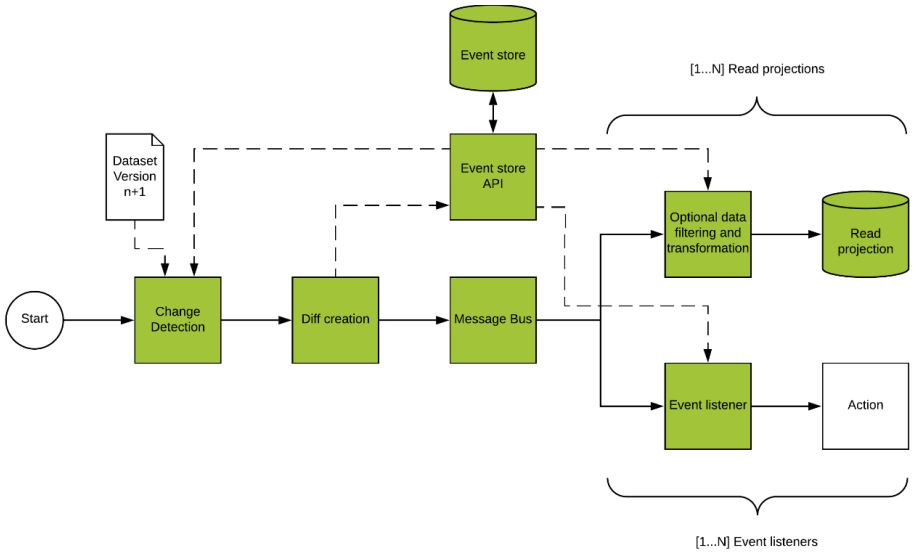

Figure 5: *Overview of the proposed event-sourced pipeline for geospatial vector data, showing the distinct components needed in order to implement the required functionality.*

From this figure, the following list of components can be derived:

- *Change Detection.* A method to detect changes to geospatial features from one version of a dataset to another.
- *Diff Creation.* A method for fast, reliable, and compact calculation and representation of changes to a geospatial feature.
- *Event Store.* A storage for events in an event-sourced system that should be able to store large amounts of events and retrieve them based on an object identifier.

- *Event Store API.* An implementation of the event sourcing mechanisms. This is where events are fetched and combined in order to retrieve the requested version of a feature.
- *Message Bus.* A mechanism for informing other systems when events occur.
- *Read projections.* A read-optimized storage for geospatial datasets.
- *Data transformers and filters.* Transformers are used to apply data transformations to geometries and attributes, such as renaming, reprojection, simplifying, and linking data. Filters are used to limit what features are written to the read projection, based on a set of pre-defined rules.
- *Event listeners.* Mechanisms for observing the event store and act when a relevant event occurs.

In addition, a method for *data conflation* is an important aspect. This process is not included in Figure 5, as the figure shows a pipeline working on a single dataset. The conflation mechanism works on a combination of two datasets.

These nine components form the building blocks of the event-based pipeline for geospatial vector data management that lies at the core of this thesis. In order to create a fully functioning pipeline, all these components have to be implemented and combined. As discussed, a public cloud computing platform is a good option for implementing this kind of pipeline. A combination of readily available components and custom code allows for a scalable system, with built in mechanisms for access control, exception handling and logging. In addition, existing solutions for deployment and orchestration can be leveraged.

While the actual implementation of this pipeline in a public cloud computing platform is outside the scope of this thesis, some work in this direction was carried out in a master's thesis by Kjelsaas (50), who implemented parts of the discussed pipeline on the Microsoft Azure cloud computing platform. However, for the purpose of this thesis, an example pipeline has been implemented. This example is implemented without taking specific cloud provider implementations into consideration and eschews several of the requirements of a production-ready pipeline in favour of readability and illustration of concepts.

The example implementation is written in the C# programming language on the .NET Core platform, and is available at https://github.com/atlefren/PhdExamplePipeline. Extracts from this implementation is referenced as individual listings in the following, to illustrate concepts and ideas. However, as the listings omit several details and examples, it is recommended to read through the provided source code. The unit tests are a good starting point. These simulates several processes of the pipeline by the use of a small example dataset and mocking of un-implemented components.

The rest of this section presents and discusses each of the pipeline components. Diff creation, storage of heterogenous geospatial data in a spatial database, and data conflation using micro-tasking are the main topics of research papers IV, III, and II, respectively. These components are discussed in greater detail, with relevant results from the research papers presented in context. Vector data change detection and pipeline implementation in a public cloud platform have been addressed partially by master students under the supervision of the candidate (50,51), while the remaining components are covered in brief, and points to possible existing solutions. All components are however illustrated

with related excerpts from the sample implementation, in order to place them in context and illustrate their intended function.

The remainder of this section is divided into four subsections. In the first subsection, the components required to convert a bulk-updated, temporal snapshot dataset into events are described. In the second subsection we examine how events are stored and distributed. The third subsection describes how geospatial events can be utilized, using read projections and event listeners. In the fourth and final subsection, we examine human-assisted data conflation using micro-tasking, which allows us to combine data from disparate datasets.

Transforming temporal snapshots to events

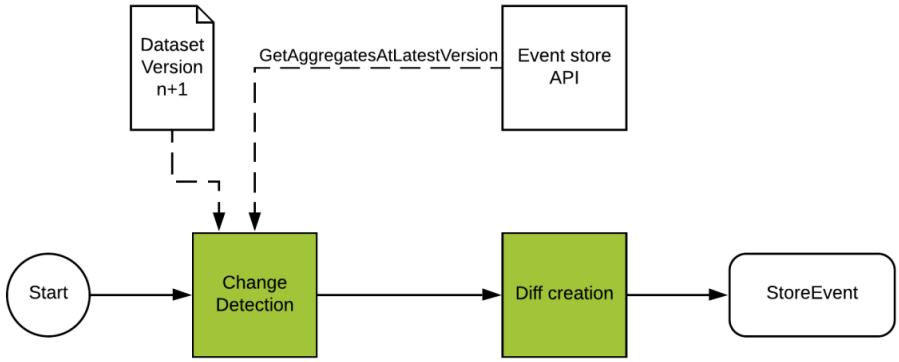

Figure 6: *The change detection and diff creation components from Figure 5 shown in more detail.*

The process of transforming a bulk-updated dataset into a stream of events required for an event sourced system consists of three operations, as shown in Figure 6. When a new version $(n + 1)$ of a dataset is delivered, the existing version (n) is retrieved from the event store. Then, the features from the current and new version are compared, and features that are changed between versions are converted into diffs. These diffs are then stored in an event store, and a message is published on a message bus. This ensures that other systems and components using the data have the ability to act on the arrival of new data, and that all changes are properly stored.

This process is implemented in the $EventSourceConverter$ class of the reference implementation. Relevant parts of this code are reproduced in Listing 1.

```
public async Task UpdateDataset(
    Guid datasetId,
    IEnumerable<Feature<TGeometry, TAttributes>> newFeatures)
{
    //Get current version (version n) of the dataset
    var oldFeatures = await _eventSourceApi.GetAggregatesAtLatestVersion(datasetId);

    //Use the new and old features to generate a list of pairs
    //with corresponding action
    var changes = await _changeDetector.FindChanges(oldFeatures, newFeatures);

    //Create a diff for each pair that is changed, created, or deleted
    var events = _featureDiffer.GetDiffs(changes).ToList();

    await Task.WhenAll(events.Select(@event
       => StoreEvent(datasetId, @event)
    ).ToArray());
}

private async Task StoreEvent(Guid datasetId, Event<FeatureDiff> @event)
{
    await _eventSourceApi.SaveEvent(@event);
    _messageBus.Publish(datasetId, @event);
}
```

Listing 1: *Extract of the EventSourceConverter class, which handles change detection, diff creation, event storage, and event messaging.*

Geospatial vector data change detection

When a new version of a bulk-updated dataset is received, each feature of the new version has to be compared to the existing data to decide what action to take. Listing 2 shows a C# interface describing this operation.

```
public interface IChangeDetector<TGeometry, TAttributes>
       where TGeometry : IGeometry
{
    public Task<IEnumerable<FeaturePair<TGeometry, TAttributes>>> FindChanges(
        IEnumerable<Aggregate<Feature<TGeometry, TAttributes>>> existingVersion,
        IEnumerable<Feature<TGeometry, TAttributes>> newVersion
    );
}
```

Listing 2: *Interface describing a change detector for geospatial vector features. Given two lists of features, the* `FindChanges` *method finds pairs of matching features and annotates them with the appropriate operation; Create, Delete, Modify, or NoOperation.*

A feature can be created, deleted, modified, or unchanged. In order to determine the correct action, we need a mechanism for change detection. In his master's thesis Zarosa (51) performed a comparison of two existing algorithms for change detection: Structural matching (39) and Hierarchical matching (40). Both algorithms were implemented and tested using real-life data from OSM. The experiment found no differences in terms of precision, but the structural matching algorithm performed marginally better in terms of accuracy. Furthermore, the structural matching algorithm was found to have an average run time that was approximately 30% slower than the hierarchical matching algorithm. However, these results may be influenced by the low number of geometries included in the experiment, and by implementation-specific details of the experiment.

Another aspect worth noting is that these algorithms only perform change detection on vector geometries. Most geospatial data also include attributes, which will have to be

considered in this context as well. In the pipeline described here, we envision combining algorithms for change detection in structured data with a change detection algorithm for geospatial vector data.

Diff creation

Once change detection is complete, the changes to each feature-pair has to be saved as an event. All un-modified features are discarded, but for created, modified, and deleted features we need an efficient format to store the diff. We found no examples applying source code version control principles to geospatial vector data in the literature, but the industry provides several examples, such as GeoGig (52), GeoDiff (53), and Sno (54).

An examination of existing literature did not yield any algorithms designed to take advantage of the mathematical properties of vector data. The commercial examples do not seem to be using any specialized algorithms either. The common approach is to apply a post- and a pre-processing step to convert geospatial vector data to a text-based or binary format, and use diffing algorithms designed for this type of data.

The GeomDiff algorithm presented in Paper IV is based on the work of Myers (55), but takes advantage of the mathematical properties of vector data. An implementation of this algorithm was created in C#. This implementation was then compared to three existing algorithms designed to diff textual, binary, and JSON data, using the beforementioned pre- and post-processing steps.

A comparison of these algorithms was carried out in a cloud computing environment, using 2.5 million real-life geometry pairs extracted from OpenStreetMap. As seen in Table 3, we found that the GeomDiff algorithm was the best choice for point geometries, but it suffered from performance degradation on linestring and polygon geometries when the vertex count approached 500. This experiment shows that a tailored algorithm for geospatial vector data is worth pursuing.

While the current GeomDiff implementation suffers from performance degradation on diff creation of polygon and linestring geometries, it performs very good on apply and undo operations, and produces patches that are small in size. Thus, finding and fixing the issues related to the performance degradation in the GeomDiff algorithm is an obvious next step when implementing the described pipeline.

Geometry Type	Algorithm	Create Time (ms)		Apply Time (ms)		Undo Time (ms)		Patch Size (b)	
		Mean	St.dev	Mean	St.dev	Mean	St.dev	Mean	St.dev
Point	TextDiff	0.22	10.92	0.47	15.64	0.32	2.32	54.0	30.0
	JsonDiff	0.38	7.21	0.21	2.51	0.16	1.62	184.0	94.0
	BinaryDiff	190.88	272.07	67.39	131.74	-	-	168.0	20.0
	GeomDiff	**0.03**	1.80	**0.02**	0.58	**0.01**	0.40	**25.0**	0.0
Linestring	TextDiff	9.01	58.56	1.00	10.98	1.04	4.61	623.44	1,733.22
	JsonDiff	**2.27**	35.96	1.12	10.08	1.06	8.23	3,064.38	9,656.37
	BinaryDiff	183.47	333.88	57.07	159.81	-	-	**357.16**	635.37
	GeomDiff	57.83	3281.33	**0.21**	8.20	**0.19**	5.22	419.63	1,355.67
Polygon	TextDiff	7.53	70.12	1.08	39.82	0.92	7.22	481.37	2,023.27
	JsonDiff	**3.50**	76.01	1.15	20.80	0.95	10.60	2,970.73	15,035.43
	BinaryDiff	224.40	571.71	69.11	272.37	-	-	**301.82**	684.04
	GeomDiff	118.09	5,159.74	**0.39**	79.77	**0.25**	7.02	306.00	1,397.86

Table 3: *Results from the geometry diff experiment. The time it took for the algorithm to create a patch, apply the patch, undo the patch, and the patch size in bytes was measured using 1,335,489 point pairs, 813,503 linestring pairs, and 433,776 polygon pairs for the GeomDiff implementation, and three other approaches. The GeomDiff algorithm performs best on all metrics for point geometries but degrades on creation on linestring and polygons.*

As noted, when discussing change detection, the distinction between geometries and features is important. The GeomDiff algorithm only operates on geometries, while the

pipeline will operate on features. However, attribute diffing can be achieved using an existing algorithm designed to handle structured data, such as JSON. This was presented in Paper IV. An example of this is provided in Listing 3, where both GeomDiff (56) and JsonDiffPatchNet (57) are used to create a diff of the complete feature.

```
public FeatureDiff Diff(
    Feature<TGeometry, TAttributes> v1,
    Feature<TGeometry, TAttributes> v2)
    => v2 == default
        ? default
        : new FeatureDiff() {
            AttributeDiff = DiffAttributes(GetAttributes(v1), v2.Attributes),
            GeometryDiff = DiffGeometry(GetGeometry(v1), v2.Geometry)
        };

public Feature<TGeometry, TAttributes> Patch(
    Feature<TGeometry, TAttributes> v1,
    FeatureDiff diff)
    => diff == default
        ? default
        : new Feature<TGeometry, TAttributes>()
        {
                Attributes = PatchAttributes(GetAttributes(v1), diff.AttributeDiff),
                Geometry = PatchGeometry(GetGeometry(v1), diff.GeometryDiff)
        };
```

Listing 3: *Relevant parts of the geospatial vector feature differ implemented in the* `FeatureDiffPatch` *class. A* `FeatureDiff` *object consists of two separate properties,* `AttributeDiff` *and* `GeometryDiff`*, which are handled by GeomDiff and JsonDiffPatchNet, respectively.*

Event storage

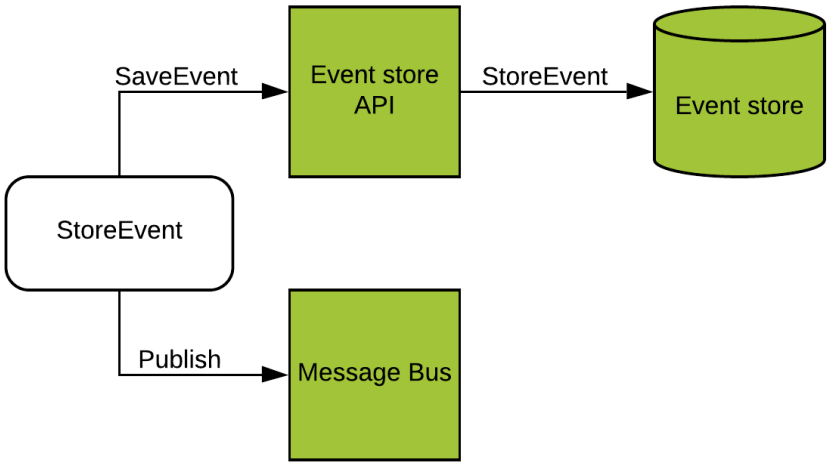

Figure 7: *Excerpt from Figure 5, showing the components needed to store and distribute events.*

When changes between versions are detected and efficient diffs are created, the events need to be stored, and other systems have to be notified. This process involves the three components shown in Figure 7; the event store and the event store API, and the message bus. These components are described in the following.

Event store

The event store mechanism is, at its core, a storage for events. A combination of events related to the same feature is referred to as an *aggregate*, and this aggregate has a unique identifier. Each event related to an aggregate has its own version number, which is incremented when a new event for an aggregate is stored. The event store should support storing an event, consisting of an aggregate identifier, a version number, and the actual event data (32). In our case, the event data is the feature diff created in the event differ. Mechanisms for storing and retrieving events are core features of the event store, as outlined in Listing 4. In addition, we need to ensure consistency, i.e. that every event is stored only once. This can be achieved using indices or locks. The event storage can be implemented using a traditional RDBMS, or more modern NoSQL databases. The key point is that we need to ensure consistency and be able to retrieve events based on version number and aggregate id.

```
public interface IEventStorage<TEventData>
{
    Task<IEnumerable<Guid>> GetAggregatesForDataset(Guid datasetId);
    Task<IEnumerable<Event<TEventData>>> GetEventsForAggregate(
       Guid datasetId,
       Guid aggregateId
       );
    Task StoreEvent(Guid datasetId, Event<TEventData> @event);
}
```

Listing 4: *Interface describing an event store. This interface supports retrieving all aggregate ids for a dataset id, retrieving all events for an aggregate, and storing an event. The interface is generic, as the actual contents of* `TEventData` *will change depending on the data stored.*

Event store API

The event store API should handle retrieving features from and storing events to the event store. To make our pipeline work, we need at least three methods; a way to save an event, a way to retrieve all features in a dataset in their latest version, and a way to retrieve the latest version of a specific feature given its ID. Listing 5 provides a minimal interface of an event store API.

```
public interface IEventStoreApi<TData, TDiff>
{
    Task SaveEvent(Event<TDiff> @event);
    Task<IEnumerable<Aggregate<TData>>> GetAggregatesAtLatestVersion(Guid datasetId);
    Task<Aggregate<TData>> GetAggregateAtLatestVersion(Guid aggregateId);
}
```

Listing 5: *The `IEventStoreApi` used in the example pipeline. The interface describes three asynchronous methods, for saving an event and to retrieve all features and a single feature in their latest version.*

This API can also be extended to include methods for retrieving features at a specific version, or from a specific point in time.

The actual implementation of the event store API is responsible for applying all events related to a feature in sequence, in order to arrive at the requested version. This process is illustrated in Listing 6.

```
public async Task<Aggregate<TData>> GetAggregateAtLatestVersion(
    Guid datasetId,
    Guid aggregateId)
    => (await _eventStorage.GetEventsForAggregate(datasetId, aggregateId))
        .OrderBy(e => e.Version)
        .Aggregate(default(Aggregate<TData>), ApplyEvent);

private Aggregate<TData> ApplyEvent(
    Aggregate<TData> aggregate,
    Event<TDiff> @event)
    => new Aggregate<TData>()
    {
        Data = _differ.Patch(aggregate.Data, @event.EventData),
        Id = aggregate.Id,
        Version = @event.Version
    };
```

Listing 6: *The GetAggregateAtLatestVersion method of the EventStoreApi class.*

The Running time of the $GetAggregateAtLatestVersion$ method is given as $n \times x$, where n is the number of events and x running time of the $ApplyEvent$ method. In order to reduce the running time, rolling snapshots (32) can be utilized.

The essence of rolling snapshot is to save the current representation of a feature at regular intervals. This reduces n, the number of events $GetAggregateAtLatestVersion$ have to loop trough. If a feature has 1010 stored versions, the running time would be $1010 \times x$. When using rolling snapshots at every 100 versions, the running time would be $10 \times x$, as we can retrieve the snapshot of version 1000, and apply ten events.

Event message bus

In Listing 1, we note that each changed feature is distributed as a message on a message bus. This message bus serves as a means to communicate to other systems and users that an event they might be interested in has occurred.

The *IMessageBus*-interface presented in Listing 7 shows what we expect from a message bus. We need a way to publish events, and a way to subscribe to events. For the example pipeline, we have chosen to keep this as a simple, in-memory solution. In practice, this component should be a shared and scalable component that can be accessed by several systems. This is an offering in most cloud platforms, and using an off-the shelf component for this ensures a fault-tolerant and resilient system (58).

```
public interface IMessageBus<TEventData>
{
    void Subscribe(Guid datasetId, Action<Event<TEventData>> callback);

    void Publish(Guid datasetId, IEnumerable<Event<TEventData>> events);

    void Publish(Guid datasetId, Event<TEventData> @event);
}
```

Listing 7: *IMessageBus interface describing methods an implementation of a massage bus class should implement.*

An important consideration is to ensure that the write to the event store and the publication to the message bus is executed as a two-phase commit. If not, situations where the event is published, but not written to storage, could occur (32). In our example, this means that the *StoreEvent* method in Listing 1 should be an all or nothing operation. If the publication of the message on the message bus fails, the whole operation should fail.

Use of event sourced datasets

So far, we have shown how an event sourced dataset can be created and stored, and how other systems and components can be notified when an event occurs. But how do we use this data? The event store API offers methods for reading data from the event store and is in many cases a viable solution for retrieving data. However, by utilizing the event store and the message bus, several other mechanisms for using an event sourced geospatial dataset can be created. Read projections and event listeners are two methods for utilizing an event sourced dataset.

Read projections

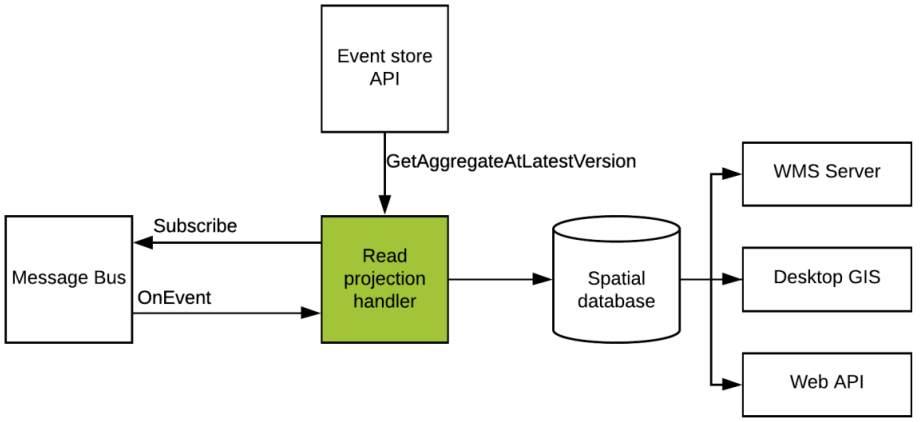

Figure 8: *The concept of read projections, showing how the system interacts with the message bus and event store API in order to retrieve feature changes, apply filters and transformations to them, and store them in a spatial database. The figure also includes some possible uses of the read projections, such as a WMS server, a Desktop GIS, or a web API.*

Read projections is a method to simplify data access in an event sourced system. The concept allows for faster data access and enables use of traditional tools which require a RDBMS or specialized formats. In addition, the use of read projections enables data filtering and transformation, as shown in Figure 8.

An important aspect of read projections is that a read projection is a read-only replica of the original dataset. In an event sourced system, the only way to update data is to dispatch an event. This means that update conflicts, the main reason for avoiding data redundancy, does not pose a problem. Thus, an event sourced system combined with read projections allows for creation of several read-only copies of the data. This simplifies horizontal scaling, which can increase the performance and stability of the system. In addition, it enables us to tailor the stored data to each application, by using data filtering and transformation, as described in the next subsection.

When working with spatial data, a spatial database is a candidate for storing read projections. Desktop GIS and WMS or tile servers usually provide mechanisms for reading from spatial databases out-of-the-box. In addition, spatial databases often provide a set of geospatial operations that can be used to query the data. However, how feature attributes and geometries are stored in a spatial database can influence both usability and performance of the system.

One promising technique are NoSQL (or "Not only SQL") databases, which emerged in the late 2000s (59), as a way to handle "big data" challenges. Document stores is one class of NoSQL systems, which store records as documents with no pre-defined schema. In Paper III, we compared the read and write performance of a traditional, schema-based layout with one table per dataset and a document-store, no-schema, single store for all datasets using the jsonb datatype in the PostgreSQL database system.

Storage layout	Import		Query – intersect		Query – intersect + attribute		Disk size
	Speed (m)	SD	Speed (s)	SD	Speed (s)	SD	(GB)
Table-based	***79***	*3.57*	***19***	*0.54*	***99***	*3.24*	***12.29***
Jsonb	*179*	*4.50*	*25*	*1.00*	*162*	*3.30*	*17.50*

Table 4: *Main findings of Paper III. The table-based layout was found to be faster both in terms of import speed and query speed. In addition, the storage footprint of the table-based layout was smaller.*

As can be seen in Table 4, we found that a one-table-per dataset layout, with explicit columns for each attribute, outperformed a large document-store combined with a geometry column, both in terms of insert and query speed, as well as in required storage space.

In light of these findings, we propose to base the read projections in the pipeline on a table-based layout. However, the premise of the HOGS system presented in Paper III was to import bulk-updated datasets from file formats. The HOGS system is implemented in Python, while we envision the proposed pipeline to be implemented in C#. This means that the HOGS system cannot be directly plugged into the pipeline, but the findings from the paper can be incorporated into the pipeline architecture.

An illustration of how a read projection can be managed is given in the *ReadProjectionWriter* class. Listing 8 shows the *CreateReadProjection* method, which handles creation of a table and subscribes to the message bus in order to receive new events.

```
public async Task CreateReadProjection(Guid datasetId)
{
    var tableName = GetTableName(datasetId);
    await _databaseEngine.CreateTable(tableName, GetColumns());
    await InsertExistingFeatures(datasetId, tableName);

    _messageBus.Subscribe(datasetId, @event =>
    {
        Update(datasetId, tableName, @event);
    });
}
```

Listing 8: *The CreateReadProjection method of the ReadProjectionWriter class. This method creates a table to store data in, populates it with any existing features, and then subscribes to the message bus in order to receive new events.*

Data filtering and transformation

In many cases, we want to go further than establishing a replica. By establishing a read projection for each application using the data, we ensure that increased load on one application does not influence performance of other apps. Since each read projection is now used exclusively by a single application, we are free to process the incoming data to fit the needs of the application. This can be achieved using pre-processing steps. Filtering and transformation are two such pre-processing steps, both shown in the *Upsert* method in Listing 9.

```
private async Task Upsert(
    Guid datasetId,
    string tableName,
    Aggregate<Feature<TInputGeometry, TInputAttributes>> aggregate)
{
    if (!await ShouldKeep(datasetId, aggregate.Data)) {
        return;
    }
    var transformedFeature = _transformFeature(aggregate.Data);
    var rowData = GetRowData(aggregate.Id, transformedFeature);
    await _databaseEngine.Upsert(tableName, rowData);
}
```

Listing 9: *The Upsert method in CreateReadProjection, which performs filtering, transformation, and maps the feature to a database row, before effectuating the insert or update statement.*

Data filtering is a method for limiting which features are written to the read projection. As can be seen in Listing 9, if the *ShouldKeep* method does not indicate that a feature should be kept it is skipped and will never arrive at the read projection database. The actual *ShouldKeep* method is passed as a parameter to the read projection writer, which means that the implementation can be tailored to the dataset in question, as long as it matches the signature provided in Listing 10.

```
Func<Guid, Feature<TInputGeometry, TInputAttributes>, Task<bool>> _shouldKeepFeature;
```

Listing 10: *Signature of the ShouldKeep method. This is an async method, which takes a dataset id and a feature as input and returns a Boolean, indicating if the feature should be kept or not.*

Data transformation is, as mentioned, the task of transforming the data from the event store before it is stored in the read projection. This is useful if there is a need to re-project the geometries or alter them in some other way (i.e. converting a polygon into its centroid, simplifying the geometry, or adding a buffer), or if the attribute schema needs to be changed in any way. This transformation can be implemented by passing a function to the *CreateReadProjection* object, which then calls this function before the feature is written to the database. The *Upsert* method in Listing 9 shows this in context.

Event listeners

To keep a read projection in sync with the event store, we subscribe to messages on a message bus, as seen in Figure 9. This process can be enhanced using data filtering and transformations. However, the use of event listeners and a message bus to build a representation of the dataset that can be queried in a traditional manner is just one example of their usage.

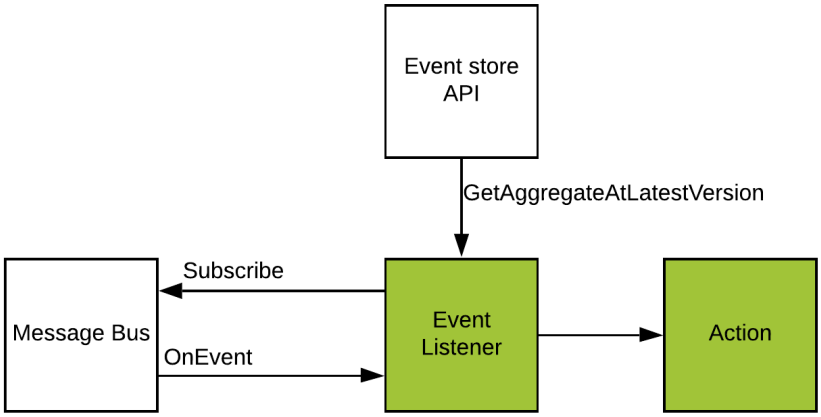

Figure 9: *An illustration of the concept of event listeners.*

Event listeners also allow us to use the events themselves to trigger processes and actions. One example is to listen for events on a flood-risk dataset, and automatically notify house owners via e-mail when the flood-risk of their property is updated. A related example is to use the same events to trigger a re-calculation of insurance premiums whenever the underlying data is changed.

Data conflation

Data conflation is the task of merging geospatial features from separate datasets which maps the same concepts. Level of detail, accuracy, and what features are mapped usually vary between datasets, depending on the purpose of the survey, and the resources available. Figure 10 (reproduced from Paper II) shows an aerial photograph of a building (left) and two building outlines (middle and right) digitized from this image. It should be relatively easy for a human observer to determine that the outline in the middle image covers the entire building, while the one in the right image follows the ridge line of the roof and is thus missing a section.

Figure 10: *Aerial imagery of a building (left) and two building outlines digitized from this image (middle and right). The purple digitization (right) is incomplete, as it follows the ridge line rather than the outline. Figure from Paper II.*

When conflating spatial features from two datasets we want to be able to select the best representation in each case. As illustrated, this task is usually relatively simple for a human, but can be difficult to achieve for an algorithm. In Paper II, we examined how to effectively utilize the micro-tasking approach to assist in data conflation.

An online experiment, simulating part of an OpenStreetMap import of building footprints was carried out. The experiment was designed to test how and if task partitioning and

prior experience working with geospatial data influenced time spent on a task and task accuracy. The experiment ran for 8 days, with a total of 164 participants completing at least one task.

		No. of obs.	Sample mean	St.dev
Total time on 6 assessments (s)	All	427	100.05	50.99
	Experienced	234	102.69	54.94
	Inexperienced	193	96.85	45.68
Number of correctly chosen footprints	All	427	4.93	1.03
	Experienced	234	**5.02**	1.02
	Inexperienced	193	4.83	1.04
Total time on 6 assessments (s)	1 assessment / task	146	95.99	51.90
	3 assessments / task	144	99.79	52.80
	6 assessments / task	137	104.66	47.97
Number of correctly chosen footprints	1 assessment / task	146	4.96	0.96
	3 assessments / task	144	4.95	1.07
	6 assessments / task	137	4.89	1.08

Table 5: *Main findings of Paper II. The upper half presents results of experience level, while the bottom section presents effects of task partitioning. Statistical analysis found that experienced participants performed marginally better in terms of accuracy, while none of the other metrics show any statistically significant differences.*

The main findings of the experiment are summarized in Table 5. We found that prior experience did not affect task completion time, but participants with prior experience performed minimally better in terms of accuracy. When examining task partitioning, we

found no indication that this influenced neither completion time nor accuracy. However, these results show that all participants achieved a high degree of accuracy, the average participant identified 82% of the correct building footprints, with an average of 16 seconds spent on each assessment.

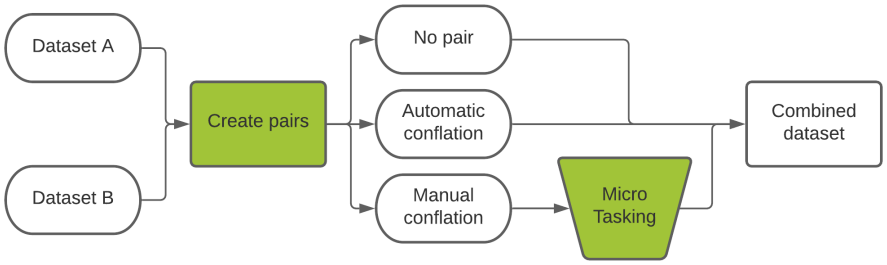

Figure 11: *A conflation process for two geospatial datasets employing micro-tasking.*

Incorporating a micro-tasking process into an automated pipeline poses a challenge, as this is an asynchronous process that relies on the availability of humans (42). However, a cloud-based architecture using message-passing to handle long-running processing times (58) is a viable approach to handle the asynchronous nature of human tasks. The conflate method outlined in Listing 11 provides an overview of how this can be achieved. This process is also outlined in Figure 11.

```
public IEnumerable<Feature<TGeometry, TAttributesOut>> Conflate(
    IEnumerable<Feature<TGeometry, TAttributesA>> datasetA,
    IEnumerable<Feature<TGeometry, TAttributesB>> datasetB)
{
    var pairs = GetPairs(datasetA, datasetB);
    var toConflate = pairs.Where(p => p.NeedsConflation());
    foreach (var pair in toConflate)
    {
        _conflatorQueue.AddConflationTask(pair.A, pair.B);
    }

    return pairs
        .Where(p => !p.NeedsConflation())
        .Select(p => p.GetFeature(_mapAttributesA, _mapAttributesB));
}
```

Listing 11: *The conflate method in the `DataConflator` class. This method receives features from two datasets, with possibly different attribute schemas, and returns those which does not require manual comparison. This filtering is done using the provided `_requiresConflation` methods, which compares two features to check if they are similar enough to warrant manual inspection. Features which does not require manual inspection are returned, while feature pairs which require conflation are added to a `_conflatorQueue` for manual inspection.*

Features from the two datasets are compared using a provided function, and those in need of manual inspection are returned as pairs. These feature-pairs are passed as messages to an asynchronous queue, which in turn can present them as tasks to a crowdsourcing-worker. The user interface for this task and how the conflated features are returned to the system are omitted from the example application.

Concluding remarks

This thesis argues that the principle of event sourcing is a viable and effective solution to many of the problems related to managing a large amount of heterogenous spatial datasets. By leveraging the services available through a public cloud provider, a scalable, resilient, and performant solution can be created. In addition, I have shown how data from different datasets can be combined through the use of micro-tasking, and how data can be used through read projections and event listeners.

Many of the findings in this thesis is the result of applying ideas and findings from computer science and the software industry to the geospatial domain. These domains are by no means separate magisteria, as witnessed by the digitization of the map brought forward by the digital revolution. However, as the debate over whether spatial is special or not shows, there is a divide between these areas of research. This thesis is an attempt to build yet another bridge over this gap, and to strengthen the connection made by previous researchers and professionals.

Concepts from the software industry can often be applied to geospatial problems, but they often require some modification to perform optimally. The GeomDiff algorithm presented in Paper IV is a prime example of this. The concept of diffing dates back nearly fifty years, but no specialized algorithms for diffing geospatial data was found in the literature. Thus, identifying concepts from other fields of research and adapting them to the problem at hand has proven an effective path towards progress.

This thesis is not the final chapter written on event sourcing of geospatial datasets. The work leaves several open threads. An actual implementation of the pipeline in a public

cloud computing environment is an obvious next step. Another, unsolved, issue is geospatial vector data change detection. Some solutions do exist, but none of them are perfect, and none addresses the problem of identifying changes to features, as opposed to geometries. Since this thesis is written as part of an industrial PhD programme, these challenges are logical next steps for Norkart AS, who is funding this work.

While these issues are both interesting and challenging, the scope of a thesis has to be limited somewhere. I hope to continue working on these issues, and that others will join in. Nevertheless, the results presented herein establishes that an event based pipeline is a viable solution to the problem of managing an increasing stream of heterogenous geospatial vector data.

References

1. Goodchild MF, Egenhofer MJ, Kemp KK, Mark DM, Sheppard E. Introduction to the Varenius project. International Journal of Geographical Information Science. 1999 Dec 1;13(8):731–45.

2. Veenendaal B, Brovelli MA, Li S. Review of Web Mapping: Eras, Trends and Directions. ISPRS International Journal of Geo-Information. 2017 Oct;6(10):317.

3. Byrne D, Pickard AJ. Neogeography and the democratization of GIS: a metasynthesis of qualitative research. Information, Communication & Society. 2016 Nov 1;19(11):1505–22.

4. Haklay M, Weber P. OpenStreetMap: User-Generated Street Maps. IEEE Pervasive Computing. 2008 Oct;7(4):12–8.

5. Haklay M. How good is volunteered geographical information? A comparative study of OpenStreetMap and ordnance survey datasets. Environment and Planning B: Planning and Design. 2010;37(4):682–703.

6. Goodchild MF. Citizens as sensors: The world of volunteered geography. GeoJournal. 2007;69(4):211–21.

7. Open Knowledge International. Open Definition 2.1 [Internet]. 2016 [cited 2017 Jan 12]. Available from: http://opendefinition.org/od/2.1/en/

8. Deloitte. The impact of the open geographical data. 2014.

9. US Copyright Office. Copyright Law of the United States and Related Laws Contained in Tıtle 17 of the United States Code. Government Printing Office; 2012.

10. Worboys M. Event-oriented approaches to geographic phenomena. International Journal of Geographical Information Science. 2005 Jan 1;19(1):1–28.

11. Cooper A, Peled A. Incremental updating and versioning. In: 20th International Cartographic Conference. Beijing; 2001. p. 2804–9.

12. Fowler M. Event Sourcing [Internet]. martinfowler.com. 2005 [cited 2020 Apr 16]. Available from: https://martinfowler.com/eaaDev/EventSourcing.html

13. Oppenheimer D. Machine Learning with Humans in the Loop [Internet]. Algorithmia Blog. 2017 [cited 2018 Nov 9]. Available from: https://blog.algorithmia.com/machine-learning-with-human-in-the-loop/

14. Biewald L. Why human-in-the-loop computing is the future of machine learning [Internet]. Computerworld. 2015 [cited 2018 Nov 9]. Available from:

https://www.computerworld.com/article/3004013/robotics/why-human-in-the-loop-computing-is-the-future-of-machine-learning.html

15. Erichsen ASS. Evaluation of the Micro-Tasking Method for Importing High-Detail Building Models to OpenStreetMap. Trondheim: NTNU; 2016.

16. Sveen AF, Erichsen ASS. Evaluation of the Micro-Tasking Method for OpenStreetMap Imports. Free and Open Source Software for Geospatial (FOSS4G) Conference Proceedings [Internet]. 2017 Sep 22;17(1). Available from: https://scholarworks.umass.edu/foss4g/vol17/iss1/20

17. Nossum AS. The best of both worlds: combining geometry and key-value stores using PostGIS and HStore — Alexander Nossum, Norkart AS [Internet]. FOSS4G; 2014 Sep 10; Portland, Oregon, USA. Available from: https://vimeo.com/106223530

18. Lessons from applying the systematic literature review process within the software engineering domain. Journal of Systems and Software. 2007 Apr 1;80(4):571–83.

19. Kortum PT, Bangor A. Usability Ratings for Everyday Products Measured With the System Usability Scale. International Journal of Human–Computer Interaction. 2013 Jan 1;29(2):67–76.

20. Mell P, Grance T. The NIST definition of cloud computing. 2011;

21. Shilkov M. From 0 to 1000 Instances: How Serverless Providers Scale Queue Processing [Internet]. Binaris Blog. 2018 [cited 2020 May 8]. Available from: https://blog.binaris.com/from-0-to-1000-instances/

22. Martin RC. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education; 2008. 464 p.

23. Thomas D, Hunt A. The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition. Addison-Wesley Professional; 2019. 372 p.

24. Pandas [Internet]. Available from: https://pandas.pydata.org

25. SciPy developers. SciPy [Internet]. Available from: https://www.scipy.org/scipylib/index.html

26. NumPy Developers. NumPy [Internet]. Available from: https://numpy.org

27. Laney D. 3D data management: Controlling data volume, velocity and variety. META Group Research Note. 2001;6(70).

28. Stats - OpenStreetMap Wiki [Internet]. [cited 2020 Jul 22]. Available from: https://wiki.openstreetmap.org/wiki/Stats

29. Thill J-C. Is Spatial Really That Special? A Tale of Spaces. In: Popovich VV, Claramunt C, Devogele T, Schrenk M, Korolenko K, editors. Information Fusion and Geographic Information Systems: Towards the Digital Ocean [Internet]. Berlin, Heidelberg: Springer; 2011 [cited 2020 Jul 14]. p. 3–12. (Lecture Notes in Geoinformation and Cartography). Available from: https://doi.org/10.1007/978-3-642-19766-6_1

30. Vector drivers — GDAL documentation [Internet]. [cited 2020 Jul 22]. Available from: https://gdal.org/drivers/vector/index.html

31. Seto T, Sekimoto Y. Comparing the Distribution of Open Geospatial Information between the Cities of Japan and Other Countries. In: Ferreira J, Goodspeed R, editors. CUPUM 2015 [Internet]. Cambridge, Massachusetts; 2015. Available from: http://web.mit.edu/cron/project/CUPUM2015/proceedings

32. Young G. CQRS documents. Technical report, cqrsinfo.com; 2010.

33. Debski A, Szczepanik B, Malawski M, Spahr S, Muthig D. A scalable, reactive architecture for cloud applications. IEEE Software. 2018 Mar;35(2):62–71.

34. Miller W, Myers EW. A file comparison program. Software: Practice and Experience. 1985;15(11):1025–40.

35. Hunt JW, MacIlroy MD. An algorithm for differential file comparison. Bell Laboratories Murray Hill; 1976. (Bell Laboratories Computing Science). Report No.: #41.

36. Open Knowledge International. Global Open Data Index [Internet]. 2017 [cited 2018 Nov 1]. Available from: http://index.okfn.org

37. Lu D, Mausel P, Brondízio E, Moran E. Change detection techniques. International Journal of Remote Sensing. 2004 Jun 1;25(12):2365–401.

38. Chawathe SS, Rajaraman A, Garcia-Molina H, Widom J. Change Detection in Hierarchically Structured Information. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data [Internet]. New York, NY, USA: ACM; 1996 [cited 2019 Aug 27]. p. 493–504. (SIGMOD '96). Available from: http://doi.acm.org/10.1145/233269.233366

39. Gombos˘i M, Z˘alik B, Krivograd S. Comparing two sets of polygons. International Journal of Geographical Information Science. 2003 Jun 1;17(5):431–43.

40. Wang Y, Zhang Q, Guan H. Incrementally Detecting Change Types of Spatial Area Object: A Hierarchical Matching Method Considering Change Process. ISPRS International Journal of Geo-Information. 2018 Jan;7(2):42.

41. Mooney P, Corcoran P. The Annotation Process in OpenStreetMap. Transactions in GIS. 2012;16(4):561–79.

42. Difallah DE, Catasta M, Demartini G, Ipeirotis PG, Cudré-Mauroux P. The Dynamics of Micro-Task Crowdsourcing. In: Proceedings of the 24th International Conference on World Wide Web - WWW '15 [Internet]. New York, New York, USA: ACM Press; 2015 [cited 2017 Jul 7]. p. 238–47. Available from: http://dl.acm.org/citation.cfm?doid=2736277.2741685

43. Ipeirotis PG. Analyzing the Amazon Mechanical Turk marketplace. XRDS: Crossroads, The ACM Magazine for Students. 2010 Dec 1;17(2):16.

44. Barth A. Importing 1 million New York City buildings and addresses [Internet]. 2014 [cited 2017 Feb 22]. Available from: http://www.openstreetmap.org/user/lxbarth/diary/23588

45. Schleuss J, Ureta O, McConchie A, Sambale M. Let's get LA on the map!: The Los Angeles Building Import Case Study. In: State Of The Map US [Internet]. Seattle; 2016. Available from: http://stateofthemap.us/2016/lets-get-la-on-the-map/

46. Juhász L, Hochmair HH. OSM Data Import as an Outreach Tool to Trigger Community Growth? A Case Study in Miami. ISPRS International Journal of Geo-Information. 2018;7(3):113.

47. Sugumaran R, Armstrong MP. Cloud Computing. In: International Encyclopedia of Geography [Internet]. American Cancer Society; 2017 [cited 2020 Jul 31]. p. 1–4. Available from: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118786352.wbieg1017

48. McGrath G, Brenner PR. Serverless Computing: Design, Implementation, and Performance. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). 2017. p. 405–10.

49. García López P, Sánchez-Artigas M, París G, Barcelona Pons D, Ruiz Ollobarren Á, Arroyo Pinto D. Comparison of FaaS Orchestration Systems. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). 2018. p. 148–53.

50. Kjelsaas LF. A cloud-based pipeline for Event Sourcing of geospatial data [Master's Thesis]. [Trondheim, Norway]: NTNU; 2020.

51. Zarosa B. The detection of changes in OSM spatial objects. A comparison of two methods [Master's Thesis]. [Trondheim, Norway]: NTNU; 2020.

52. Marin J. Redefining Geospatial Data Versioning: The GeoGig Approach [Internet]. 2014 [cited 2020 Mar 18]. Available from: https://www.directionsmag.com/article/1329

53. Razmjooei S. Tracking, calculating and merging vector changes with Input and QGIS - Lutra Consulting [Internet]. Lutra Consulting. 2019 [cited 2020 Mar 20]. Available from: https://www.lutraconsulting.co.uk/blog/2019/11/23/input-geodiff/

54. Coup R. Sno, our new open source tool for distributed data versioning [Internet]. Koordinates Blog. 2020. Available from: https://koordinates.com/blog/sno-our-new-open-source-tool-distributed-data-versioning/

55. Myers EW. AnO(ND) difference algorithm and its variations. Algorithmica. 1986 Nov 1;1(1):251–66.

56. Sveen AF. GeomDiff — an algorithm for differential geospatial vector data comparison. Open Geospatial Data, Software and Standards. 2020 Jul 10;5(1):1–11.

57. Bishop W. JsonDiffPatch.Net [Internet]. Available from: https://github.com/wbish/jsondiffpatch.net

58. Sousa TB, Ferreira HS, Correia FF, Aguiar A. Engineering Software for the Cloud: Messaging Systems and Logging. In: Proceedings of the 22nd European Conference on Pattern Languages of Programs [Internet]. New York, NY, USA: Association for Computing Machinery; 2017 [cited 2020 Aug 17]. p. 1–14. (EuroPLoP '17). Available from: https://doi.org/10.1145/3147704.3147720

59. Cattell R. Scalable SQL and NoSQL data stores. ACM SIGMOD Record. 2011 May 6;39(4):12.

Papers

Paper I

The Open Geospatial Data Ecosystem

Sveen, Atle Frenvik. The Open Geospatial Data Ecosystem.
Kart og plan 2017 ;Volum 77, årg. 110.(2) s. 108-120.

Not included due to copyright restrictions.

Paper II

Micro-tasking as a method for human assessment and quality control in a geospatial data import

Sveen, Atle Frenvik; Erichsen, Anne Sofie S.; Midtbø, Terje. Micro-tasking as a method for human assessment and quality control in a geospatial data import. Cartography and Geographic Information Science 2020 ;Volum 47.(2) s. 141-152

Available at: http://dx.doi.org/10.1080/15230406.2019.1659187

Paper III

Efficient storage of heterogeneous geospatial data in spatial databases

Sveen *J Big Data* (2019) 6:102
https://doi.org/10.1186/s40537-019-0262-8

Journal of Big Data

RESEARCH Open Access

Efficient storage of heterogeneous geospatial data in spatial databases

Atle Frenvik Sveen*

*Correspondence:
atle.f.sveen@ntnu.no
Norwegian University
of Science and Technology,
Trondheim, Norway

Abstract

The no-schema approach of NoSQL document stores is a tempting solution for importing heterogenous geospatial data to a spatial database. However, this approach means sacrificing the benefits of RDBMSes, such as existing integrations and the ACID principle. Previous comparisons of the document-store and table-based layout for storing geospatial data favours the document-store approach but does not consider importing data that can be segmented into homogenous datasets. In this paper we propose "The Heterogeneous Open Geodata Storage (HOGS)" system. HOGS is a command line utility that automates the process of importing geospatial data to a PostgreSQL/PostGIS database. It is developed in order to compare the performance of a traditional storage layout adhering to the ACID principle, and a NoSQL-inspired document store. A collection of eight open geospatial datasets comprising 15 million features was imported and queried in order to compare the differences between the two storage layouts. The results from a quantitative experiment are presented and shows that large amounts of open geospatial data can be stored using traditional RDBMSes using a table-based layout without any performance penalties.

Keywords: NoSQL, Document-store, Geospatial data, Spatial database, Relational database, Database benchmark

Introduction

New sources of geospatial data, such as the Internet of Things (IoT), Volunteered Geographic Information (VGI), and Open Geospatial Data, are becoming increasingly popular. This shift creates a demand for new ways to collect, manage, store, and analyse geospatial data. These challenges are mirrored in the general computer science concept of *big data*, a term describing datasets that are too large to be managed and processed by traditional technologies [1].

Laney [2] characterizes *big data* using the 3 Vs; Volume, Velocity, and Variety. These properties relate to geospatial data as well. Massive geospatial datasets originating from sensors are characterized by both high Volume and high Velocity, and open geospatial datasets from disparate sources comes with a high degree of Variety. This means that *geospatial big data* can be treated as a subset of *big data*, and opens up the possibility of using *big data* techniques to handle geospatial data [3, 4]. NoSQL (or Not Only SQL) data stores is one proposed solution to some of the challenges posed by *big data*. These data stores offer ways to handle the 3 Vs utilizing new techniques and architectures.

However, most new technology is no silver bullet. The promises of NoSQL may seem tempting, but there are several negative consequences of this approach as well. Chandra [5] uses the acronym Basically Available, Soft state, Eventual consistency (BASE) to describe NoSQL databases and contrast them with the ACID principle of relational databases. BASE also points to some of the drawbacks of NoSQL databases, such as the possibility of temporary inconsistencies. Another aspect is the lack of a universal query language. In light of this, we want to heed the advice from Stonebraker and Hellerstein [6] and examine if we really need to abandon the principles of Relational Database Management Systems (RDBMSes). In particular, we want to investigate if a combination of automated import routines and RDBMSes can offer the same advantages as NoSQL solutions when it comes to management and storage of heterogenous geospatial data.

In order to achieve this, we have implemented the Heterogeneous Open Geodata Storage (HOGS) system. This is a command line utility, written in Python, that leverages the open source GDAL/OGR geospatial library to automate imports of heterogenous geospatial data to a PostgreSQL/PostGIS database. By using both a traditional relational database layout and a NoSQL document-store layout we are able to benchmark both the import and query performance of the two storage layouts.

Background

RDBMSes dates back to the 1970′s [6], and Spatial database systems has been a term for about 30 years [7]. Today several of the best-known RDBMSes offer spatial capabilities according to the OGC Simple Feature Access specification. These spatial capabilities are often provided through an extension, such as PostGIS for PostgreSQL or Oracle Spatial for Oracle. In this paradigm, data types for spatial geometries are available alongside traditional data types and special SQL operators are available for spatial queries and operations. This means that a geometry can be treated as a normal column in a relational database table [8].

NoSQL data stores emerged in the late 2000 along with the "Web 2.0" movement [9]. The rise of these "not only SQL" systems was triggered by the need to handle "big data", or datasets that are too large to be managed and processed by traditional technologies [1]. This typically involves sacrificing or weakening the Atomicity, Consistency, Isolation, and Durability (ACID) principle underlining traditional RDBMSes [10].

There is no entirely agreed upon definition of NoSQL, but Cattell [9] offers six key features of such systems:

- Horizontal scaling.
- Replication and distribution over many servers.
- Simple call interface.
- Weakening of the ACID principle.
- Distributed indexes and RAM.
- The ability to add new attributes to records dynamically.

NoSQL data stores can also be categorized by capabilities and intended uses. Ameya et al. [11] presents five different types of NoSQL data stores; Key-value stores, column-oriented databases, document-stores, graph databases, and object-oriented databases.

The most interesting NoSQL data store type in the context of collections of open geospatial data is document-stores, with two well well-known examples being MongoDB and CouchDB. Document-stores store data as documents, reminiscent of records in a relational database, but without a pre-defined schema. Each document in the store has its own structure, and can include nested structures. A unique key is used for indexing the documents, which are usually stored using standard formats such as JSON (JavaScript Object Notation) or Extensible Markup Language (XML). The "no schema" approach of document-stores makes them popular to web developers. Partly due to their facilitation of quick integration of data from different sources, but also because they reduce the need for up-front database schema design [12].

These properties also make document-stores interesting for working with collections of open geospatial data. Such datasets originates from disparate sources and uses different file formats, coordinate systems, and attribute schemas [13]. Collecting open geospatial datasets in a traditional RDBMS requires a lot of work related to schema design and data import, where both attributes and geometries potentially have to be mapped, translated, and converted. The prospect of a "no schema"-solution that enable easy import of heterogenous datasets from a wide array of sources is intriguing. Maintaining an up-to-date collection of open geospatial data carries a lot of potential for developing value-added services and analyses, and the premise of NoSQL document-stores is that this can be achieved with less overhead. Both MongoDB and CouchDB offer spatial capabilities, using the JSON-based GeoJSON standard [14].

Another approach to tap into the benefits of a document-store is using an RDBMS that implements a document-store datatype. In these systems, a JSON or XML datatype with support for indexing and querying is made available to the RDBMS user. A document-based JSON storage type is implemented by several well-known RDBMSes, such as MySQL, Oracle, and PostgreSQL [15, 16]. These solutions have proved comparable to the NoSQL data-stores. For instance, Linster [17] reports a benchmark where the PostgreSQL document-store outperformed MongoDB on selecting, loading, and inserting a complex document dataset consisting of 50 million records.

Related work

Examples and benchmarks of NoSQL document-store datatypes for storing geospatial data are scarce in the existing literature. In the following we review the studies that most closely resembles the work we present.

A preliminary study by Navarro-Carrión et al. [18] examined the feasibility of using a NoSQL document-store to store EU land cover and land use data. In their experimental set-up, they used two PostgreSQL/PostGIS instances. One implemented a relational model, while the other implemented a NoSQL document-store model. Using these instances, they evaluated the query times of a bounding box search clause iteratively run using varying cell sizes. Using a dataset of more than 10.4 million soil occupation observations for roughly 2.5 million polygon geometries, they found that the document-oriented model was about 19% faster than the relational model. The authors point out that for several workflows a document-oriented model should be considered, and specifically points to massive polygon retrievals. An issue worth

```
SELECT
     column->>'key' as key
FROM
    tablename
WHERE
    column->>'key' = 'value';
```

Fig. 1 PostgreSQL json query example

noting is that they found the query syntax for JSON queries "somewhat convoluted" (see Fig. 1 for an example of the syntax).

Amirian et al. [19] performed a benchmark of three different storage strategies for "geospatial big data" using Microsoft SQL Server 2012. Four geospatial datasets containing 100,000, 1 million, 10 million, and 100 million polygons, was stored using a relational, a spatial, and an XML-based layout. Performance of these strategies where evaluated based on single feature and range query retrieval, as well as a scalability test. In their setup the XML document (NoSQL document-store) layout provided the best performance and scalability, but the authors recommend a polyglot geospatial data persistence approach for geospatial big data handling.

Maia et al. [20] evaluated the performance of storing VGI in the document-based NoSQL data store MongoDB. Their system stored geographic locations as points in MongoDB using the GeoJSON format. An important takeaway from their work is the fact that document-based NoSQL databases provide greater flexibility when storing heterogenous data and does not require any previous knowledge of the data schema. Their study also compared the performance of the NoSQL setup with a relational setup using PostgreSQL. While their results are considered preliminary, they "favoured the use of NoSQL in the persistence layer of a VGIS, especially when dealing with large amounts of data". It should however be noted that the read-time benchmarks performed did not include any spatial filters.

Bartoszewski et al. [21] compared the spatial query performance of MongoDB and PostgreSQL/PostGIS. Using point and polygon data, they performed point-in-polygon-, radius-, and composite nearest neighbour and intersection queries. Their results show that MongoDB outperforms PostGIS in the point-in-point (3× faster) and compound (6× faster) queries. However, with increasing radii, PostGIS outperforms MongoDB by a factor of about 3× in the radius queries. The authors also note that NoSQL databases are lacking in terms of available geospatial operations compared to RDBMSes, but postulate that this will change in the future.

Santos et al. [22] evaluated relational (PostGIS), document-based (MongoDB), and graph-based (Neo4J) databases with a focus on the needs of mobile users that involve constant spatial data traffic. Their goal is to "highlight aspects in which different spatial DBMS architectures behave differently", rather than provide a benchmark. They defined four query sets, based on operations typically performed in mobile spatial applications: Nearby Points of Interest, Map View, Urban Routing, and Position Tracking. For each set they defined a set of database queries. Their results show that PostGIS in general provides the best performance, and "provides the most spatial

features". However, they note that MongoDB outperformed PostGIS in radius and k-NN queries. In addition, MongoDB is easy to scale horizontally.

Methodology

This section covers the implementation of the HOGS system and the experimental setup for the benchmarks performed on the system. First some common terminology is presented, then the architecture and implementation of the HOGS system is presented, before the experimental setup is described.

In this context we consider geospatial data to be described by the atomic unit of a *Feature*. A feature is a geographic *shape* (e.g. point, linestring, or polygon) as well as a list of accompanying key-value *attributes*. An example of a feature is a building footprint represented by a vector geometry describing a polygon, accompanied by attributes such as address, name of the owner, the year it was built, etc. A collection of features of the same type is a *Dataset* (or Feature collection). To continue the example, all building footprints in a city, municipality, or country makes up a specific building footprint feature collection. All features in a dataset shares the same attribute schema. Features belonging to a dataset are distributed as one or more files in one of several file formats and coordinate systems.

The HOGS system should be able to import multiple feature collections without any prior knowledge about the schema apart from what can be inferred from the data itself. The user supplies a list of files and what target dataset they belong to, as well as information about the database they are to be imported to. The Python programming language was chosen to implement the system, due to its multi-platform availability and the integration with the open source geospatial libraries GDAL/OGR and GEOS. The use of existing tools for common operations ensures a reduction of complexity and allows the system to support a wide range of geospatial file formats.[1]

Three overarching guidelines was followed when designing the system. First, the system should be *simple*. This is achieved by limiting the scope of the system, confining it to importing data. Second, the system should be *fast*. This is achieved by means of parallelization, exploiting the data structure to split the import into smaller tasks. Third, the system should offer *reproducibility*. This means that there should be no manual steps in the update procedure, so subsequent imports will behave the same way. This is ensured by the use of a configuration file.

Storage layouts

The two different *storage layouts* offered by HOGS determine how features and datasets are stored in the database. In the traditional *table-based* layout we create one database table per dataset. Each feature is a row in this table, with a column for each attribute, a geometry column, and a feature id column. While this approach could allow us to specify the geometry type as well, we opted for the generic *Geometry* data type, as some of our datasets contains mixed-type geometries. An example of the table-based layout is provided in Fig. 2.

[1] The list of supported vector formats in GDA/OGR at http://gdal.org/1.11/ogr/ogr_formats.html currently lists 78 formats.

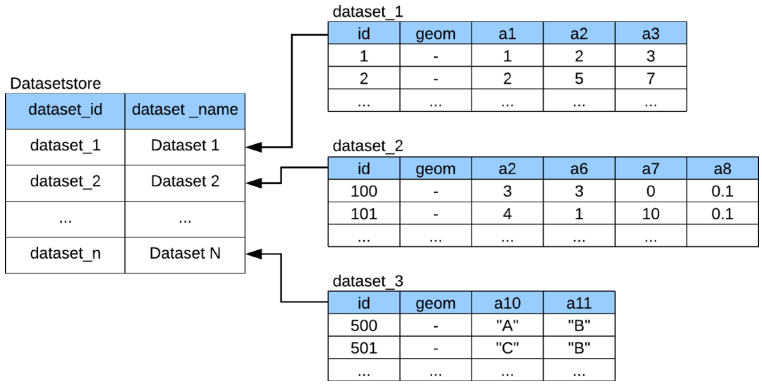

Fig. 2 Illustration of the table-based layout. Each dataset has its own table, with each attribute as a separate column. The table name is referenced in the metadata-table

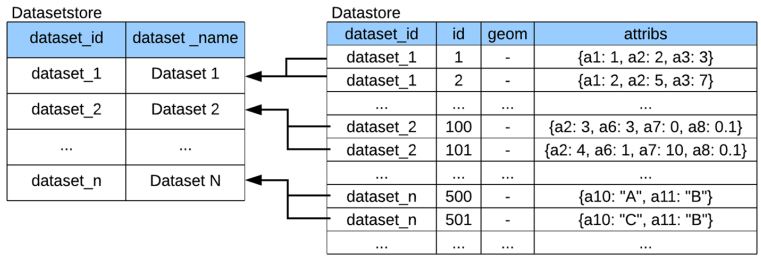

Fig. 3 Illustration of the jsonb layout. All datasets are stored in the common feature table, with their attributes in a jsonb-column. The *dataset_sid*-columns links features to a dataset in the metadata-table

In the NoSQL document-store inspired *jsonb* layout, we create a single database table that holds features for all datasets. Each row in this table is a feature, with the dataset id stored in a column. Geometry and feature id are also separate columns, similar to the table-based layout, as shown in Fig. 3. The main difference is that all the attributes are stored in a column of the jsonb type. The layout of the jsonb layout feature table is shown in Fig. 3. Another aspect of the jsonb layout is how it uses database views to emulate the table-based layout. For each dataset stored in the feature table a database view that expose the attributes as individual columns is created. This is done since most GIS tools are designed to work with the traditional table-based layout. By hiding the underlying structure from these tools, we ensure that they still work as expected.

HOGS support dataset versioning by using incremental version numbers with associated timestamps. When importing a dataset with an existing dataset id, this is considered a new version of the same dataset and the version number is increased. This means that an import can be run several times on the same database, but the storage layout of a previously initialized database cannot be changed.

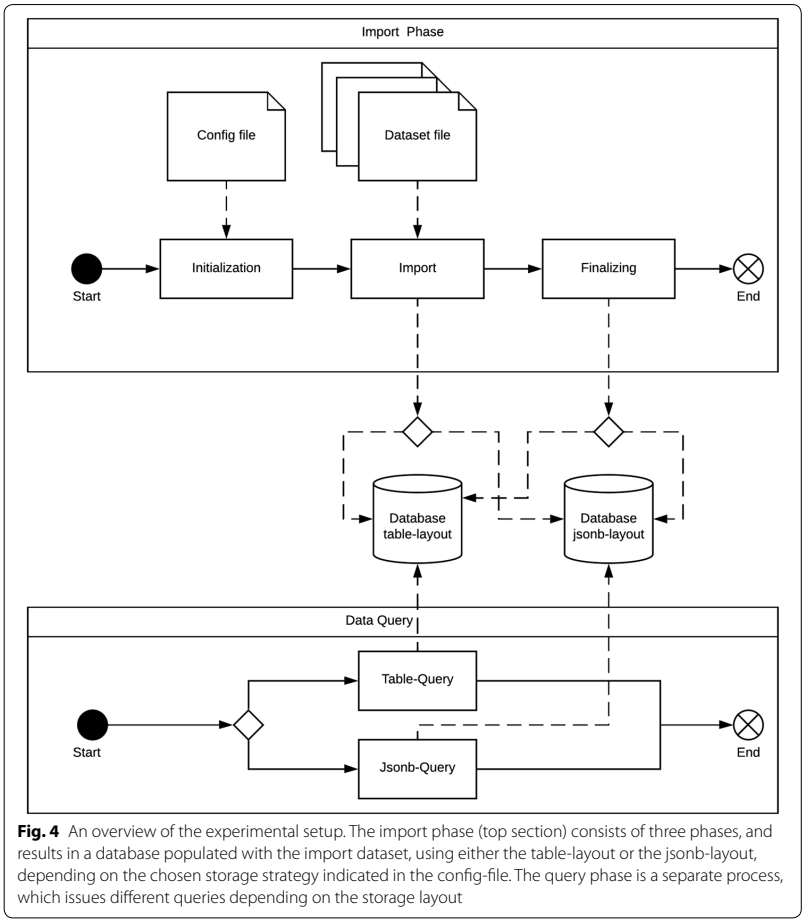

Fig. 4 An overview of the experimental setup. The import phase (top section) consists of three phases, and results in a database populated with the import dataset, using either the table-layout or the jsonb-layout, depending on the chosen storage strategy indicated in the config-file. The query phase is a separate process, which issues different queries depending on the storage layout

Import workflow

The actual import process consists of three phases: the initialization phase, the import phase, and the finalizing phase, as shown in the upper portion of Fig. 4. The content of these phases depends on the chosen storage layout and the previous state of the database.

In the *initialization phase* the configuration file is read and HOGS connects to the provided database. The first time HOGS connect to a database two metadata tables are created. These holds information about the stored datasets and determines the storage layout. If the jsonb layout is chosen, the aforementioned feature table is also created. The next initialization step is to parse the list of files associated with each dataset to be imported. The first file in each dataset is read using GDAL/OGR, to determine the attribute schema of the dataset. This information is stored in a metadata table. For the table-based layout the schema is used to create a temporary import table for each dataset. For the jsonb layout this information is used to create or update the database views.

In the *import phase* the files for each dataset is read and parsed, geometries are checked for errors and optionally transformed to geographic coordinates in the WGS84 datum (EPSG:4326), and then data is written to the database using a COPY statement. This was found to be the fastest operation when writing a large number of rows to the database. The geometry is copied using the EWKB format, the jsonb-attributes as a json encoded string, and the other columns are copied as native data types. Since the COPY statement bypasses table constraints on the database the geometries are validated using the GEOS library before they are written to the database.

Since each file in an import is independent of the other files, this phase can be executed in parallel. This can reduce the import time drastically and is an optimization worth implementing. HOGS achieve parallelization by creating a pool of import workers using the Python multiprocessing module. The size of this pool is set by the user and should correspond to the number of available CPU cores. In principle this pool could be distributed on individual machines as well, with one machine acting as a master node, coordinating the work.

When all files belonging to a dataset is imported, the dataset import proceeds to the *finalizing stage*. The contents of this stage depend on the chosen storage layout. For the jsonb layout this phase consists of creating, or updating, the aforementioned views and updating the metadata table to point to the correct version. For the table-based layout the finalizing stage creates an index on the geometry column, swaps the current version of the table with the temporary table, and stores the previous table with an identifier including its version number. When all datasets have finished the finalizing phase the import is completed.

Experimental setup

The HOGS system implements both a NoSQL storage approach and a traditional table-based storage layout. Therefore, we utilize HOGS in our laboratory setup to examine if there are any differences in import speed and query performance between the two layouts. We performed a quantitative analysis consisting of a series of imports and database queries. Using the same collection of datasets, we measured three features of each data storage layout: *import speed*, *query speed*, and *database size*.

All benchmarks where performed using an open geospatial dataset from the Norwegian Mapping authority known as *N50*. This is a 1:50,000 scale topological dataset of the Norwegian mainland, containing eight sub-datasets (feature collections), covering features such as area cover, transportation networks, place names, and height contours. Each of these sub-datasets have different attribute schemas and use different geometry types. The dataset is delivered in the Norwegian text-based geospatial file format SOSI, divided by dataset type and municipality. In total, the complete dataset contains 3415 files, totalling 7.9 GB on disk after extraction. This corresponds to approximately 15 million features, more specifically 2 million point features, 10 million linestring features, and 3 million polygon features. An overview of the N50-dataset is provided in Fig. 5.

The experimental setup consisted of a standard enterprise hardware setup, equipped with an Intel Core i7-4710MQ Processor, 32 GB RAM, and a 300 GB HDD, running Windows 10. PostgreSQL 9.6.3 with PostGIS 2.3 was installed using a Docker-image. The installation used the default configuration and was wiped between each run. HOGS

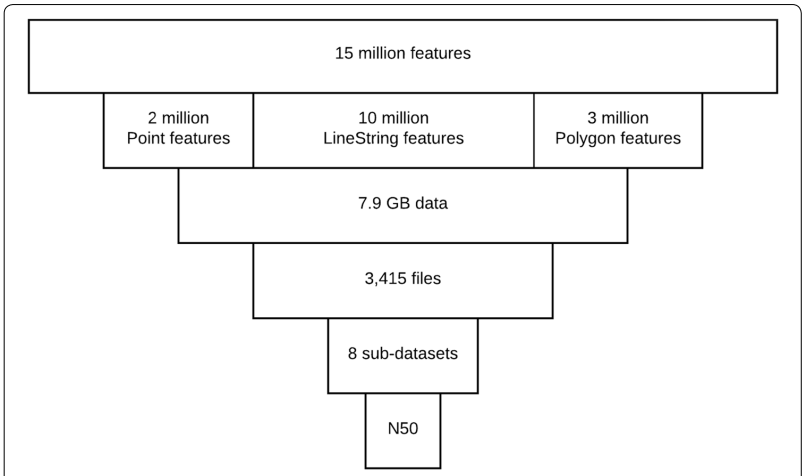

Fig. 5 Overview of the N50 dataset

Table 1 Benchmark results for the two examined storage layouts

	Import		Query—intersect		Query—intersect/attribute		Disk size
	Speed (m)	SD	Speed (s)	SD	Speed (s)	SD	(GB)
Table-based	*79*	3.57	*19*	0.54	*99*	3.24	*12.29*
Jsonb	179	4.50	25	1.00	162	3.30	17.50
Difference	100		6		63		5.21

The better results for each metric are emphasized

itself was run using Python 2.7 on the Windows Subsystem for Linux. This means that the complete experiment was run on a single machine, with no network speed and latency to consider.

Results

Timing from the experiments are affected by several factors. We have chosen to focus on the relative difference between the two storage layouts, not the elapsed time on its own. The results of the actual benchmarks are summarized in Table 1 and presented in detail in the following sections.

Import benchmark

Import speed is calculated as the time it takes from HOGS is provided with a configuration file containing a list of datasets and associated files until the data in these files are available in the provided database in the specified layout. This is the upper portion of Fig. 4. In our case this means the time it takes to read the 3415 SOSI files from disk and store their contents in the database.

```
def get_intersects(self, table_name, geom):
    with self.conn.cursor() as cur:
        query = sql.SQL('''
            SELECT * FROM {}
            WHERE ST_Intersects(geom, ST_GeomFromWKB(%s, 4326))
        ''').format(
            sql.SQL(table_name)
        )

        cur.execute(query, (psycopg2.Binary(geom),))
        res = []
        for record in cur:
            res.append(record)
        return res
```

Fig. 6 Intersect query used for benchmarking

We relied on the built-in logging capabilities of HOGS, noting the time an import started and finished. The import was run five times for each storage layout, with HOGS presented with a new database instance on each run.

Our benchmarks show that the average import speed for the table-layout is 1 h and 19 min, while the jsonb layout on average took 3 h. The results indicate that the table-based layout is 56% faster than the jsonb layout with regard to the import phase.

Query benchmark

Database query optimization is complex and it is impossible to provide a benchmark that covers all usage patterns of a generalized geospatial data storage such as the one described here. However, we chose to base our query benchmarks on the usage pattern of a known system and use data gathered from the logs of this system.

The system in question performs a series of intersection queries using a query polygon against a series of datasets in order to find areas of interest. From the query logs of this system we extracted 840 query geometries. These polygons cover areas in the range 1–100 m^2 on the Norwegian mainland and are distributed according to the needs of the users of the system. The query benchmark is depicted in the lower portion of Fig. 4, and is independent of the design of the import phase, as it only relates to the resulting database contents and layout.

Two queries were designed to be run against each of the eight datasets in the n50 dataset. One plain intersection query using the PostGIS *ST_Intersect* and one query consisting of an intersection as well as an attribute query (see Figs. 6 and 7). For the attribute query we chose the attribute "objekttypenavn", which is present for all our datasets, and for each dataset we used all the distinct values of this attribute. This means that both queries were executed about 7000 times.

Each of these series of queries were run five times for each database layout, and the total query time for each layout was averaged. For the plain intersection queries the average time was 19 s for the table-based layout, and 25 s for the jsonb layout. This

```
def get_intersects_with_objtype(self, table_name, geom, objtype):
    with self.conn.cursor() as cur:
        query = sql.SQL('''
            SELECT * FROM {}
            WHERE ST_Intersects(geom, ST_GeomFromWKB(%s, 4326))
            AND objekttypenavn = %s
        ''').format(
            sql.SQL(table_name)
        )

        cur.execute(query, (psycopg2.Binary(geom), objtype, ))
        res = []
        for record in cur:
            res.append(record)
        return res
```

Fig. 7 Intersect and attribute query used for benchmarking

means that the plain intersection queries are 25% faster using the table-based layout. For the intersection queries with an additional attribute query the average query time in the table-based layout was 99 s, while the jsonb-layout took 162 s on average. This means that intersection queries with attribute queries are 39% faster using the table-based layout.

Database size

This benchmark measures the actual size on disk used to store the datasets using the two different storage layouts. The size of the databases was measured using the PostgreSQL system table *pg_database*, and the operator *pg_database_size*. These numbers show that the table-layout database uses 12.29 GB on disk, while the jsonb-layout use 17.5 GB. This means that the table-based layout takes up 30% less space than the jsonb-layout.

Discussion

The performed benchmarks show that the table-based layout performs better than the NoSQL-inspired jsonb-layout on all metrics. Insertion speed is the metric with the largest difference. Here, the table-based layout is able to insert the test-data more than twice as fast as the jsonb-layout. These findings contradicts similar studies found in literature [18, 23], which report that NoSQL document stores or data types outperform relational layouts.

However, many factors influence benchmark results, and while the setups in the related studies are similar there are several differences in design that may explain the difference in results. We suspect that the most important factor in our setup is table size. Since our two layouts are both implemented in PostgreSQL/PostGIS, and both layouts use the PostGIS geometry types, the main difference between them is the way attributes are stored, and how many tables are used. This difference holds the explanation to why the table-based layout performs better.

Both layouts use a spatial index for the geometry column. In the case of insertion, the way this is handled differs. For the table-based layout the index is created after data is inserted into the table. In the jsonb-layout this is not an option, as we are inserting data into a common table. This means that for the jsonb-layout, the spatial index has to be created in the initialization phase and updated in-place during the import, which is more time-consuming than creating an index after all data is added. In the case of data queries, the main difference is table size, with the common table in the jsonb-layout being larger. While this table is indexed on dataset id, it is still faster to directly query a table with just the relevant features than to select these using an index.

None of the examined related studies used data that could be logically segmented into sub-datasets, and thus the table sizes would have been similar in both cases. This may explain why our findings differ. However, many geospatial datasets can be segmented into separate datasets by partitioning on what types of features are being mapped. If this is the case, our results show that a table-based layout is favourable. A counterpoint is that the "one table per dataset" approach can be combined with the jsonb-layout as well. While this is technically true, a key feature of NoSQL data stores is that there is no need to logically separate data in tables. In order to keep with this philosophy, we chose to implement one common table for the jsonb-layout.

Another important aspect of a database used for managing open geospatial data is usability. Navarro-Carrión et al. [18] noted that the query syntax used for the PostgreSQL JSON data type is "somewhat convoluted", an assessment we find to hold true (see Fig. 1 for an example). In addition, we found that widely used desktop GIS packages such as QGIS are unable to read attributes stored as jsonb with the same ease as they read traditional tables. This was mitigated by creating database views that maps the jsonb-syntax to a traditional relational table-layout, with one dataset per table and one attribute per column.

We used the HOGS system to perform benchmarks on the Norwegian n50-dataset, delivered as files in the SOSI format. This does not imply that the system is limited to one file format. Due to the use of the GDAL/OGR library, a plethora of geospatial vector formats (78 at the time of writing) can be imported using HOGS. For example, we have successfully imported data downloaded from OpenStreetMap using HOGS.

Conclusions and further work

We have found that, for homogenous collections of spatial datasets, a traditional one-table per-dataset layout outperforms a NoSQL document-store combined-table layout. The traditional layout performs better on both insertion and query speed, and it uses less storage space. We expected that the NoSQL approach would enable an easier insertion routine, but with the HOGS system leveraging GDAL/OGR we found that the overhead of creating individual tables for each dataset can be automated and introduces no extra complexity.

We also found that while a single table containing a heterogenous mix of features from different datasets intuitively sounds easier to work with, this kind of layout is not compatible with a range of off-the-shelf WMS-servers and desktop GIS packages. In practice this means that a NoSQL layout must emulate a traditional table-based layout using views in order to work with such applications.

These findings differ from the other studies examined. While one explanation for this discrepancy may be the fact that we used data that could be segmented into sub-datasets, this shows that further examination is required. A more thorough benchmark-setup, including a larger pool of datasets is a natural next step. Leveraging other sources of open geospatial data, such as OpenStreetMap, and European INSPIRE-data, would enable us to verify our results on a wide selection of geometry types and attribute schemas. Another possible route is to enable a cloud-based lab-setup, where automation in used to create, run, and tear down the database and import environments between each test-run. This would enable us to exclude all possible side-effects of running the benchmarks on a single hardware setup, which would also allow for adjustment of additional parameters, such as processor speeds and available memory.

In terms of further work, a third storage layout worth examining is the so-called Data Lake [24]. In this concept, the data is stored "as is" in raw format, and only processed when needed [25]. This allows for easy storage of vast amounts of data, but we envision this would present its own performance issues related to queries, where both geometries and attributes will have to be parsed and transformed. However, we find this concept interesting, and would like to investigate how it can be applied to geospatial data.

In conclusion, the results presented in this paper indicate that the NoSQL layout is slower, both in terms of import and query speed, when considering heterogenous geospatial data. In addition, the NoSQL layout does not offer any additional simplification of the import process. Based on these conclusions, we cannot recommend the use of the jsonb-datatype in PostgreSQL for storing geospatial data that can be segmented into homogenous datasets. This statement holds as long as the storage-space requirements does not exceed the capabilities of a single database instance. This in turn means that relatively large amounts of open geospatial data can be efficiently stored and queried using traditional RDBMS technologies. This approach is beneficial, as it enables the use of existing software integrations and does not require a weakening of the ACID-principle.

Abbreviations

ACID: Atomicity, Consistency, Isolation, and Durability; BASE: Basically Available, Soft state, Eventual consistency; EWKB: Extended Well-Known Binary; HOGS: Heterogeneous Open Geodata Storage; IoT: Internet of Things; JSON: JavaScript Object Notation; NoSQL: "Not Only SQL"; RDBMS: Relational Database Management System; SOSI: Samordnet Opplegg for Stedfestet Informasjon—Systematic Organization of Spatial Information; VGI: Volunteered Geographic Information; XM: Extensible Markup Language.

Acknowledgements

The author would like to thank Dr. Alexander Salveson Nossum for providing valuable feedback on the contents of the paper, and Dr. Birgitte Hjelmeland McDonagh for valuable help with grammar, style, and proofreading.

Authors' contributions

AFS is the sole author of this manuscript, as well as the developer of the HOGS system presented herein. The author read and approved the final manuscript.

Funding

This work was supported by Norkart AS and The Research Council of Norway [Grant Number: 261304l].

Availability of data and materials

The datasets used for then benchmarking in this manuscript are available from the corresponding author on reasonable request.

Competing interests

The author declares no competing interests.

Received: 25 June 2019 Accepted: 4 November 2019
Published online: 18 November 2019

References

1. Chen M, Mao S, Liu Y. Big data: a survey. Mobile networks and applications. US: Springer; 2014. p. 171–209.
2. Laney D. 3D data management: controlling data volume, velocity and variety. META Group Research Note 6; 2001.
3. Lee J-G, Kang M. Geospatial Big Data: challenges and opportunities. Big Data Res. 2015;2:74–81. https://doi.org/10.1016/j.bdr.2015.01.003.
4. Li S, Dragicevic S, Castro FA, et al. Geospatial big data handling theory and methods: a review and research challenges. ISPRS J Photogramm Remote Sens. 2016;115:119–33.
5. Chandra DG. BASE analysis of NoSQL database. Fut Gen Comput Syst. 2015;52:13–21. https://doi.org/10.1016/j.future.2015.05.003.
6. Stonebraker M, Hellerstein J. What goes around comes around. Readings in database systems. 2005;4:1724–35.
7. Güting RH. An introduction to spatial database systems. VLDB J. 1994;3:357–99. https://doi.org/10.1007/BF01231602.
8. OGC. OpenGIS® implementation standard for geographic information—simple feature access—part 1: common architecture, vol. 93. Wayland: Open Geospatial Consortium Inc; 2010.
9. Cattell R. Scalable SQL and NoSQL data stores. ACM SIGMOD Rec. 2011;39:12. https://doi.org/10.1145/1978915.1978919.
10. Leavitt N. Will NoSQL databases live up to their promise? Computer. 2010;43:12–4. https://doi.org/10.1109/MC.2010.58.
11. Ameya N, Anil P, Dikshay P. Type of NOSQL databases and its comparison with relational databases. Int J Appl Inf Syst. 2013;5:16–9.
12. Chasseur C, Li Y, Patel JM. Enabling JSON document stores in relational systems. In: WebDB; 2013. p. 14–15.
13. Sveen AF. The open geospatial data ecosystem. Kart og plan. 2017;77:108–20.
14. MongoDB. MongoDB Manual; 2018. https://docs.mongodb.com; https://docs.mongodb.com/manual/reference/geojson/index.html. Accessed 11 Apr 2018.
15. Del Alba L. Faster Operations with the JSONB Data Type in PostgreSQL; 2017. https://www.compose.com/articles/faster-operations-with-the-jsonb-data-type-in-postgresql/. Accessed 11 Apr 2018.
16. Petković D. JSON integration in relational database systems. Int J Comput Appl. 2017;168:14–9. https://doi.org/10.5120/ijca2017914389.
17. Linster M. Postgres outperforms MongoDB and ushers in new developer reality. In: The EDB Blog; 2014. https://www.enterprisedb.com/node/3441. Accessed 10 Apr 2018.
18. Navarro-Carrión JT, Zaragozí B, Ramón-Morte A, Valcárcel-Sanz N. Should eu land use and land cover data be managed with a Nosql document store? Int J Des Nat Ecodyn. 2016;11:438–46. https://doi.org/10.2495/DNE-V11-N3-438-446.
19. Amirian P, Basiri A, Winstanley A. Evaluation of data management systems for geospatial big data. Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics). Cham: Springer; 2014. p. 678–90.
20. Maia DCM, Camargos BDC, Holanda M. Performance analysis on voluntary geographic information systems with document-based NoSQL database. Stud Comput Intell. 2018;718:181–97. https://doi.org/10.1007/978-3-319-58965-7_13.
21. Bartoszewski D, Piorkowski A, Lupa M. The comparison of processing efficiency of spatial data for PostGIS and MongoDB databases. In: Kozielski S, Mrozek D, Kasprowski P, et al., editors. Beyond databases, architectures and structures. Paving the road to smart data processing and analysis. New York: Springer International Publishing; 2019. p. 291–302.
22. Santos PO, Moro MM, Davis CA. Comparative performance evaluation of relational and NoSQL databases for spatial and mobile applications. In: Chen Q, Hameurlain A, Toumani F, editors. Database and expert systems applications. New York: Springer International Publishing; 2015. p. 186–200.
23. Amirian P, Basiri A, Winstanley A. Efficient online sharing of geospatial big data using NoSQL XML databases. In: 2013 fourth international conference on computing for geospatial research and application. New York: IEEE; 2013. p. 152–152.
24. Dixon J. Pentaho, Hadoop, and Data Lakes. In: James Dixon's Blog; 2010. https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/. Accessed 9 Sept 2019.
25. Miloslavskaya N, Tolstoy A. Big Data, Fast Data and Data Lake Concepts. Procedia Comput Sci. 2016;88:300–5. https://doi.org/10.1016/j.procs.2016.07.439.

Publisher's Note

Paper IV

GeomDiff — an algorithm for differential geospatial vector data comparison

Sveen *Open Geospatial Data, Software and Standards* (2020) 5:3
https://doi.org/10.1186/s40965-020-00076-4

Open Geospatial Data, Software and Standards

SOFTWARE Open Access

GeomDiff — an algorithm for differential geospatial vector data comparison

Atle Frenvik Sveen

Abstract

Diffs, a concept known from source code version control systems such as git, is interesting for geospatial, event-based workflows. We investigate how the native mathematical structure of vector geometries can be utilized in order to create a diffing algorithm tailored to geospatial vector data. Diffing algorithms are a well-researched area which dates to the 1970ies; however, we find that geospatial diffing operations tends to be carried out using generic algorithms combined with a pre- and post-processing step. We created GeomDiff, an algorithm and storage format tailored to geospatial vector data. The creation time, apply/undo time, and patch size of GeomDiff was compared to three other generic algorithms by running an online experiment using 2.5 million real-world geometry pairs from OpenStreetMap. We found that the GeomDiff algorithm performs better than or on-par with the alternatives on point-geometries, and complex geometries with a small (< 500) vertex count. We argue that there are both computation time and storage space improvements to be gained by using a tailored diffing algorithm for geospatial vector data. These promising first results encourages further refinement of the algorithm in order to handle complex geometries efficiently as well.

Keywords: Geospatial data management, Diffing, Event based workflows

Introduction

In computer science, a diff[1] is a set of machine-executable instructions to transform version *n* of source code or documentation into version *n + 1* [1]. The concept of diffs is an essential component of source code version control systems such as *git* [2], one of the fundamental building blocks of modern software engineering. Other application areas also take advantage of the diff concept, enabling real-time collaborative editing tools such as Google Docs [3]. The concept of diffs is also important when working with event-based workflows, where messages describing changes are a core component [4]. A recent, geospatial, application of the concept is "Sno" [5] which uses git to apply version control to geospatial data.

The methods used for creating a diff and the format it is stored in will affect both *creation* time, *storage requirements*, and *apply* and *undo* time. These metrics affect the overall performance and requirements of a diff-based workflow. Using a diffing algorithm and diff storage format tailored to geospatial vector data has the possibility to provide an efficient, performant, and reliable event-based workflow for geospatial data management.

Diffing algorithms specifically tailored to geospatial data are rare in the literature. Thus, we implemented "GeomDiff", an algorithm for geospatial diffing. GeomDiff takes advantage of the native mathematical properties of geospatial vector data in order to improve the performance of geospatial event-based workflows.

We review existing literature on diffing algorithms and formats in general and provide an overview of existing solutions for versioning of geospatial vector data using diffs. The concepts used to implement the GeomDiff algorithm is then explained and presented.

To evaluate the proposed algorithm, we perform a large-scale experiment using real-life data in a controlled

[1]Also known as an *edit script*, a *changeset*, a *patch*, or a *delta*

Correspondence: atle.f.sveen@ntnu.no
Norwegian University of Science and Technology, Trondheim, Norway

and replicable cloud-based environment. The GeomDiff algorithm is compared to three other approaches to diffing of geospatial vector data in order to investigate how it performs on creation time, apply/undo time, and storage requirements.

Implementation

Motivation

Creating a diff is the process of finding changes between two versions of an object and describing them. Change detection is in itself a topic that is widely studied with regards to Remote sensing and image processing [6], as well as computer science [7]. The term "diff" itself was introduced by Hunt & MacIlroy [8], which described a program which "[…] reports differences between two files, expressed as a minimal list of line changes to bring either file into agreement with the other". The GNU diff program is based on the work carried out by Miller & Myers [9], and Myers [10], who found that the dual problems of finding a longest common sub-sequence of A and B and finding a shortest edit script for transforming A into B are equivalent to finding a shortest/longest path in an edit graph.

The diff program and its iterations are focused on comparing text files and are therefore well suited for tracking revisions to text and computer source code. Variations of this approach serves as a building block of version control systems [2]. Other researchers have focused on creating systems for detecting changes in hierarchically structured information, such as data stored in a database [11] and binary data [12]. Dedicated diff tools and formats have also been created for formats such as JSON [13] and XML [14].

We did not find examples of specialised diffing algorithms for geospatial data in the literature, but some approaches from the industry was identified and are presented in the following. The GeoDiff library [15] aims to simplify vector data management by "keep[ing] track of changes, calculate the differences, merge and consolidate the differences". However, the library seems to focus on changes at a dataset level. A related idea is to apply version control concepts to geospatial vector data. This has been attempted several times by various actors. GeoGit, later renamed GeoGig, was released in 2014 and "allows for decentralized management of versioned geospatial data" [16]. However, an inspection of the source code does not indicate that the project employs diffs but stores each separate change as a new geometry. A more recent approach to version control of geospatial data is Sno, which is built on top of git [5]. This means that this system uses a text-based differ at its core, but presumably with some modifications.

Principles of GeomDiff

Existing diffing algorithms for textual data, binary data, or format-specific algorithms can be applied to geospatial vector data using pre- and post-processing steps. However, geospatial geometries are natively mathematically defined as vectors in N-dimensions. By converting them to text or some other format, we lose the ability to utilize mathematical relations to describe changes in the geometries. This is the main idea behind GeomDiff; to take advantage of the opportunities presented by the mathematical nature of vector geometries in order to create a more efficient algorithm.

Table 1 presents a selection of example geometries in their original and modified state, along with an example of a change script. For point geometries we record the operation (create, delete, or modify) as well as the change to the coordinate expressed as a delta. In order to support reverting a change script, the current value before deletion is recorded in a delete operation.

While a point consists of a single coordinate pair, other geometry types are more complex. These consists of one or more, possibly nested, ordered lists of coordinate pairs. Linestrings are described by a single, ordered list of coordinate pairs, where each coordinate pair describes a vertex. The linestring can be both created and deleted in a similar fashion to a point, but a modification is more complex. Changes to each vertex can be described using the edit script outlined for coordinates, but we need to keep track of the indices of the changed vertices as well, as illustrated in Table 1 (id = 4).

A polygon is even more complex, as it may contain hulls. Thus, a polygon consists of n ordered lists of coordinates, and each of these can be added, deleted, or modified. In addition, each of the vertices in each list can be added, deleted, or modified. However, just as the coordinate edit script is used to represent each change to a linestring, a linestring change script can describe the change to each ring of a polygon (Table 1, id = 5). Using this hierarchical pattern, multi-geometries are handled by adding another layer; multi-point change scripts are lists of point change scripts, multi-linestrings and multi-polygons extends linestring and polygon change scripts in the same manner.

Geospatial data is in many cases represented as a collection of features. A feature is a combination of a geometry and a set of textual or numeric attributes or properties. In order to create a feature patch, the attributes must be handled as well. While this is an important aspect of a geospatial data versioning workflow, the GeomDiff algorithm is not designed to work on features. However, feature attributes are

Table 1 Edits to geometries and their associated change scripts. Ids 1–3 are modification, creation, and deletion of a point, id = 4 is modification of a linestring by inserting, modifying, and deleting vertices. Id =5 is modification of a polygon by modifying one vertex in the shell and deleting the hull. Geometries are described using the WKT format

Id	Geometry Type	Original	Modified	Change Script
1	Point	(10.53 60.10)	(10.52 60.10)	Modify, (-0.01 0)
2	Point	Null	(10.53 60.10)	Create, (10.53 60.10)
3	Point	(10.53 60.10)	Null	Delete, (10.53 60.10)
4	LineString	(1 1, 2 2, 3 3, 4 4)	(0 0, 1 1, 2.5 2.5, 3 3)	{0: Insert, (0 0), 1: Modify, (0.5 0.5) 3: Delete, (4 4)}
5	Polygon	((0 0, 10 0, 5 10, 0 0), (1 1, 2 2, 2 1, 1 1))	((0 0, 10 0, 6 10, 0 0))	{0: Modify, {2: (1, 0)}, 1: Delete, (1 1, 2 2, 2 1, 1 1)}

usually simple key-value pairs that can be represented using formats such as JSON or XML. As previously discussed, specialized diffing algorithms for these formats exists and can be used. An example of this approach is implemented in the attached file FeatureDiffer.cs.

GeomDiff implementation

The main principles described in the previous section are implemented using a generic *Diff* class, as outlined in Listing 1. Here, the value is a generic property, which means that we can represent change scripts for all geometry types using this class. In the case of a point, the *PointDiff* inherits from *Diff* using the *CoordinateDelta* as *TComponent.* Describing changes to a linestring geometry is more complex, as we need to keep track of the vertex indices. This is where the Index and Operation properties on the Diff class are used, as the *LineStringDiff* class uses a *List* < *PointDiff* > as *TComonent*. The same pattern is repeated for the other geometry types.

While the presented class hierarchy represents changes, it does not describe how these changes are detected. In the case of point geometries, the difference is expressed by the change in each coordinate, which is a straightforward mathematical computation. For linestrings and other

```
public abstract class Diff<TComponent>
{
       public int Index {get; set;}
       public TComponent Value {get; set;}
       public Operation Operation {get; set;}

}


public class CoordinateDelta
{
    public double? X { get; set; }
    public double? Y { get; set; }
    public double? Z { get; set; }
}

public class PointDiff: Diff<CoordinateDelta> {}


public class LineStringDiff: Diff<List<PointDiff>> {}
```

Listing 1: C# Class hierarchy used to describe diffs relating to diferent geometry types. The abstract, generic Diff class serves as a basis, and the CoordinateDelta class is the simplest example of a diffed component. A PointDiff is implemented by providing the Diff class the CoordinateDelta. A LineStringDiff is more complex, as it consists of a list of PointDiffs.

```
List<TComponent> PatchList(List<TComponent> components, List<IDiff<TComponent>> diffs)
{
    var patched = new List<TComponent>();

    var numElements = Math.Max(components.Count - 1, diffs.Max(v => v.Index));

    for (var index = 0; index <= numElements; index++)
    {
        //insert all inserts at this index
        var inserts = Util.GetDiffs(index, Operation.Insert, diffs);
        patched.AddRange(inserts.Select(insert => insert.Apply(null)));

        //skip the component if it is marked for deletion
        var delete = Util.GetDiff(index, Operation.Delete, diffs);
        if (delete != null) continue;

        //check if the original list still has elements
        var element = Util.GetAt(index, components);
        if (element == null) continue;

        //find and apply any modifications
        var modify = Util.GetDiff(index, Operation.Modify, diffs);
        patched.Add(modify != null ? modify.Apply(element) : element);
    }

    return patched;
}
```

Listing 2: Simplified version of the PatchList method used for applying a patch, implemented in C#.

sequences of coordinates, we employ a generalized version of the Myers diff algorithm [10], where the input is two lists of components and a function to compare them. By utilizing the recursive strategy presented above, this approach works both for comparing individual vertices in a linestring and for comparing linear rings in a polygon. Thus, diff creation can be implemented by combining this approach with a method for compacting diffs by merging consequent inserts and deletes into modify operations.

Applying a diff to a geometry follows the same recursive pattern. A simple mathematical operation can patch a single coordinate. A list of components (coordinates, linear rings, or geometries) is patched using the *PatchList* method, reproduced in Listing 2. Undo operations use the same method, but a pre-processing step reversing the diff is applied first.

Another important aspect is a storage format for the created diffs. Serializing and deserializing the generated C# objects is an easy solution, but this introduces an unnecessary coupling to a specific implementation. In addition, this is not an efficient approach in terms of storage requirements.

Thus, we created a binary format for storing diffs, inspired by the WKB (Well Known Binary [17]) format. The format consists of a header, describing the geometry type and dimensions, and the actual diff elements. Writer and reader implementations to convert to and from C# objects are created as part of the implementation.

Benchmark design and implementation

The performance of the GeomDiff algorithm was examined by performing a large number of diff *creation*, *apply*, and *undo* operations on data from OpenStreetMap. In addition, the same operations were performed on the same data, using three other diffing implementations. This ensures that we can perform a statistical analysis to test our hypotheses.

The benchmark was performed in a Microsoft Azure cloud environment. This allows for easy scale-out in order to handle high workloads in parallel. Furthermore, it ensures consistent hardware performance at a reasonable price. The setup consists of a virtual machine running a PostgreSQL/PostGIS database, an Azure Function App, and an Azure storage account for message passing and temporary storage. In addition, NodeJS and Python scripts were developed to import test-data, orchestrate the benchmark, and analyse the results.

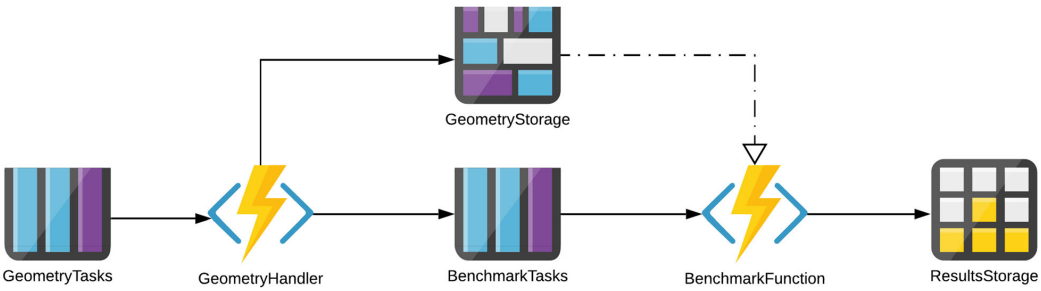

Fig. 1 The Azure pipeline for running the benchmark. The GeometryTasks queue is loaded with one message per geometry pair. The GeometryHandler processes each message, by fetching the two geometries in the pair and storing them in a blob store. It then puts four messages on the BenchmarkTasks queue, one for each algorithm under test. This message has a reference to the blob id of the geometry pair. The benchmark function then loads the geometry pair and executes the three operations (create, apply and, update) using the specified algorithm, while logging time usage. The results are then stored in a table storage for later retrieval and analysis

An extract from the open licensed OpenStreetMap (OSM) dataset [18] was used as test data. A full history extract for Norway was manually downloaded from geofabrik.de and imported to a PostGIS database, using a custom-made script [19]. To prepare the dataset for the benchmark, a series of queries was run to identify geometry pairs. A geometry pair is defined as two versions of the same geometry. In cases where the OSM dataset had more than two versions of a geometry, we chose the minimum and maximum version to represent the two versions. Since the OSM data model stores all linestring and polygon vertices as points, and referencing them from a "ways" table, we selected all points with at least one tag to represent points. Linestrings and polygons was created from the ways table, based on whether they were closed or not. This process created a benchmark dataset consisting of 1,335,489 point-, 813,503 linestring-, and 433,776 polygon-pairs. For each linestring- and polygon pair the *NumVertices* variable (Eq. 1) was computed and stored.

$$NumVertices = avg(numVertices(geometry_{v1}), numVertices(geometry_{v2})) \quad (1)$$

The test pipeline consists of an Azure storage account and a Function App, as depicted in Fig. 1. The data flow in this setup is highly parallelisable and can be scaled to handle increased workloads with minimal effort. The first stage of the pipeline is a storage queue (*GeometryTasks*), populated by a NodeJS script. This queue contains one message per geometry pair in the benchmark dataset, consisting of an id and two version numbers, as well as the geometry type. Attached to this queue is a function app (*GeometryHandler*) which fetches the corresponding geometries from the PostGIS database and stores them in a blob store (*GeometryStorage*). In addition, this message puts four messages on a second queue (*BenchmarkTasks*), with a blob id and a differ name. The second function app (*BenchmarkFunction*) is triggered by the BenchmarkTasks queue and fetches the geometry pair from GeometryStorage and runs the benchmark using the indicated algorithm. On completion the results are saved to a table storage (*ResultsStorage*).

The actual benchmark of the algorithms under test are performed in the *BenchmarkFunction* function app. This is a serverless instance running .NET core code [20]. The actual algorithm implementations are written to conform to a common interface, as depicted in Listing 3, and made available as a NuGet package, which is imported by the *BenchmarkFunction* app.

```
public interface IGeometryDiff
{
    byte[] CreatePatch(IGeometry oldGeom, IGeometry newGeom);
    IGeometry ApplyPatch(IGeometry oldGeom, byte[] patch);
    IGeometry UndoPatch(IGeometry newGeom, byte[] patch);
}
```

Listing 3: IGeometryDiff interface used to implement the diff algorithms

Table 2 Diffing algorithms used in the benchmark

Algorithm	Format	Library	Link
TextDiff	Text	Diff Match Patch	https://github.com/google/diff-match-patch
JsonDiff	JSON	jsondiffpatch.net	https://github.com/wbish/jsondiffpatch.net
BinaryDiff	Binary	Deltaq	https://github.com/jzebedee/deltaq
GeomDiff	Vector Geometry	GeomDiff	https://github.com/atlefren/GeomDiff

The algorithm implementations under test are listed in Table 2. The three other algorithms are based on open source implementations of three different diffing formats; textual data, binary data, and JSON-data. All these algorithms require a pre- and a post-processing step, where the geometry is converted to the appropriate format and back. These algorithms where chosen as they represent existing approaches to handling diffing of geospatial vector data. The Bsdiff algorithm [12], used in the BinaryDiff implementation does not support the undo operation, but we still chose to include it in the benchmark, as it represents another approach to diffing. The pre- and post-processing steps use the open source NetTopologySuite [21] library to convert the geometries into appropriate formats. Well Known Text (WKB) [17], Well Known Binary (WKB) [17], and GeoJSON [22] was chosen as formats to convert to text, binary, and json, respectively.

Results

For each of the three geometry types in the test dataset we performed hypothesis testing on four metrics:

- Creation time: The time it takes to create a diff given two versions of a geometry.
- Apply time: The time it takes to create version n + 1 of a geometry, given version n and a diff.
- Undo time: The time it takes to roll back to version n of a geometry, given version n + 1 and a diff.
- Patch size: The physical size of the created diff.

We expect the GeomDiff algorithm to exhibit faster creation-, apply- and, undo-time for point, linestring and polygon geometries compared to the other algorithms. In addition, we expect the GeomDiff algorithm to produce smaller patches.

The statistical testing was performed using the following procedure, implemented as a Python script [23]. All statistical tests were performed using a significance level of 0.05. For each metric of each geometry type the recorded data for each of the four algorithms was loaded. First, all errors were counted, recorded (see Table 6), and then removed before further analysis. An error is either an exception thrown by the code, or an instance where the patch did not create the expected result.

Second, a D'Agostino and Pearson's test [24] was applied to check each group for normal distribution. Since none of the groups were normally distributed ($p < 0.05$), a Kruskal-Wallis H-test [25] was then applied to test H_0, that the samples from all algorithms came from the same distribution. Since H_0 was rejected in all cases ($p < 0.05$), we continued with a post hoc test to perform pairwise comparisons between the four algorithms. Using Conover's test [26], we found that none of the pairs where statistically similar ($p < 0.05$). This means that all differences between the mean values for each algorithm are significant.

Point geometries

For point geometries (Table 3), a total of 1,335,489 geometry pairs were checked for each algorithm. Overall, the BinaryDiff algorithm is slower than the fastest algorithm by a factor of 1000 on create and apply. The TextDiff and JsonDiff algorithms show comparable results, apart from patch size. The GeomDiff algorithm produces

Table 3 Benchmark results for point geometries. Best results in each case in bold. The standard deviation of patch size for points using the GeomDiff algorithm is 0, as a point change is described using two doubles. This means that the size of a point patch will always be the size of two doubles and metadata of a fixed size

Algorithm	Create Time (ms)		Apply Time (ms)		Undo Time (ms)		Patch Size (b)	
	Mean	St.dev	Mean	St.dev	Mean	St.dev	Mean	St.dev
TextDiff	0.22	10.92	0.47	15.64	0.32	2.32	54.0	30.0
JsonDiff	0.38	7.21	0.21	2.51	0.16	1.62	184.0	94.0
BinaryDiff	190.88	272.07	67.39	131.74	–	–	168.0	20.0
GeomDiff	**0.03**	1.80	**0.02**	0.58	**0.01**	0.40	**25.0**	0.0

Table 4 Benchmark results for linestring geometries. Best results in each case in bold

Algorithm	Create Time (ms)		Apply Time (ms)		Undo Time (ms)		Patch Size (b)	
	Mean	St.dev	Mean	St.dev	Mean	St.dev	Mean	St.dev
TextDiff	9.01	58.56	1.00	10.98	1.04	4.61	623.44	1733.22
JsonDiff	**2.27**	35.96	1.12	10.08	1.06	8.23	3064.38	9656.37
BinaryDiff	183.47	333.88	57.07	159.81	–	–	**357.16**	635.37
GeomDiff	57.83	3281.33	**0.21**	8.20	**0.19**	5.22	419.63	1355.67

the smallest patch in the shortest time and is also the fastest to apply and undo.

Linestring geometries

For linestring geometries (Table 4), a total of 813,503 geometry pairs were checked for each algorithm. The mean number of vertices is 24, the 99th percentile 236. When it comes to performance, the GeomDiff algorithm is considerably slower to create patches, albeit with a large standard deviation, but it is still the fastest on create and undo time. The JsonDiff algorithm is the fastest to create patches, but the patches created by the JsonDiff algorithm are on average larger than patches created by the BinaryDiff algorithm by a factor of 8.5.

Polygon geometries

For polygon geometries (Table 5), a total of 433,776 polygon pairs with a mean vertex count of 28 (99th percentile 299) were checked. In terms of performance, the polygon dataset exhibits much the same trends as the linestring data. The standard deviations are large, and the BinaryDiff and GeomDiff algorithms are considerably slower than TextDiff and JsonDiff when it comes to create time, but at the same time they produce the smallest patches.

Error counts

The error counts (Table 6) show that the GeomDiff algorithm encountered 22 and 34 create errors, and 33 and 45 patch and undo errors on linestrings and polygons, respectively. The TextDiff algorithm failed to undo 38,480 linestring pairs (5%) and 18,396 (4%) polygon pairs correctly.

For point geometries the rates are close to zero (< 1 ‰) for all metrics.

The create errors for the TextDiff algorithm are all "Invalid URI: The Uri string is too long.". This error originates in the Diff Match Patch library, which uses URL encoding provided by the C# standard library. This shows that the limiting factor for string lengths, and by extension vertex count, are the URL encoding method.

For the GeomDiff algorithm, all create errors are "Timed out after 60000 ms". This is a hard limit built into the GeomDiff library to avoid long-running operations to block for an unreasonable amount of time.

Vertex number effects

For linestring and polygon geometries, the GeomDiff algorithm exhibits an unusually large standard deviation on the Create Time metric. In order to investigate possible causes for this, we identified the upper 99 percentile and removed observations with values higher than this. This is shown in Table 7. We see that by removing 1% of the observations the standard deviation is reduced by two orders of magnitude.

One possible explanation for this is that the create time for the GeomDiff algorithm increases as the number of geometry vertices increase. This explanation is supported by the create failures on 22 linestring and 34 polygon geometries. In these cases, the algorithm ran for 60 s before timing out. Examining the geometries which caused the errors, we find an average vertex count of 1677 and 1576 for linestrings and polygons, respectively. For the top 1 (slowest) percentile, the vertex count averages were 300 and 364. These numbers are both a substantial increase from the full population, which on average has a vertex count of 24 for linestrings and 28

Table 5 Benchmark results for polygon geometries. Best results in each case in bold

Algorithm	Create Time (ms)		Apply Time (ms)		Undo Time (ms)		Patch Size (b)	
	Mean	St.dev	Mean	St.dev	Mean	St.dev	Mean	St.dev
TextDiff	7.53	70.12	1.08	39.82	0.92	7.22	481.37	2023.27
JsonDiff	***3.50***	76.01	1.15	20.80	0.95	10.60	2970.73	15,035.43
BinaryDiff	224.40	571.71	69.11	272.37	–	–	***301.82***	684.04
GeomDiff	118.09	5159.74	***0.39***	79.77	***0.25***	7.02	306.00	1397.86

Table 6 Error counts for the tested algorithms, grouped by geometry type and operation. A create error represents a situation where the algorithm threw an exception during execution, while apply and undo errors represents situations where applying or undoing a diff does not produce the expected geometry

Algorithm	Point			Linestring			Polygon		
	Create	Patch	Undo	Create	Patch	Undo	Create	Patch	Undo
TextDiff	0	1	4	3	3	38,480	1	1	18,396
JsonDiff	0	1	1	0	0	0	0	0	0
BinaryDiff	0	1	–	0	0	–	0	0	–
GeomDiff	0	1	1	22	33	33	34	45	45

for polygons. In other words, large vertex counts seem to indicate long running times.

To further investigate whether the vertex count variable influences create time, we calculated the Pearson correlation coefficient [27] between creation time and vertex count, as shown in Table 8. We see that the correlation change between the whole population and the top 1 percentile is substantial for the GeomDiff algorithm (+ 0.17 / + 0.81), while it is relatively stable or decreasing for the other algorithms (- 0.02 / - 0.01 for the TextDiff algorithm). Thus, we suspect that the vertex count in linestring and polygon geometries affects the creation time for the GeomDiff algorithm significantly, and especially for large numbers of vertices.

By grouping the create time results by vertex count and computing average creation time for each group (Fig. 2 and Fig. 3), we find that all algorithms except the BinaryDiff algorithm show an increase in creation time with increasing number of vertices. However, for the GeomDiff algorithm, there is a sharp increase when exceeding a vertex count of 500, for both linestrings and polygons.

Discussion

Our data shows that the GeomDiff algorithm outperforms the other tested algorithms by a large margin when working with point geometries. It creates the smallest patches in the shortest time and is also fastest at applying and undo patches.

When it comes to more complex geometries (Table 4 and Table 5), the results are more varied. The JsonDiff algorithm is the fastest for creating both linestring- and polygon-patches, while the BinaryDiff algorithm creates the smallest patches. However, the JsonDiff algorithm creates the largest patches, while the BinaryDiff algorithm is the slowest one in both creation and apply time. Moreover, this algorithm does not support the undo operation.

The results for the GeomDiff algorithm with regards to linestrings and polygons are more complex. The algorithm is the fastest on both apply and undo, and it produces patches not much larger than the BinaryDiff algorithm. However, the creation time shows a large variance. Based on our test data, we found that this is related to number of vertices in the diffed geometries. When this exceeds 500 vertices, we see a sharp increase in creation time. In addition, we recorded several occurrences where the algorithm timed out after 60 s for some geometries with large vertex counts.

However, both the mean and 99th percentile of vertex counts in both linestrings and polygons are considerably lower than 500 in the OSM test-dataset. This means that, for datasets comparable in complexity to OSM, the vertex issue is not likely to be major. In addition, diffs are usually created only once, but applied and undone multiple times. Thus, faster apply and undo speeds are more important than creation times. Nevertheless, the fact that the GeomDiff algorithm degrades, and sometimes fails, on geometries with a high vertex count is not ideal. This behaviour is worth determining the cause of and remedy before the algorithm can be considered ready to use in a real-life situation where performance and repeatability is essential.

Table 7 Create time for linestring and polygon geometries with the upper 99 percentile values excluded from the analysis

Algorithm	Linestring		Polygon	
	Mean	St.dev	Mean	St.dev
TextDiff	4.60	14.14	2.72	9.20
JsonDiff	1.12	2.46	0.99	2.33
BinaryDiff	165.05	180.63	180.43	195.48
GeomDiff	2.44	12.68	1.28	7.62

Table 8 Pearson correlation coefficient between creation time and vertex count for the full population and the top 1 percentile

Algorithm	Linestring		Polygon	
	All	Top 1%	All	Top 1%
TextDiff	0.50	0.48	0.45	0.44
JsonDiff	0.30	0.23	0.27	0.16
BinaryDiff	0.01	0.03	0.01	-0.01
GeomDiff	0.26	0.43	0.30	0.48

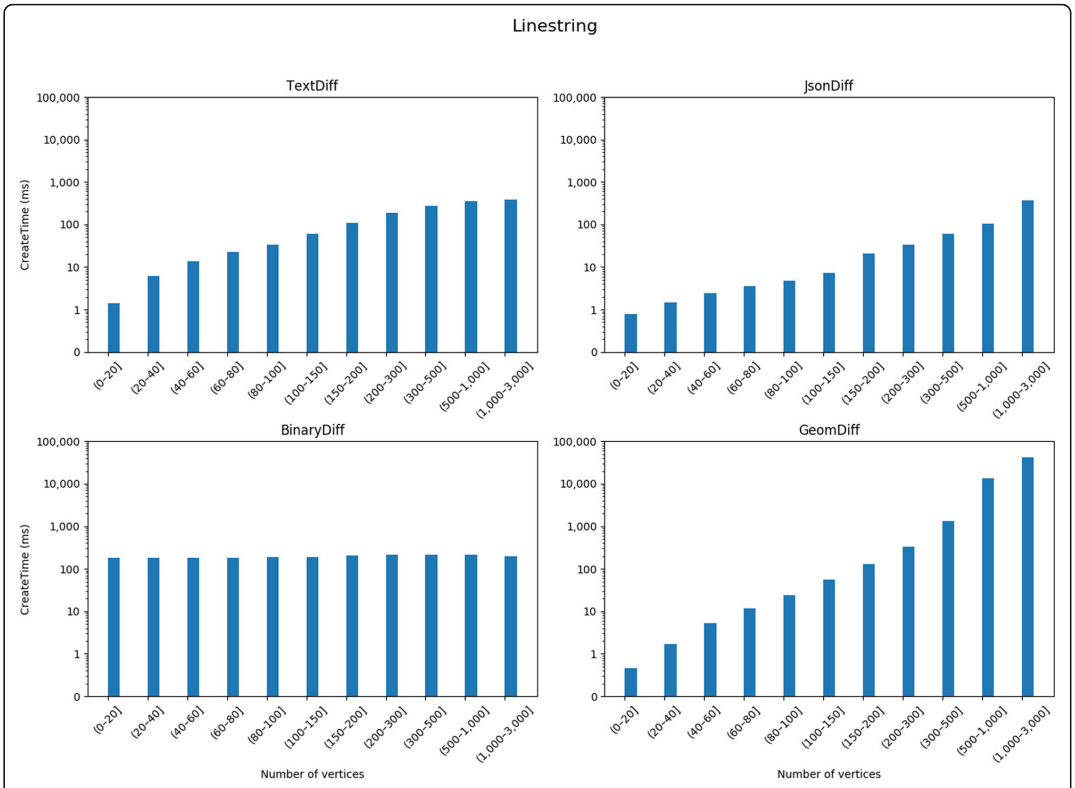

Fig. 2 Average create time for linestring patches, grouped by vertex count

The error rates are low for all algorithms, except for the TextDiff undo algorithm. One possible explanation for these errors are floating-point issues. Since the TextDiff algorithm uses the text based WKT format as an intermediary step, it is possible that some rounding errors have been introduced when the undo operation is applied. However, we have not investigated this issue further.

One shortcoming of our experiment is that the testdataset did not include multi geometries. This is because the OSM dataset does not contain multi points and multi linestrings, and that multi polygons were considered too time-consuming to create from the OSM data format. However, we suspect that multi-geometries will show results similar to or worse than linestrings and polygons. Since multi-geometries adds more layers to the recursive hierarchy of components, more time will be spent traversing this hierarchy.

The use of a commercial cloud platform as the testbed for our experiment allowed us to test a large number of operations in parallel at a reasonable price. This would have been costly and complex to achieve using on-site hardware. However, each execution of a function app runs on an instance. This instance runs multiple concurrent executions in parallel, which mean that executions may compete for the same CPU resource [28]. This may affect the performance of each execution, compared to running them in complete isolation. However, we argue that the large amount of geometries tested will mitigate this issue and spread the effect evenly.

Conclusions

We have shown that efficient diffing algorithms for geospatial vector data can be created by taking advantage of the native mathematical properties of the data. The GeomDiff algorithm performs comparable to, or better than, the three generic diffing algorithm we have compared it to. However, it suffers from performance degradation as the vertex count increases. In many situations this will not pose a problem, but it is a serious shortcoming that should be addressed.

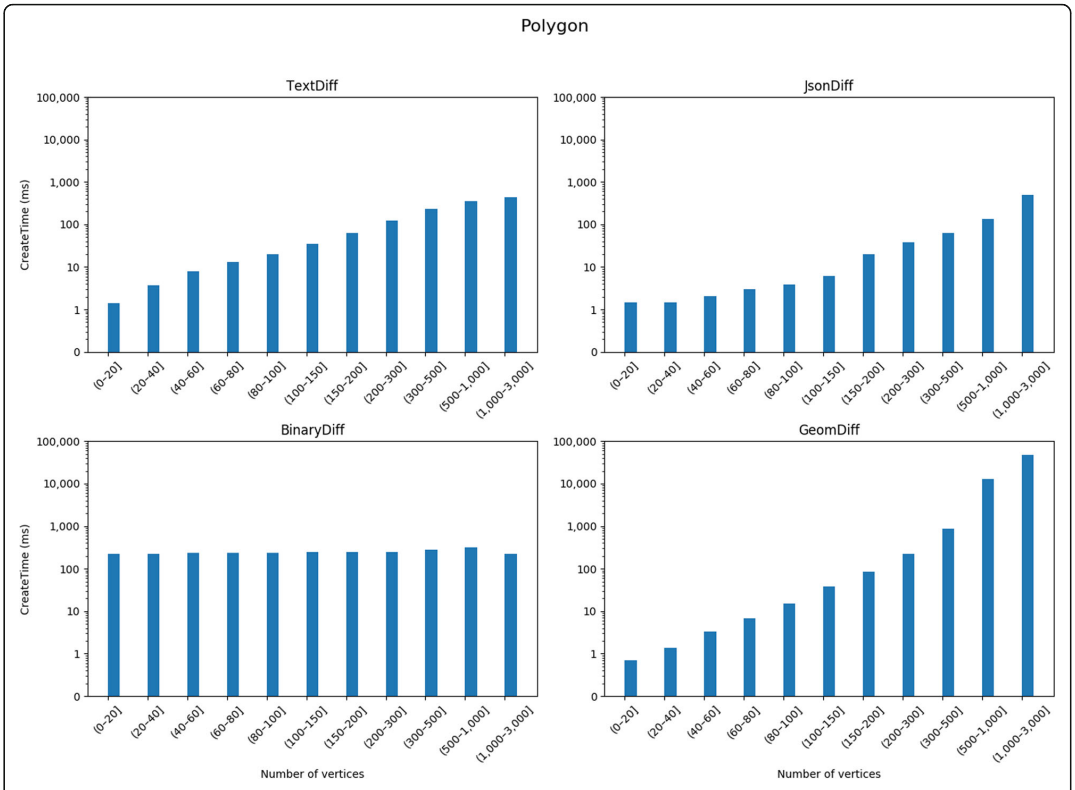

Fig. 3 Average create time for polygon patches, grouped by vertex count

Geospatial diffing formats have several use-cases. Storage of spatiotemporal data is one example. In the "object change" model described by Worboys [29], significant storage reductions can be achieved by storing each change as a diff as opposed to storing the complete, changed version. Geospatial diffs will also be a key concept when designing a system that uses the principles of Event Sourcing [30] to handle geospatial data.

We have not found any published geospatial diffing algorithms in the literature. However, we found some examples from the industry. We suspect that this indicates that if work on this topic has been carried out, it has been done in the industry. Another possibility is that when the need for geospatial diffing has occurred, generic diffing algorithms have been found sufficient. In the light of our findings we question this conclusion, as we have shown that it is possible to create tailored algorithms for geospatial diffing that outperforms generic algorithms.

With the increasing amount of geospatial data being collected, created, and updated we foresee an increased demand for efficient strategies for storage and processing. Event sourcing and object change models are an interesting approach to this challenge. Since geospatial diffing algorithms are a key aspect of these techniques, we encourage more research into this field.

Availability and requirements

- **Project name:** GeomDiff
- **Project home page:** https://github.com/atlefren/GeomDiff
- **Operating system(s):** Platform independent
- **Programming language:** C#
- **Other requirements:** .NET Standard 2.0 compatible .NET implementation
- **License:** BSD-3-Clause
- **Any restrictions to use by non-academics:** No restrictions apart from those imposed by the license.

Abbreviations

JSON: JavaScript Object Notation; OSM: OpenStreetMap; WKB: Well Known Binary; WKT: Well Known Text; XML: Extensible Markup Language

Acknowledgements
The author would like to thank Dr. Birgitte H. McDonagh, Dr. Alexander S. Nossum, and Dr. Terje Midtbø for valuable input during manuscript preparation.

Author's contributions
A.F.S is the sole author of this manuscript, and carried out the implementation of the GeomDiff library, all supporting scripts, performed the statistical analysis, and wrote the manuscript. The author read and approved the final manuscript.

Funding
This work was supported by Norkart AS and The Research Council of Norway [grant number: 261304 I].

Availability of data and materials
The data generated by the benchmark pipeline is available at https://github.com/atlefren/geomdiff_article_results
The OpenStreetMap data used by the benchmark are available at http://download.geofabrik.de [24]
The derived OpenStreetMap geometry pairs used in the benchmark are available from the corresponding author on reasonable request.

Competing interests
The author declare that they have no competing interests.

Received: 11 May 2020 Accepted: 26 June 2020
Published online: 10 July 2020

References

1. Raymond ES. The jargon file, version 4.4.7. 2003 [cited 2019 Nov 11]. Available from: http://catb.org/jargon/html/D/diff.html.
2. Ruparelia NB. The history of version control. SIGSOFT Softw Eng Notes. 2010; 35(1):5–9.
3. D'Angelo G, Di Iorio A, Zacchiroli S. Spacetime Characterization of Real-Time Collaborative Editing. Proc ACM Hum-Comput Interact. 2018;2(CSCW):41 1–41:19.
4. Michelson B. Event-Driven Architecture Overview. Patricia Seybold Group [Internet]. 2006;2. Available from: http://www.customers.com/articles/event-driven-architecture-overview/.
5. Coup R. Sno, our new open source tool for distributed data versioning. Koordinates Blog 2020. Available from: https://koordinates.com/blog/sno-our-new-open-source-tool-distributed-data-versioning/.
6. Singh A. Review article digital change detection techniques using remotely-sensed data. Int J Remote Sens. 1989;10(6):989–1003.
7. Cho J, Ntoulas A. Chapter 45 - Effective Change Detection Using Sampling. In: Bernstein PA, Ioannidis YE, Ramakrishnan R, Papadias D, editors. VLDB '02: Proceedings of the 28th International Conference on Very Large Databases. San Francisco: Morgan Kaufmann; 2002 [cited 2020 Mar 18]. p. 514–25. Available from: http://www.sciencedirect.com/science/article/pii/B9781558608696500524.
8. Hunt JW, Mcllroy MD. An algorithm for differential file comparison. Bell Laboratories Murray Hill; 1976. (Bell Laboratories Computing Science). Report No.: #41.
9. Miller W, Myers EW. A file comparison program. Software: Practice and Experience. 1985;15(11):1025–40.
10. Myers EW. AnO (ND) difference algorithm and its variations. Algorithmica. 1986;1(1):251–66.
11. Chawathe SS, Rajaraman A, Garcia-Molina H, Widom J. Change Detection in Hierarchically Structured Information. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data. New York: ACM; 1996 [cited 2019 Aug 27]. p. 493–504. (SIGMOD '96). Available from: http://doi.acm.org/10.1145/233269.233366.
12. Percival C. Naive differences of executable code. 2003 [cited 2020 Jan 15]. Available from: http://www.daemonology.net/bsdiff/.
13. Nottingham M, Bryan P. JavaScript Object Notation (JSON) Patch. 2013 [cited 2020 Mar 17]. Available from: https://tools.ietf.org/html/rfc6902.
14. Urpalainen J. An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors. 2008 [cited 2020 Mar 18]. Available from: https://tools.ietf.org/html/rfc5261.
15. Razmjooei S. Tracking, calculating and merging vector changes with Input and QGIS - Lutra Consulting. Lutra Consulting. 2019 [cited 2020 Mar 20]. Available from: https://www.lutraconsulting.co.uk/blog/2019/11/23/input-geodiff/.
16. Duffy L. Boundless GeoGit: A Different Approach to Geospatial Data Management. POB. 2014 [cited 2020 Mar 18]. Available from: https://www.pobonline.com/articles/100258-boundless-geogit-a-different-approach-to-geospatial-data-management?v=preview.
17. ISO. ISO/IEC 13249–3:2016 Information technology — Database languages — SQL multimedia and application packages — Part 3: Spatial. 5th ed. 2016. 1328 p.
18. Haklay M, Weber P. OpenStreetMap: user-generated street maps. IEEE Pervasive Computing. 2008;7(4):12–8.
19. Atle Frenvik Sveen. node-osm-read. 2020. Available from: https://github.com/atlefren/node-osm-read.
20. Sveen AF. BenchmarkFunction. 28.01.20202 [cited 2020 Jan 28]. Available from: https://github.com/atlefren/GeometryDiffBenchmark.
21. NetTopologySuite. 2019. Available from: https://github.com/NetTopologySuite/NetTopologySuite.
22. Gillies S, Butler H, Daly M, Doyle A, Schaub T. The GeoJSON Format. 2016 [cited 2020 Mar 19]. Available from: https://tools.ietf.org/html/rfc7946.
23. Sveen AF. geodiff_stats. 2020 [cited 2020 Apr 10]. Available from: https://github.com/atlefren/geodiff_stats.
24. D'agostino R, Pearson ES. Tests for departure from normality. Empirical results for the distributions of b2 and √b1. Biometrika. 1973;60(3):613–22.
25. Kruskal WH, Wallis WA. Use of ranks in one-criterion variance analysis. J Am Stat Assoc. 1952;47(260):583–621.
26. Conover WJ, Iman RL. On multiple-comparisons procedures. Los Alamos Sci Lab Tech Rep LA-7677-MS. 1979;1:14.
27. Lee Rodgers J, Nicewander WA. Thirteen ways to look at the correlation coefficient. Am Stat. 1988;42(1):59–66.
28. Shilkov M. From 0 to 1000 Instances: How Serverless Providers Scale Queue Processing. Binaris Blog. 2018 [cited 2020 May 8]. Available from: https://blog.binaris.com/from-0-to-1000-instances/.
29. Worboys M. Event-oriented approaches to geographic phenomena. Int J Geogr Inf Sci. 2005;19(1):1–28.
30. Fowler M. Event Sourcing. martinfowler.com. 2005 [cited 2020 Apr 16]. Available from: https://martinfowler.com/eaaDev/EventSourcing.html.

Publisher's Note

ISBN 978-82-326-4816-0 (printed ver.)
ISBN 978-82-326-4817-7 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

NTNU

Norwegian University of
Science and Technology