Master's thesis

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

Lars Furu Kjelsaas

A cloud-based pipeline for Event Sourcing of geospatial data

Master's thesis in Engineering & ICT
Supervisor: Terje Midtbø, Atle Frenvik Sveen
May 2020

NTNU
Norwegian University of
Science and Technology

Lars Furu Kjelsaas

A cloud-based pipeline for Event Sourcing of geospatial data

Master's thesis in Engineering & ICT
Supervisor: Terje Midtbø, Atle Frenvik Sveen
May 2020

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

NTNU
Kunnskap for en bedre verden

NTNU

Fakultet for ingeniørvitenskap
Institutt for bygg og miljøteknikk

Masteroppgave

(TBA4925 - Geomatikk, masteroppgave)

Vår 2020
for
Lars Furu Kjelsaas

A cloud-based pipeline for event-sourcing of geospatial data

BAKGRUNN

For å utvikle gode IT-løsninger basert på geografiske data er datatilgang viktig. De rette datasettene må være tilgjengelig i rett format for hver enkelt applikasjon. Løsninger må også kunne skalere på en sømløs måte.

En tradisjonell arkitektur der geografiske (vektor) data lagres i en romlig database skalerer dårlig, både med tanke på lese- og skrive-hastighet, samt data-volum. Hvis man i tillegg ønsker å bevare historikk i dataene øker lagringsbehovet med en størrelsesorden.

I en arkitektur basert på Event Sourcing distribueres endringer på data på en hendelses-kø, som en eller flere abonnenter følger med på for å for å holde rede på de dataene de er interessert i. I en geografisk kontekst kan dette være data fra ett eller flere datasett, filtrert på både geografisk område og attributtinformasjon. En slik arkitektur har potensiale til å løse utfordringene knyttet til både hastighet, volum og historikk, samt innføre nye muligheter for overvåkning.

OPPGAVEBESKRIVELSE

Studenten skal gi en oversikt over hvordan Event Sourcing kan brukes på geografiske vektor-data. Dette innebærer å redegjøre for lignende løsninger, og peke på styrker og svakheter. En pipeline for å sette opp en slik løsning på en kommersiell sky-plattform skal implementeres og testes. Denne skal være konfigurerbar, slik at forskjellige komponenter kan inngå ved behov. Selv om geografiske data fra en rekke kilder, som data-strømmer fra sensorer og eksisterende endringslogger kan inngå i en slik pipeline er hovedfokus på ikke-versjonert data. Dermed må systemet kunne håndtere algoritmer for å detektere slike endringer.

Studenten skal også redegjøre for utfordringer en slik arkitektur innebærer, spesielt med hensyn til sikkerhet og etterprøvbarhet av data.

I dette inngår at studenten skal:

- Studere litteratur relevant for det aktuelle temaet.
- Undersøke og eventuelt vurdere eksisterende løsninger.
- Utvikle nødvendige komponenter for å lage en slik pipeline, samt lage grensesnitt for å bruke eksisterende algoritmer der disse finnes.
- Gjennomføre kvantitative eksperimenter i en software-lab-setting for å understøtte funnene.
- Reflektere over og redegjøre for utfordringer knyttet til sikkerhet og etterprøvbarhet.
- Peke på nye muligheter en slik arkitektur tilrettelegger for.

ADMINISTRATIVT/VEILEDNING

Arbeidet med masteroppgaven startet 2. januar, 2020

Masteroppgaven skal leveres digital på Inspera innen 11. juni, 2020

Veiledere ved NTNU:

Professor Terje Midtbø

Atle Frenvik Sveen

Oslo, mai, 2020

1 Abstract

Many geospatial datasets are updated in bulk, and new versions are published as full, new datasets. For some applications, the historical perspective and the change of data over time is vital information for the use of the data.

By introducing Event Sourcing, the changing nature of the underlying data can be presented more accurately and efficiently. By storing events that contain the changes to objects, rather than storing static states, more temporal information can be represented. The approach may also enable new ways of using and distributing the data to other systems.

To transform snapshotted data to an event stream of changes, one must compute the difference between two versions of a dataset to capture the changes. This is computation intensive for large datasets, especially when working with the nuances of geospatial objects. As differences could be represented on a single-object basis, parallelizing the computation could help handle the amount of computation required in a timely fashion.

Azure Functions and other serverless architectures represent a new paradigm of cloud services that promises scaling of resources as need arises without having to worry about the setup of underlying server hardware. The ability to scale from zero to massive, parallel processing seems well fit for the scheduled, large processing jobs that is required for Event Sourcing large geospatial datasets.

A pipeline was built using a durable orchestration function that handled dataflow between multiple functions. This allowed processing to be parallelized over clusters of virtual machines. By utilizing open standards and defined data objects, the pipeline was made with modularity in mind, with the possibility of swapping out components if the need arises in the future.

Initial results from the implemented pipeline show promising results, but more work is needed to find the optimal setup. The system can process hundreds of updates, but struggle when the number of required updates increase beyond this. One possible route for scaling capacity further is to introduce multiple layers of orchestrators to further divide up the parallel workflow. This thesis has experimented in the crossing point between geocomputation, Event Sourcing and serverless computing and discovered some possibilities. Further work should reveal interesting results, as the combination has many synergies and similarities.

2 Sammendrag

Mange geografiske datasett blir oppdatert samlet, med jevnlig publisert av fullverdige, nye versjoner. For en del applikasjoner er imidlertid det historiske perspektivet og endringen av data mellom versjoner viktig informasjon for å kunne nyttiggjøre dataene.

Ved å bruke Event Sourcing kan dette representeres mer nøyaktig og effektivt. Ved å lagre hendelser som inneholder endringene til objekter, istedenfor statiske data, kan mer informasjon om tid bli lagret. Tilnærmingen kan også muliggjøre nye måter å bruke og distribuere dataene på mellom systemer.

For å transformere stillbilde-data til en hendelsesstrøm med endringer, er det nødvendig å beregne forskjellen mellom datasett for å fange opp endringene. Dette er beregningsintensivt for store datasett, spesielt når fokuset ligger på endringsnyanser til geografiske objekter. Ettersom forskjeller kan bli representert på enkeltgjenstandsbasis, kan parallellisering av beregningen bidra til å øke gjennomstrømmingen av data, og dermed redusere kjøretiden.

Azure Functions og andre serverløse arkitekturer er et nytt paradigme innen skytjenester, som tilbyr skalering av ressurser etter behov uten å måtte bekymre seg for oppsettet av underliggende maskinvare. Evnen til å skalere fra null til massiv, parallell prosessering virker godt egnet for de planlagte, store prosesseringsjobbene som er nødvendige for Event Sourcing av store geografiske datasett.

En prosesseringsløype ble laget ved hjelp av en «vedvarende orkestreringsfunksjon» som håndterte dataflyten mellom ulike funksjoner. Dette tillot parallellisering av prosessering over flere klynger med virtuelle maskiner, noe som ellers ville vært utfordrende. Ved å bruke åpne standarder og tydelig definerte dataobjekter ble prosesseringsløypen laget med tanke på modularitet, med mulighet for å bytte ut enkeltkomponenter ved behov.

De første resultatene fra den implementerte prosesseringsløypa viser lovende resultater, men det er nødvendig med mer arbeid for å finne et best mulig oppsett. Systemet kan behandle hundrevis av oppdateringer, men sliter når antall oppdateringer øker utover dette. En mulighet for å skalere kapasiteten videre er å introdusere flere lag med orkestreringsfunksjoner, for å dele den parallelle arbeidsflyten ytterligere. Denne avhandlingen har utforsket krysningspunktet mellom geografiske beregningsmetoder, Event Sourcing og serverløs databehandling og kommet fram til lovende funn. Videre arbeid bør kunne avdekke interessante resultater, siden teknologiene har flere likheter og mulige synergier.

3 Preface

This paper is written as the final work of a Master of Science degree in Engineering and ICT with a specialization in Geomatics, at Norwegian University of Science and Technology (NTNU), Department of Civil and Environmental Engineering.

I would like to thank Professor Terje Midtbø. Not only for his role as supervisor, but also for his mentorship during my years at the Geomatics specialization, introducing me to the Norwegian Geomatics community and allowing me opportunities I would otherwise not have had.

My thanks also go to co-supervisor Atle Frenvik Sveen of NTNU and Norkart AS. He presented the initial idea of the thesis and provided great feedback on both application and thesis throughout the process. Importantly, he also provided the dataset used in a format fit for the project, and a cloud subscription that I could use for my application. I hope to have more discussions about Event Sourcing of geospatial data and cloud computing with you in the future.

Thanks to Eileen for keeping me sane in the final weeks of the work. Thanks to Kari, for help with academic research, layout and proofreading. Thanks to Kjartan for an insightful presentation of the concept of Event Sourcing.

A final thanks goes to very helpful university employees, the librarians for help in finding relevant literature, and the administrative workers for quick help along the way.

Oslo, May 21st, 2020

Lars Furu Kjelsaas

Table of Contents

List of figures

List of tables

4 Introduction

Most geospatial datasets are updated, revised, or in other ways changed over time. The frequency of these updates varies from “almost never” to instantaneous. Many of the most central governmental geospatial datasets in Norway, and abroad, are updated on a monthly or bi-monthly schedule, in a bulk-update fashion, where a new version of the dataset completely replaces the old. For many applications, this is a viable solution. In other applications there may be no need to keep track of changes, as the underlying data doesn’t change fundamentally over time and older data is fine. However, for some applications the historical perspective is vital, and we want to represent both the past and the present in an efficient manner.

Event-sourcing is one method for keeping the historical perspective. Storing the changes to data rather than static objects lets us represent the dynamic nature of data. How can we transform traditional data structures of versioned databases of objects to an event store in an efficient manner? This question forms the underlying motivation for this thesis. More specifically, this thesis will attempt to answer this question by answering the following research questions:

1. Is a cloud-based implementation of a diff-based event generation pipeline a viable solution for producing an event-stream from traditional, bulk-updated data?
2. How can such a pipeline be implemented in a modern, cloud-based computing environment?
3. How does such a pipeline perform in terms of scalability and throughput?
4. How does an event generation pipeline for versioned geospatial data fit into a larger software architecture in terms of integration of data consumers?

These questions are answered in the following. To establish a common platform of understanding, the first part is a review of relevant theory and practice. This includes relevant research done in the fields of geospatial and spatiotemporal representation and processing, event-based and event-sourced architectures, database technology and cloud computing.

The findings of the background section form the groundwork for the Method & Pipeline Implementation section, in which the actual implementation of a difference-based event generation pipeline is presented and explained. This part covers the major architectural decisions made during the implementation and presents the final pipeline. Then, relevant results and takeaways from the work is presented before an overall discussion of the work and possible future work are described.

5 Background

5.1 Geospatial data

Spatial, or geospatial, data is a term used to describe data where the position, shape and/or size in the world is an important part. This could be anything from a list of points representing traffic signs along a road, polygons representing different types of soil, and so on. The term is usually used when the data is analyzed or presented in a spatial context (Worboys & Duckham, 2004).

Geospatial data are used in a wide variety of circumstances, like ridesharing services keeping track of vehicles and users (Wang, 2017), predicting future landslides (van Westen et al., 2008) or epidemiologists monitoring diseases (Pfeiffer et al., 2008).

5.1.1 Representing geospatial data

How geospatial data is represented is important for facilitating the storage, processing, and visualization of them. The representation is central to how a problem can be solved, and how easy such a solution is. While geospatial data can be represented in several different ways, the two major representations are the raster format and the vector format (Peuquet, 2002).

By dividing the relevant geographical area into an array of grid cells with varying values representing some real-world phenomenon, the raster format is created. The vector format is structured as a series of vectors with coordinate values in two (or three) dimensions. The raster format corresponds to a field-based representation, while the vector format is an object-based representation.

In fact, it has been shown that discrete objects and continuous fields are the only possible bases for describing the geographic world if the foundation is aggregation of point sets (Goodchild et al., 2007).

While a field-based representation is excellent for representing statistical and demographic data, such as median income, average rainfall or number of inhabitants in an area, object representation is more fit for translating objects such as roads or houses into the virtual world.

One aspect to note is that a vector dataset can represent a field-based view of the world, such as when dividing the world into non-overlapping polygons of data such as administrative or statistical divisions.

5.1.2 OGC Simple Features

Open Geospatial Consortium (OGC) is “[…] a worldwide community committed to advancing geospatial location information and services as a vital force for progress” (OGC, 2020). As an international organization comprised of many actors within the geospatial community, OGC seek to establish international standards for geospatial representation. A widely adopted standard is OGC Simple Features, outlining an object-based representation of different kinds of features. Features are in this work defined as abstractions of real-world phenomena, like a road, a lamppost or a forested area (OGC, 2010).

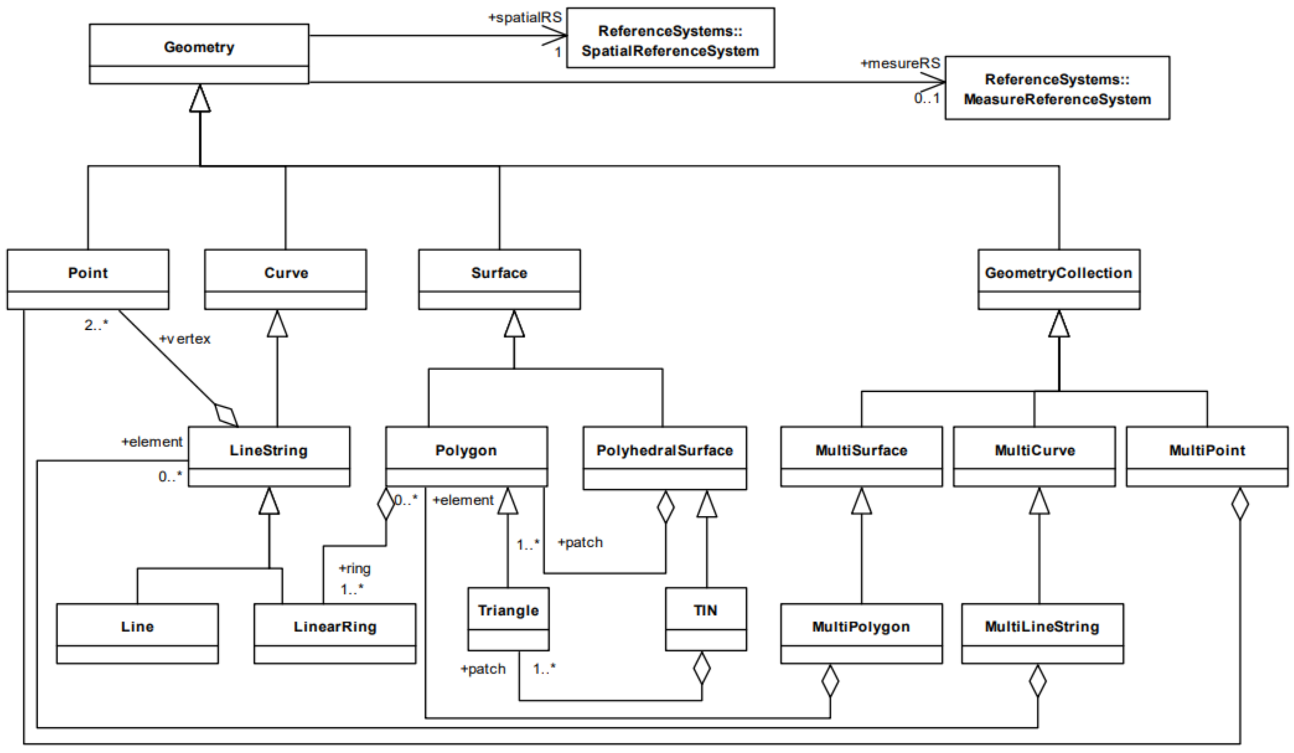


Figure 1 - Geometry class hierarchy, OGC Simple Features (OGC, 2010)

5.1.2.1 Geometry

Geometry is the root class of the system, an abstract class holding common functionality that the different kinds of geometries can inherit from. All geometries have one or more points with coordinate values in a reference system.

5.1.2.2 Points

The core building block of the OGC Simple features and most other object-based representations is its smallest part, the 0-dimensional geometric object called *Point*. A *Point*, like in a mathematical representation of points, has no area, no length, circumference, or such.

It represents a single, discrete point in space. The more complicated structures are built up of multiple points and the increasingly complex relations these can have.

5.1.2.3 Curves

Curves are used to represent 1-dimensional lines, usually structured by a series of vertices represented by points and an interpolation between them. In OGC Simple Features, only the *LineString* subclass exists, which draws straight lines as edges between the vertices to form a series of lines. A pair of *Points* defines each *Line* representing a line segment.

5.1.2.4 Surfaces

"Surface is a 2-dimensional geometric object." (OGC, 2010), although the standard outlines two subclasses, the most relevant for this thesis is *Polygon*. This is an area defined by a line forming a continuous, exterior boundary and any number of "holes" inside of it (Figure 2).

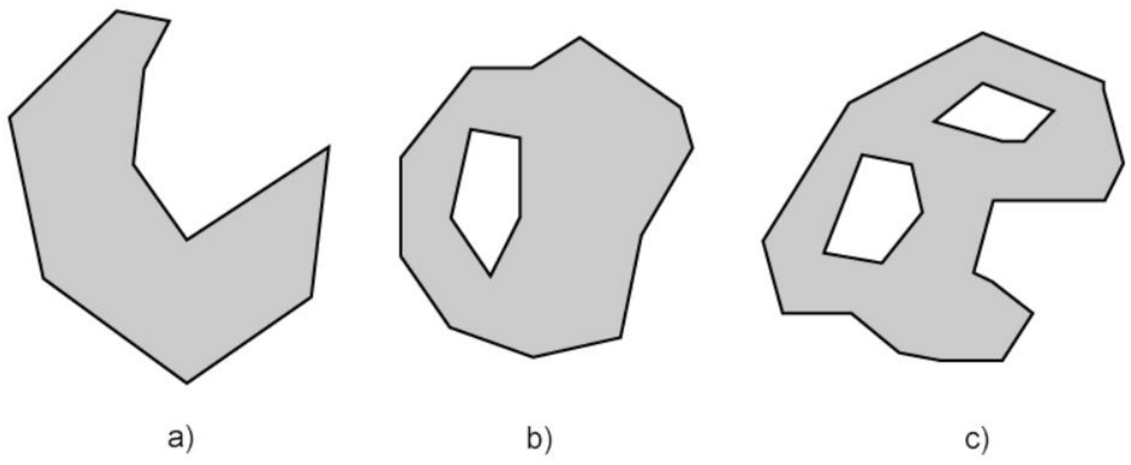


Figure 2 - Examples of polygon surfaces with 0 (a), 1 (b) and 2 (c) "holes" (OGC, 2010)

5.1.2.5 GeometryCollections

"A GeometryCollection is a geometric object that is a collection of some number of geometric objects" (OGC, 2010). It has different subclasses specifically containing objects such as *MultiPoint*, *MultiLineString* and *MultiPolygon*. *GeometryCollection* and its subclasses are used to represent more complex geographic features, as well as collections of multiple other objects.

5.1.3 Well-Known Text (WKT) and Well-Known Binary (WKB)

Together with OGC Simple Features, the text markup language Well-Known Text (WKT) was created to represent vector geometry objects. A more compact, binary version, Well-Known Binary (WKB), can be used for data transfer and storage (Stolze, 2003).

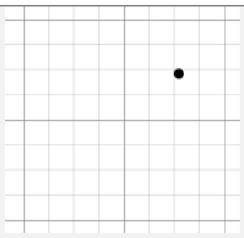
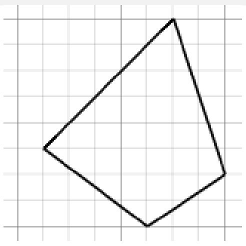
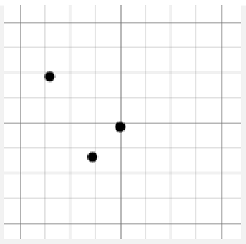
Simple Features	***Example***	***WKT representation***
Point		POINT (20 15)
Polygon		POLYGON (20 15, 15 20, 10 40, …)
Multipoint		MULTIPOINT ((20 15), (25 20), (10 40))

Table 1 - Examples of representation of some OGC Simple Features as Well-Known Text

5.1.4 Traditional storage of geospatial data

Storage of geospatial data, and most structured data for that matter, is often tied to a database, which is "[…] a collection of data organized in such a way that a computer can efficiently store and retrieve the data" (Worboys & Duckham, 2004). Efficient storage and retrieval of data depend not only on properly structured data in the database to provide satisfactory performance, but also optimized structures, representations, and algorithms for operating on data (Worboys & Duckham, 2004). A spatial database system is a database with additional capabilities for handling large volumes of spatial objects. Storage structures, indexing and retrieval of data and manipulation of data must be adapted for the additional spatial context.

According to Schneider (2017), a spatial database should be able handle the following requirements:

1. The spatial database system should include the features normally in a traditional database system and build further upon this foundation.
2. It should offer spatial data types as special data types for the representation of geographic objects with spatial data types. An example of this could be basing types on OGC Simple Features.
3. It should provide operations that can perform geometric computations on spatial objects.
4. It should provide spatial predicates that check relationships and other properties between spatial objects, such as topological relationships.
5. It should offer a spatial query language for spatial queries
6. The previous points should be implemented by providing effective data structures for spatial data types and efficient implementations of spatial operations, predicates, joins and indexes.

Some implementations of spatial databases are built as extensions to mature relational database management systems (RDBMS), while others exist on their own or as part of Geographic Information System (GIS) software.

5.1.5 Processing of geospatial data

Many geospatial analyses are computationally expensive. Because of this, research has been done within the geocomputation field on the subject of high-performance computing (Gahegan, 2017). Some of this work has been focused on finding better algorithms and tuning existing ones for better performance, but focus has also been on parallel computing, "[…] a computational technique in which multiple operations are executed at concurrently rather than sequentially" (Shekhar & Cugler, 2017).

By using the power of multiple Central Processing Units (CPUs) simultaneously, a result can be found much faster than when using a single core. However, converting an algorithm to utilize parallel computation is a non-trivial task. Depending on the specific task at hand, the challenge can vary from simple to impossible. An early work within the Geographic Information Science (GISc) field found promising results when processing spatial statistics (Armstrong & Marciano, 1995).

5.2 Temporal and spatiotemporal data

Temporal data is a term used to describe data with a temporal component, or a notion of time, attached. In other words, temporal data is data that changes over time in some way. The time-component can describe several types of change, such as incrementing or decrementing a numerical value, change of a textual value, or the movement of a point, change of size or other geographic feature change. Another form of temporal data are events happening along a time axis, which change the aggregated results from a "before" to an "after" state (Peuquet, 2002, 2017). Temporal data with a geospatial component is often referred to as spatiotemporal data.

5.2.1 Stages or degrees of spatiotemporal datasets

How changes over time is represented in spatial datasets have been subject to development. Traditionally, cartography had an inherent static view of the world, which according to Peuquet (2017) might have held back development. However, temporal elements have been introduced to geospatial datasets. The representation can be divided into four, distinct stages of increasing degree (Worboys & Duckham, 2004; Worboys, 2005).

5.2.1.1 Stage Zero: Static representations

A static representation contains no temporal dimension at all. As far as the dataset is concerned, the information within is static and unchanging. If changes are made to the dataset, all previous history is overwritten.

5.2.1.2 Stage One: Temporal snapshots

Snapshotting refers to capturing a moment in time, like when taking a snapshot with a camera. By storing multiple different versions of the same dataset at different times, one can represent the dataset at all the different times, both keeping some historical data and being able to compare data over time to infer some trends. Until recently, this was the most common approach for spatiotemporal models (Worboys, 2005).

However, no explicit information about the changes are stored. If the changes themselves is the important part to show, trying to parse differences from one snapshot to another might be difficult, depending on the changes. If a change has happened, there is also no information about when it occurred more precise than at some point in time between the snapshots.

The most naïve approach for incorporating a temporal dimension into geospatial data is to store each temporal snapshot of a dataset. However, this will quickly become unfeasible due to running out of space (Worboys & Duckham, 2004). For any number of snapshots, the space requirements are large. This is due to the necessity of storing unchanged features in every snapshot. Data duplication is generally to be avoided if possible, as the duplicates require extra space and makes updates to the data more complicated. Within RDBMSs this phenomenon is known as redundancy, and is generally to be avoided except when creating backup and recovery plans (Elmasri & Navathe, 2016). Redundancy in geospatial databases have been a focus when constructing spatial indexes (Gaede, 1995).

Although not necessarily stored as such, many datasets are published like snapshots. A new version of a dataset is published every month, every few months or after a given amount of work is done. The users then either overwrite previous data (stage zero representation) or store them as different snapshots (stage one representation).

Of course, a great example of snapshot data is actual snapshots in the form of aerial and satellite photos used for data collection. Another example would be storing old and new municipal borders after a change as separate datasets.

5.2.1.3 Stage Two: Object lifelines

By storing different versions of single objects, rather than the whole dataset, more granularity can be achieved.

This approach is called object lifelines and can be represented using objects indexed by their id and version, with only the latest version being shown to the end user. Using object lifelines, changes to an object like creation, destruction and adjustments can be represented explicitly and get connected to a specific time. There is also less redundant storage of data stored compared to snapshots, as only changed data gets a new object version. An example of a dataset structured in this manner is Open Street Map (Section 5.3).

5.2.1.4 Stage three: Events and actions

How does a representation of geospatial data look like when events and actions are used to represent static data rather than the other way around?

Mourelatos (1978) divides situations into states and occurrences, or actions. Occurrences are then divided into processes (activities) and events (performances). Events are divided into developments and punctual occurrences.

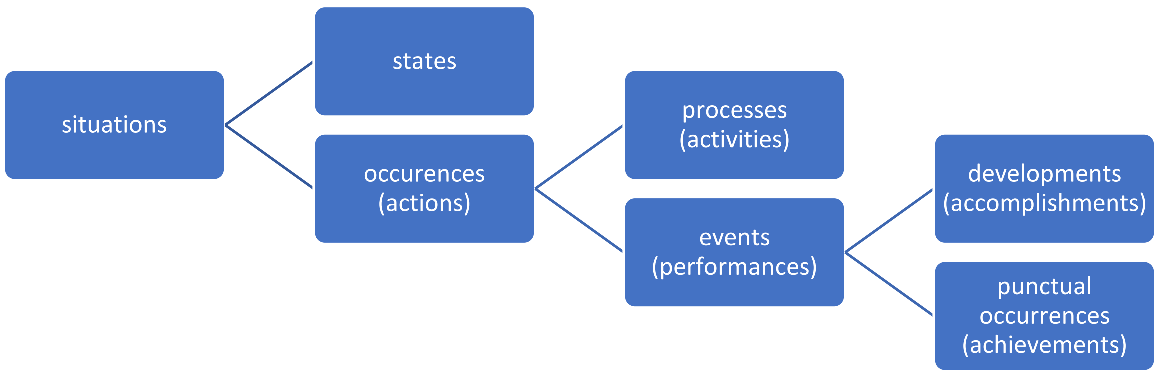


Figure 3 - Division of different "situations". (Adapted from Mourelatos 1978)

By representing our data as different kinds of occurrences (Figure 3), all situations can be represented. This is because objects go through states like creation, deletion and are changed in the form of updates. These changes can be "[…] described as an event or collection of events – something of significance that happens." (Peuquet, 2017). The change event is the focus, rather than the new state.

5.3 OpenStreetMap (OSM)

OpenStreetMap, or OSM, is one of the most extensive examples of crowd-sourcing of geospatial data (Haklay & Weber, 2008). OSM is an open-licensed world-spanning database of vector-based geospatial data. Data collection and editing follows the same crowd-sourcing principle which drives the online encyclopedia Wikipedia, where a collaboration of volunteers, each with small contributions, together create a large dataset. As existing OSM data is extended or corrected, new versions of already registered objects are stored as new versions of the object. This is the object timeline structure presented in section 5.2.1.3.

5.4 Event-driven architectures and Event Sourcing

Software applications based on the notion of events are not limited to geospatial data. A growing number of applications in use today trigger different functionality within software when an event occurs. These events can originate either from the outside world or within the system itself (Hohpe, 2006). An event within this context might be defined more practically than the theoretical approach exemplified by Mourelatos in Section 5.2.1.4. An event can simply be "a notable thing that happens inside or outside your business" (Michelson & Seybold, 2011). The focus is business-driven, and several pieces of central work has been published in the form of blog posts from industry veterans, rather than in peer-reviewed papers.

A more special grouping within event-driven architectures is Event Sourcing. In event-sourced systems, not only are events used for messaging and triggers, they are also used to represent the application state in storage. Debski et al. (2018) describes it as "An advanced version of commit-log". While writing a log file like Write-ahead-logging (WAL) (Mohan et al., 1992) implemented in some file systems can represent the same information, the key to Event Sourcing is that the events themselves represent the foundation, rather than being a backup log that can be used for recovery. By capturing all changes to the state as a sequence of events, the state itself can be represented (Fowler, 2005).

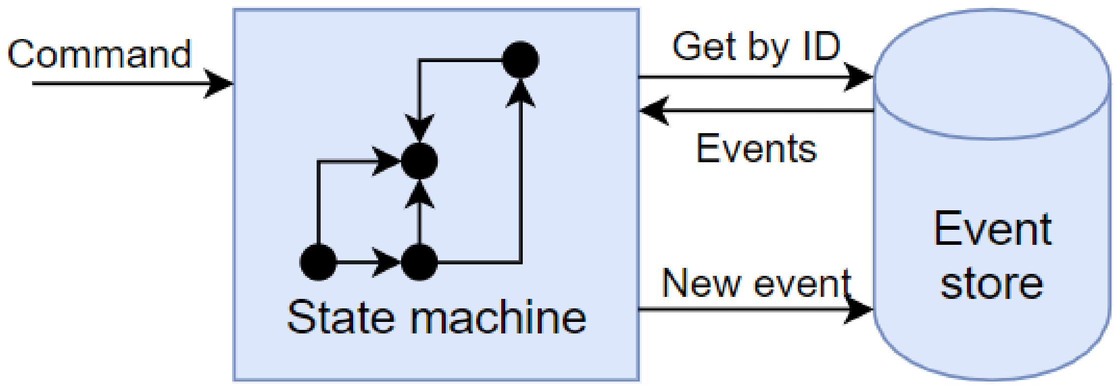


Figure 4 - Event sourcing pattern (Adapted from Debski et al. 2018).

Figure 4 shows the general layout of the Event Sourcing pattern. The state machine is responsible for calculating and representing the current state of the application. After the state machine receives a command, it requests all events stored for a given ID from the Event Store. These are passed back, and the events are applied to the state in chronological order, producing the current (or any requested) state for the ID. Any modifications to the state the command requires can then be written back to the store as new events.

This is similar to the envisioned Stage Three representation described in Section 5.2.1.4. Although little scientific work has been published on Event Sourcing, large companies like Netflix (Avery & Reta, 2017) has adopted the approach for complex, commercial tasks and it is described as a mature method (Debski et al., 2018).

Event-sourced systems are constructed using the concepts of append-only writing and immutable data, meaning that once something has been written it is never changed again. This has a few advantages. In domains such as banking, where systems handle sensitive and critically important data, it is imperative that an audit log is kept ensuring that the system is working as intended. By making the log the central piece of storage it can be guaranteed that the log is the correct sequence of events that happened in the system without elaborate testing, as it is the central source of facts (Young, 2014).

The event log is not only useful as an audit log, it also allows easier reproduction of software bugs that have occurred in the past, as one can reconstruct the state at the exact time the bug occurred after the fact. Running past events through a system can also be a very efficient way of testing new software versions.

If the immutable data is stored on a storage device that is also immutable, no tampering with the event history can be done. Keeping a full log of all events and actions within a system with no way of changing it is a good security measure when planning for so-called superuser attacks, where someone with administrator access tries to misuse or sabotage the system (Young, 2014).

A challenge that event-sourced architectures face, is that as the event history gets longer, constructing the current state takes longer time as well. In practice, this problem is fixed by combining the approaches of Stage One and Stage Three representations (Section 5.2.1), snapshots and events. As data is fetched, at fixed intervals or when queries take too long, the state built by the current request can be stored as a snapshot. The immutable, append-only structure of data ensures that snapshots and cache never get outdated.

5.5 Related technology

5.5.1 Functional programming

When building a program based around functions, one is practicing functional programming. A program should be built upon functions that takes in an input and returns a result, that always is the same for the same input parameters (Hughes, 1989). This means that the function cannot depend on any persistent state that mutates over time or produce any side-effects in other parts of the code. Such a function is called a pure function and can be seen as a computer analogy to mathematical functions in that they have a deterministic outcome (Milewski, 2014). The modern use of the term “functional programming” usually include cases where parts, and not necessarily whole applications are built on these principles. Common procedures within Big Data processing such as MapReduce lend heavily from functional programming principles, utilizing the deterministic nature of functions to efficiently process large amounts of data in parallel (Dean & Ghemawat, 2008).

5.5.2 Command Query Responsibility Segregation – CQRS

Separation of concerns is desirable when developing software (Dijkstra, 1976). By separating functionality for different concerns, each part can solve its problem in the most optimal way.

This idea of separation of concerns can be applied to the handling of state and storage. Command-Query Separation (CQS) was developed as part of the work on the Eiffel programming language by Bertrand Meyer (Meyer, 1988). CQS divides all methods into two types. The first has a void return type, called Command. It can mutate state and is not a pure function since it can have side-effects. The second type of function in a CQS system is called a query. A query has a non-void return type and is not allowed to mutate state.

Command Query Responsibility Segregation (CQRS) applies the CQS principles to database reads and writes (Young, 2014). When choosing which database system or setup one should use for a task, a tradeoff that must be considered are whether the system should be optimized for fast writes or fast reads. Quick lookups on changing data often depend on constructing tables and trees for different indices, slowing down ingestion of new data.

By separating the system into write and read models, each can be optimized for their own load. All commands go to one model, and all the queries go to another model (Figure 5). For most systems, queries are what you need to scale. Data are written once and read many times (Young, 2014). For most queries, it is sufficient to be eventually consistent (Brewer, 2000).

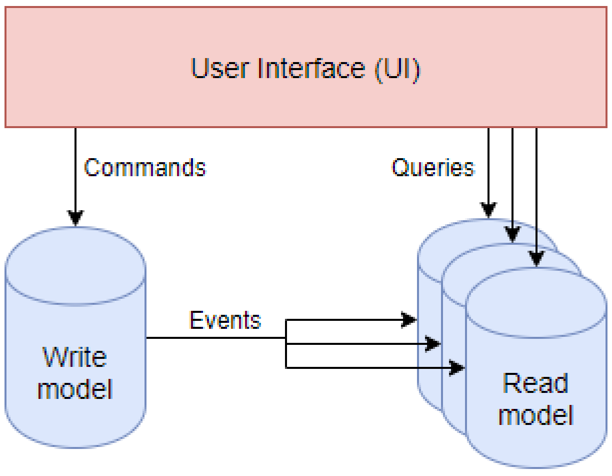


Figure 5 - Separation of writing and reading in a CQRS + Event sourcing system (Adapted from Debski et al. 2018)

If it is accepted that the reads might not be fully up to date, but just eventually consistent, it is possible to separate the reading and writing part of the data storage. Event Sourcing and events is one way to bridge the gap between the models, and the two patterns are often seen together.

According to Greg Young, who coined the term CQRS, it is not possible to implement Event Sourcing without CQRS (Young, 2014).

5.5.3 Domain-Driven Design – DDD

When using the Event Sourcing approach, a common strategy is naming the events in a way that adheres to the Domain-Driven Design (DDD) modelling philosophy. The core principle of DDD is that what is stored reflects the domain one is working in. If the system is tracking the movement of ships between ports, the event of a ship leaving port should not be represented as removing one ship from the list of ships in port or changing the location parameter for the ship in the database. These approaches are prone to error, as they are derivatives of the real event happening. By storing an event named "*ShipDeparted*", and then drawing conclusions based on that data, one layer of abstraction is removed from the model. It is then easier to track what the program is doing and catch any illogical behavior. In practice, the models change more often than the actual underlying behavior, and representing the behavior is therefore desirable (Fowler, 2005; Young, 2014).

To be able to follow the Domain-Driven Design principles, it is imperative that the developer, or the system, has information that allow the representation to reflect the real world.

5.6 Relevance for spatiotemporal datasets

Spatial datasets covering larger areas and comprehensive data can grow quickly. The same is true if one wishes to keep a record over time, being able to query how a distribution or dataset looked like at a given point in the past. Storing a full version of the dataset for every relevant timestamp quickly becomes unfeasible, and deleting older data removes potentially relevant data from the system.

An event-based approach shares similarity with some popular file compression techniques, opting to store differences between data rather than full representations of them. This is in many cases much more compact. Rather than storing the results of a sequence of operations, you store the operations themselves.

Another relevant comparison is to source code version control systems like git (*Git*, 2020; Spinellis, 2012). By storing the changes made to a file, rather than multiple full copies of it, they offer a space efficient way to track changes over time. This makes it possible to go back to an earlier version if necessary, and the lightweight nature makes it easier to store copies remotely. Comparing different updates to the underlying data also makes collaborative work on the same files much simpler and is one of the reasons why source code version control systems are a cornerstone of modern software development (Ruparelia, 2010).

5.7 Cloud computing

The concept, and usage of, cloud computing has taken hold in the last years. Cloud services are based around the principle the responsibility of hosting is offloaded to a "cloud" of servers somewhere in the world, connected to the internet, rather than maintaining your own servers. You deploy your data, code, and other resources, to a remote location and rent computational capacity rather than purchasing your own servers.

Many of the largest software, hardware and service companies in the world have established themselves within the cloud service industry. Major cloud providers include Amazon, Microsoft, Google and IBM (Kratzke, 2018).

There are advantages to cloud computing compared to more classic server solutions. Time spent purchasing, configuring, and maintaining infrastructure is reduced. Instead, focus can be directed towards development of the software running on the infrastructure.

When the software is deployed as part of a large pool with storage and processing power scaled to handle many applications at the same time, peaks in storage and processing demand for a single application can usually be handled by the much larger system. There is less of a risk of a web page belonging to a small company suddenly becoming popular overnight, and the company behind it not being able to handle all the web page requests coming in. In a similar vein, there is no need for over scaling hardware to hedge for future growth, as these actions can be taken when or if the need arises. This is not only a concern for growing web pages, "peak loads" can also occur in applications where processing is happening regularly, but not constantly.

Cloud computing carries a lower risk of downtime compared to a privately hosted server, and in case of such incidents the recovery time is usually faster. This is due to the advantages of scale, where multiple redundant systems and servers can be setup without a large upfront investment. A company whose main business is selling cloud services are also more likely to possess specialists within the field of accessibility and are more likely to have on-site operations staff that monitor and intervene if any error situation occurs.

Modern cloud computing platforms also offer a wide array of instrumentation for setup of services (Spillner, 2017). An emerging field within software development and IT management is DevOps, aiming to automate the processes for building, testing, and releasing software with code and scripting. One example is Infrastructure as Code, where the setup of servers, databases and connections are defined through scripts rather than button presses in menus or a series of commands, so that they can be easily replicated and updated as necessary. The availability of such DevOps instrumentation is one of the key benefits of using established cloud platforms (Spillner, 2017).

5.7.1 From storage to processing – the different layers

One common categorization of cloud services is by layers of abstraction, where different services are classified by how closely coupled they are to actual server hardware.

5.7.2 Infrastructure as a service (IaaS)

The most "low-level" type of cloud computing service with little abstraction offered is Infrastructure as a service, or IaaS. Within the scope of IaaS, you can rent services such as a server, a virtual machine, a virtual network or similar. You are renting specific hardware setups in different structures. IaaS "[…] provides the physical computing resources that are configured by the user to meet variable needs" (Sugumaran & Armstrong, 2017).

The customer has control over and is responsible for updating and maintaining the operating system and any software needed to fulfill any further necessary requirements for the given system. IaaS provides maximum flexibility for consumer-created software but does not try to hide the operation complexity of the application (Kratzke, 2018).

5.7.3 Platform as a service (PaaS)

Platform as a Service, or PaaS, refers to the practice of a service delivering configurable foundational software components such as databases and the middleware that handles flows of information among applications (Sugumaran & Armstrong, 2017). Platforms can provide the necessary "wiring" in a solution, but do not typically solve consumer needs directly. Examples of PaaS might include a managed database service with all required tooling supplied. With an IaaS solution renting server capacity, setting up a virtual machine on the server and then hosting a database on that might have solved the same issue. The PaaS approach offers less flexibility, but also less setup work and tuning of hardware.

5.7.4 Software as a service (SaaS)

Traditionally, the type of cloud service positioned closest to the end user and furthest away from managing hardware is Software as a Service, or SaaS. "[SaaS] is generally manifested as managed, network-enabled applications" (Sugumaran & Armstrong, 2017). This means that SaaS can provide a finished solution that fully covers a need that a consumer might have.

Sugumaran and Armstrong mention services such as Google Apps or GIS software delivered through the internet browser as examples of SaaS. This is a significantly different kind of cloud service than renting servers or virtual machines.

5.7.5 Virtual machines and container architectures

An important feature of cloud processing is to be elastic and flexible (Kratzke, 2018). Over time, IaaS architectures have moved away from providing actual servers, and towards virtual machines. An abstraction on top of the physical hardware, a virtual machine might be run on any server within a larger set of servers. The flexibility of virtual machines gives providers the opportunity of utilizing the underlying hardware better, by dividing larger servers into smaller virtual machines.

While virtual machines provide benefits over physical servers, they still are full replications of systems, and have a large base footprint (Kratzke, 2018). A more recent development have been in container-based architectures like Docker (Felter et al., 2015; Merkel, 2014). By using advanced system functionality and technology, container architectures allow creation of lightweight, virtual machine-like systems with all required functionality for an application and little more. This allows for much smaller footprints for each container, and the adoption of architectures where system load can easily be distributed to many containers through load balancing. Typically, one container might contain one component of the architecture and all its dependencies.

5.8 Serverless computing

A common problem one can encounter when running services in the cloud is the need for determining how much computation power and storage space the service might need. Proper prediction of the required resources is essential for cost management, as these parameters directly controls how much the service costs (Dillon et al., 2010; Eivy, 2017).

Ideally, one could answer the question of how much processing and storage is needed with the answer of “Just enough for the application.” This is what serverless computing aims to deliver, by abstracting allocation of resources away from the customer.

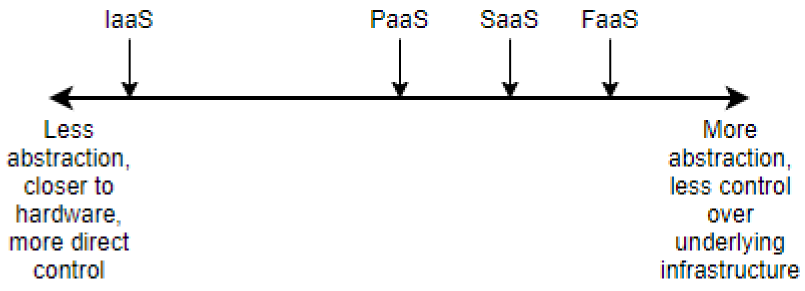

IaaS
PaaS
SaaS
FaaS
Less abstraction, closer to hardware, more direct control
More abstraction, less control over underlying infrastructure

Figure 6 - Simplified differences between different cloud services

A further abstraction built on top of what was discussed earlier, serverless computing aims to let the customer not have to worry about the cloud architecture and infrastructure at all. Backend code is deployed to the cloud as functions and gets run when called, without any long-lived server application needing to be dedicated to it. When the function is called, it runs, and when it is finished delivering its output, it is like it was never there. FaaS systems have the advantage that they can scale to zero when not under any load, which is not normally possible for other cloud setups (McGrath & Brenner, 2017). The programs, or scripts, in a FaaS architecture have properties like those of pure functions (Section 5.5.1). This means that they contain no internal or persistent state, and simply execute according to their input.

Many of the available commercial serverless systems are mostly limited to functions and scripts based on predefined templates and specifications (Enes et al., 2020), but there are also examples of FaaS solutions which provide full-fledged programming language support.

The use of serverless computing has been growing the last few years with advancing technology in the area, and the growth is expected to continue the coming years (Varghese & Buyya, 2018).

5.8.1 Function orchestration

Basic FaaS systems are structured around functions, with no state management involved. This might be a challenge for more complicated workflows, like pipelines requiring multiple steps and parallel processing. The FaaS model still lacks adequate coordination mechanisms between functions in more complex solutions (Baldini, Cheng, et al., 2017; Garcia Lopez et al., 2019).

Introducing an orchestrator function with extended capabilities, with the responsibility of managing data flow and execution order of functions, reduce the need for an external service is a way to handle this requirement.

This idea is quite similar to a technique called sagas, originally developed for handling long lived transactions in databases (Garcia-Molina & Salem, 1987). Originally proposed as a way of splitting up longer transactions into many smaller steps to free up resources in between steps, this principle has since been introduced as a tool used in modern web development for complex state changes. (*Redux-Saga*, n.d.)

FaaS platforms and function orchestration is an emerging field. This means that existing literature on the field is limited, and the industry seems to lead the way when it comes to new developments.

A comparison of different FaaS orchestration solutions shows several differences between the commercial solutions provided. Although direct comparisons of the different solutions is difficult due to differing scopes and models, Garcia Lopez et al. (2019) provides some main takeaways. AWS Lambda from Amazon is the most mature solution, with a clear billing model, low overhead, and some support for parallel execution. A weakness however is its limited scripting language, and the fact that the orchestrator itself is not a function, which limits function composition (Baldini, Cheng, et al., 2017).

IBM Composer from IBM performs close to AWS Lambda for short-running applications and is easier to set up than its competitors. This is in line with its focus of targeting more simple workflows.

Azure Durable Functions (ADF) from Microsoft does not measure up in terms of performance, as the system produces significant overhead compared to the other systems on all loads. However, it is by far the most advanced in terms of programmability, with full-blown support for commonly used programming languages such as C# and JavaScript, and powerful syntactic structures for asynchronous and parallel programming.

While these findings are accurate at the time of writing, Garcia Lopez et al. (2019) stresses that they are likely to change over time, as all examined solutions are in active development.

5.8.2 Cost control in cloud computing

Moving processing operations from local servers to the cloud have been shown to be a resource-effective and therefore cost-effective way to compute (Van Eyk et al., 2018). A sign of this is the wide adoption of cloud and migration of existing solutions to the cloud in the industry. However, predicting the cost of a solution and comparing different cloud products based on this has been difficult (Eivy, 2017). As the industry matures, this is expected to become less of a problem, but currently there still are challenges.

Serverless systems costs are based on use rather than allocated resources, and this might be preferable for some uses.

5.8.3 Security concerns in cloud computing

Cloud computing allow us to offload some of our security concerns to someone else. Many concerns still remain however, data is vulnerable to attacks wherever it is stored and processed (Ryan, 2013). For instance, it is still important to limit who can access the data, even though the cloud provider might streamline the setup of access restrictions. The threat of a malicious insider with access is still relevant in the cloud, and might be larger due to the number of employees or subcontractors tasked with running the cloud service (Hubbard & Sutton, 2010).

Sending large amounts of data across the internet and most likely across national borders to servers maintained by someone else do come with a slew of security challenges. In some instances, it is the main hurdle for widespread adoption of cloud solutions. Strong isolation between different users that host services in shared resources is also vital (Baldini, Castro, et al., 2017). This is because deploying malicious code to a server can bypass security measures if it is allowed to interact with other services hosted on the same hardware.

6 Methods and pipeline implementation

6.1 The task

The pipeline implementation needs to accomplish several tasks. First, it needs a way to look at two different versions of a spatial dataset and pair up features for event generation. Second, it needs a way to create events that describes the changes. As these are computationally intensive tasks for larger datasets, they need to be implemented in a way that can scale well. While the actual implementation of the algorithms used for these tasks is outside the scope of this thesis, parallelization of the tasks (Section 5.1.5) can still be utilized in order to make them scale well. In addition to these tasks, the final event data needs to be made available to other applications through an output mechanism.

6.2 Implementation

Processing of large amounts of spatiotemporal data and making them quickly available would, following a traditional approach to server technology, require a large investment in processing power. Since the arrival of new dataset versions are of a periodic nature, with a lot of time spent idling, maintaining such an infrastructure would be cost-ineffective. By solving this problem through cloud-based, serverless computation easier flexibility and scalability can hopefully be achieved.

6.3 Technology choices

Based on the review of cloud computing solutions and orchestration presented in Section 5.8.1, combined with own experience with the technology, it was decided to implement the processing pipeline using the Microsoft Azure cloud platform. The individual tasks of the pipeline were implemented in C# and deployed as Azure Functions. A Durable Function orchestrator was used to manage data flow from task to task. The orchestrator also took care of handling the parallelization of tasks, using the fan-out, fan-in pattern (Section 6.4.6).

6.4 Architecture overview

Figure 7 describes the pipeline and overall solution that was implemented.

Below follows an overview of the implementation, before a more throughout description of the different components follow.

Before anything was initiated, a PostGIS database setup by co-supervisor Atle Frenvik Sveen was loaded with a dataset prepared by him for the task. PostGIS is a powerful geospatial extension to the popular open-source RDBMS PostgreSQL. 15 separate tables contained yearly versions from 2005 to 2019 of OSM (Section 5.3) data.

The pipeline was initiated by a HTTP Post request to an endpoint generated by Azure, where run parameters concerning which year to fetch, how many entries and timeout values were passed in for testing purposes. The *HttpStart* function initiated by the request launched the *DurableOrchestrator*, the central component for the pipeline.

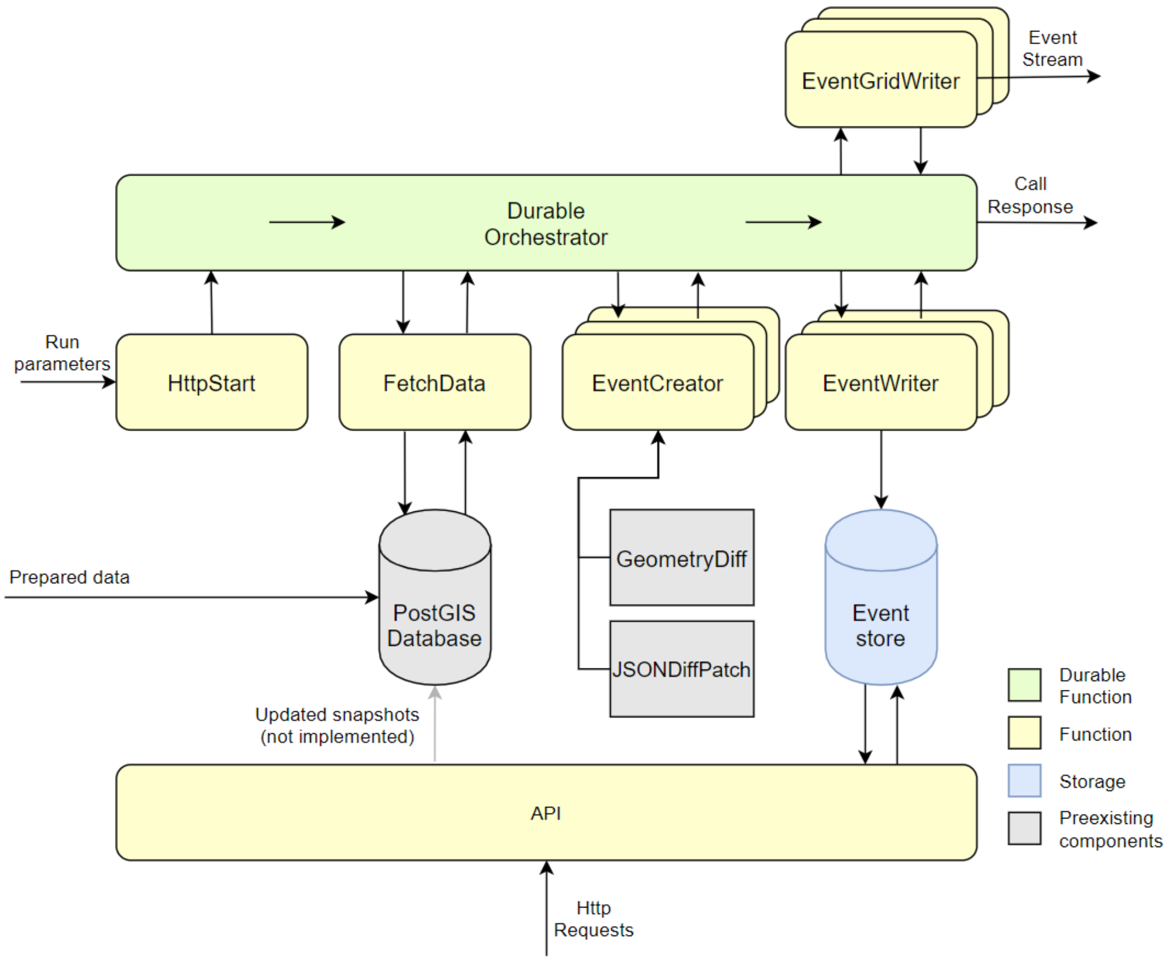


Figure 7 - Overview of the solution architecture presented

The task of the *DurableOrchestrator* function was to manage the application state and initiate other functions as they were required. It used storage tables to track how far the execution had run and went dormant whenever another task was running. It was also the component responsible for parallelizing processes. When the *DurableOrchestrator* was first launched, it returned several end points for keeping track of its status from the outside. The Call Response (on the right-hand side of Figure 7) was the return result that got written to one of these endpoints. This output was used extensively for development and testing purposes.

The first task was to fetch data from the database. This was the responsibility of the *FetchData* function, which used a database connection string from the application settings and parameters from the run parameters fed into the application to fetch the right data. The task of this function was to pair objects that was updates of each other together for easier event generation later.

The *EventCreator* function was initiated with a database entry or a pair of entries, depending on the event type required. It found the differences in geometry and descriptive tags between versions by utilizing external libraries and built events for writing. The *EventCreator* functions was launched in parallel, as each function could run independently of each other.

The last components of the main pipeline were the *EventWriter* and *EventGridWriter* functions. These took the events made in the *EventCreator* and wrote them to different output mediums. The *EventWriter* function wrote them to a persistent event store, while the *EventGridWriter* function wrote a stream of events to an *Event Grid*, the Azure infrastructure for passing events to different consumers.

The Application Programming Interface (API) was responsible for making the event data accessible in a traditional state format, essentially reversing the process in the *DurableOrchestrator* function. It took in an Http Get Request and returned a full version of the dataset in the current form, by running through all the events of the event store and applying relevant updates and deletes to recreate the current state.

6.4.1 The Durable Orchestrator

The *DurableOrchestrator* function was the central building block of the event pipeline. Implemented as an Azure Durable Function (ADF) (Gillum et al., 2019), the responsibilities of the orchestrator was managing the flow of the application. This included initiating other functions and passing them parameters, receiving results back and passing them further along the line. A setup with Azure Functions without an orchestrator would have included intermediary storage or queue structures between each step, and splitting work between multiple parallel processes would be much more difficult. The ADF is itself implemented based on Event Sourcing. Every time it is started, it executes from the top of the program. By storing the state as a series of events, it kept track of which functions it had invoked before and what result it got back. When it encountered a situation where it was awaiting a response from a function call, it skipped through the rest of the execution and turned off. It later woke back up and started retracing the event log through the code, recreating its state, whenever it was finished waiting for other processes. This repeated itself until the whole orchestrator had been executed successfully.

When initiating multiple concurrent and parallel functions, it was necessary to create a task list for the orchestrator to await results from. This way, multiple functions was started from one run of the orchestrator, and it could sleep until all of them was finished executing.

6.4.2 Prepared input data

The test data set used during development and subsequent testing of the solution was a subset of the Open Street Map dataset. The dataset mainly consisted of features located in Norway and had been transformed from State Two Object timelines (page 17) into yearly snapshots for the purpose of this application. Each table consisted of 5 columns: object id, object version, timestamp, descriptive tags, and geometry. Object version was an incrementing value representing the version order in the object timeline, and together with object id was a unique identifier.

6.4.3 Querying efficiently

When considering a cloud-based solution, the Input-Output (IO) operations and network latency would most likely be a large factor when measuring runtime. An early implementation of the pipeline fetched entries one at a time, but this soon became a bottleneck. By performing different database join operations between two versions of a table where ids persist from one version to another, finding update pairs, deletions and creations was quite simple. This was a more efficient query (See appendix) than simply fetching one entry at a time.

An inner join of two versions of the same dataset (Figure 8d), on the id field of both, resulted in all entries which was a part of both sets. These were our updated or unchanged

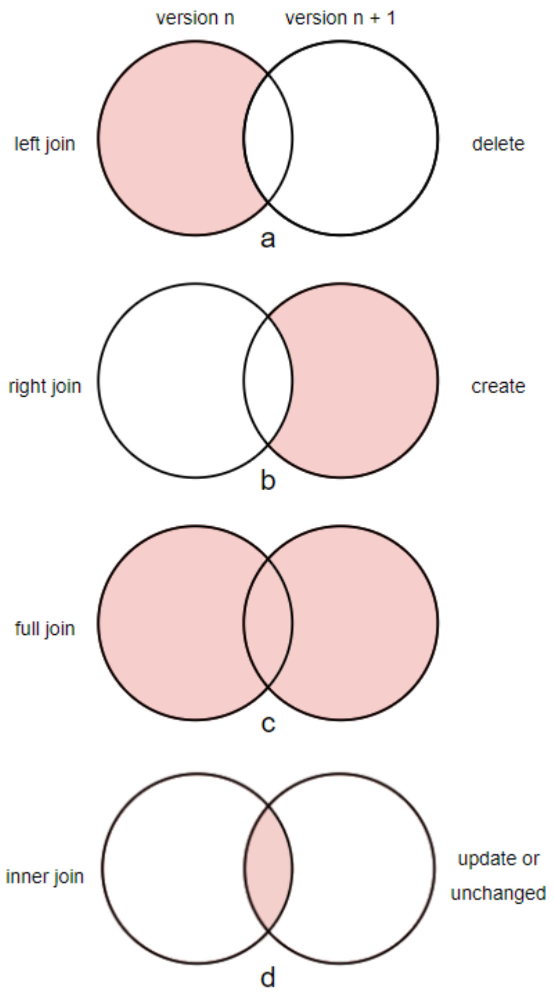


Figure 8 - Joins of datasets and their corresponding diff event type

objects, which the function would need to calculate differences for.

A left join (Figure 8a) combined with looking through the ids that had a null entry for the table of version n +1, gave us all the objects that existed in version n, but not in version n + 1. These were our deleted objects, and deletion events could be created for these objects.

A right join (Figure 8b) combined with looking at the null entries, gave us all the new entries that are in version n + 1, but not in version n. These were our creation events.

The *FetchData* function (Figure 7) took this data and structured them for further processing. It also grouped the data by version for structure in case a run required comparing more than two versions. An example of this would be looking at data for a stretch of years, such as from 2013 to 2016.

6.4.3.1 Making an agnostic framework

The task presented was to make a pipeline capable of handling different datasets, rather than implementing it specifically for the test dataset. The test dataset was distinct in some ways from the intended target, especially in the way it was versioned. It was easy to join between versions on feature id values as these were consistent across versions, this is an assumption that not necessarily holds true for other sets. Although matching entries between versions without the help of a stable identifier was considered outside the scope of this work, the task specified that the pipeline should be able to handle the introduction of such a component. Because of this, an aggregate id for internal indexing was used for an easier transition to non-indexed data. Adding a function before *FetchData* that matched different versions and gave them corresponding aggregate ids should integrate well with the solution.

6.4.4 NetTopologySuite (NTS)

NetTopologySuite (NTS) is an opensource library for .NET, adapted from the Java library JTS Topology Suite. It is an "[…] API for modelling and manipulating 2-dimensional linear geometry." It provides numerous geometric predicates and functions." It is based on the OGC Simple Features definition of objects and was used for help handling the processing of geometries in the solution. (JTS, 2020; NTS, 2007) Whenever there was need for the geospatial features of an object, NTS was used to serialize the WKB data (Section 0) from the database and then represent the geometry.

6.4.5 Creating events

After the relevant, versioned data was loaded into the program and assigned matching aggregate ids, the next step was to generate change events.

6.4.5.1 What should the event contain?

How can a diff event be identified? They are directly connected to the geographic objects they are describing, so the first part of their identifying key was the connecting aggregate id. However, any object could get changed multiple times over its lifespan, so an incremental version number was included to differentiate events. This combination of aggregate id and version number was used as a unique identifier. Updates, deletes and creations of objects might be handled differently depending on the application, so an *EventType* value was included to represent this.

The main payload, however, was the *GeometryPatch* and *TagPatch* values. The *GeometryPatch* was created using the GeometryDiff library, made by co-supervisor Sveen, which compared two incoming geometries. The *Diff* returned represented a translation from version n of the object into version n+1 and made it possible to Event Source the geometries themselves, storing them in Stage Three representation.

The *TagPatch* represented the updates to all tags (also known as attributes, tags is the name used in OSM) to the geometry. The tags in the test data set was represented as JSON objects. Finding the changes to a JSON object was a solved problem, and a mature code library was used to calculate the resulting diff patches for these as well (Bryan, P., Nottingham, 2013).

By applying the *GeometryPatch* and the *TagPatch* to an object of the old version (version n), the return value would be the same object, updated to the new version, version n + 1 (Figure 9).

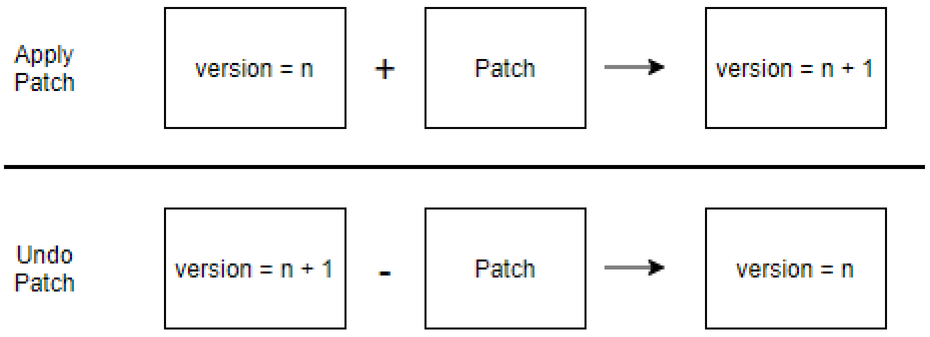


Figure 9 - Patching and unpatching an object

The patches could also be undone (Figure 9), sending the patch and the new version through an undo operation would return the old version.

The *DiffEvent* also contained the timestamp of version n + 1. It was assumed that version n is the correct representation of the data until the update occurred, after which the updated data was the new truth. This assumption holds true for most relevant problems we seek to solve and would be the best representation available with the data the application had access to. By tracking the timestamp of the *DiffEvent*, we could represent the correct state of the object at any time, not just the present.

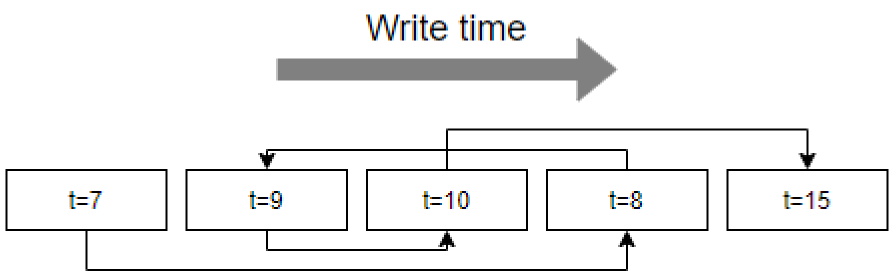


Figure 10 - Timestamping is important for ordering events in an asynchronous setting

The timestamps could also be used to make sure that the API read the events in the right order (Figure 10). In an asynchronous, append-only write scheme, if the data was not explicitly versioned, timestamps can be used for versioning, as the system does not guarantee that the events show in the right order.

Finally, a reference was included to the dataset id and version, to easier keep track of which data set the diff event was related to. As one of the underlying ideas was that multiple datasets could be fed through this pipeline and onto an event queue, it was important to be able to separate the different datasets from each other again easily, in case a listener function was only interested in a subset of the available diff events.

6.4.6 Parallelization

An important design detail is that Event Sourcing enabled aggressive parallelization of tasks. This could be done since no event was dependent on any other. The events are only used together later, when read from the event log. By keeping track of the timestamps and version ids an eventually consistent state could be guaranteed in the event log (Musa Elbushra & Lindström, 2014).

As such, the solution could use the powerful fan-out/fan-in pattern that is presented as one of the main strengths of ADF (Roberts, 2019).

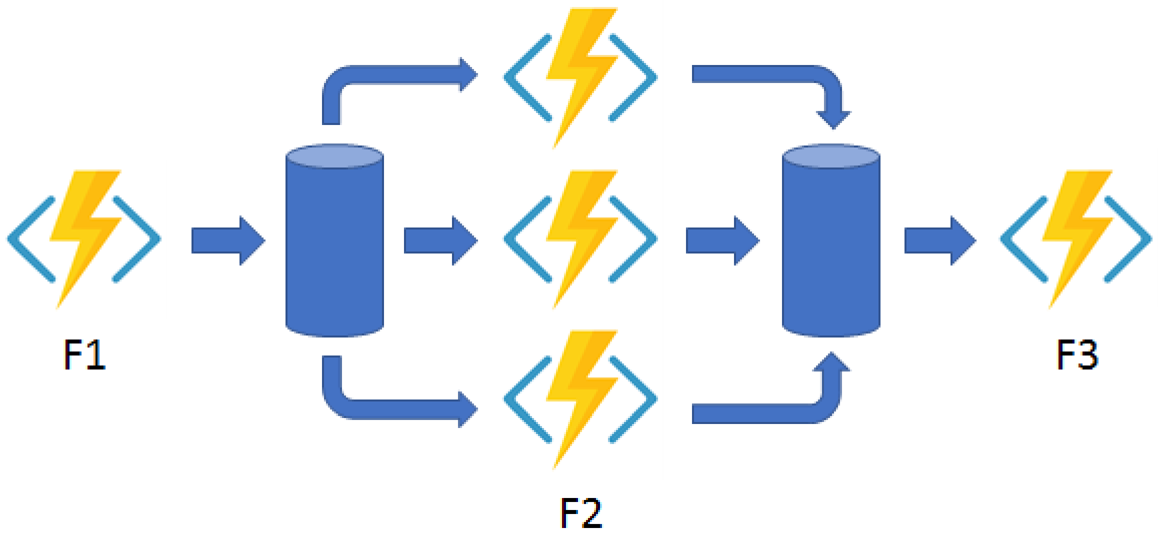


Figure 11 - Flow of the fan-out/fan-in pattern in Azure Durable Functions (Gillum, 2019a)

Normally, while it can be easy to initiate multiple parallel processes, it is much more difficult to keep track of all the awaiting functions and know when to resume the orchestrator. To do the same here, a list of all the tasks initiated was generated, and then the system awaited the completion of all the tasks in the list. As soon as all tasks were done, the orchestrator could continue to the next step, and the system had successfully fanned out and back in. Assuming full parallelization, the runtime of this part was equal to the max runtime among the tasks, rather than the sum of all the run times. Of course, more time was spent on overhead, but for sufficiently large problems this should be a smaller factor than the speedup from parallel processing.

Note that this parallel processing not only occurred across multiple processors, but across multiple virtual machines and servers. Some network latency was therefore expected.

6.4.7 Writing events to persistent storage

After all the events had been created in parallel, they had to be written to persistent storage. Although the intention is that external services can listen to an event queue for any updates, there needed to be some storage keeping track of the history. This was to keep track of what had happened in the past. If a consumer of events later had to be rebooted, it might have had to reread previous events to recreate the current state before it could resume listening to new events.

One obvious option for storage was PostgreSQL, as it was already used for the input data. It has great support for geospatial operations with PostGIS as mentioned in the architecture overview. However, the events themselves did not contain any direct geographic features, only diffs. Second, the event stream generated by the program was more suited for a document store of some kind, as the structure of events might need to change over time. As older events cannot be changed in an Event Sourcing system, the storage would have to accept different structures, and document stores are more flexible in this regard. A RDBMS might also struggle with the large spikes in writes at certain times, as it would need to hold connections to all the parallel writers. Although a document store setup could be achieved in PostgreSQL and Azure Functions (Marten, n.d.), a more lightweight solution with lower overhead costs for cloud hosting was preferred. Azure Table Storage is a simple solution made to work in tandem with different solutions in the Azure cloud environment and integrated seamlessly with the function architecture. Writing to the table store was as simple as redirecting the return value of the function to storage rather than back to the orchestrator (Figure 12).

```
[return: Table("EventLog")]
```

Figure 12 - The code necessary for writing to Table Storage from Azure functions

As mentioned, Durable Functions already use Table Storage for keeping track of its orchestration process, which also is an event-sourced approach.

Azure Table Storage lacks query options outside of the RowKey and Partition key, the combination which uniquely identified each row. However, it follows the serverless principle of no baseline cost and only paying for the used functionality. Its lightweight nature also allows it to store potentially huge volumes of data, which is ideal for the large amount of data that an event store might contain (Heath, 2017).

6.4.7.1 Choice of keys

The choice of keys for the event store was important for fast read performance later. Keys should be chosen according to the read patterns most likely to occur. For the persistent storage, it was assumed that the most common reads would be getting all events related to either a whole dataset or a single aggregate id. The partition key that decided what information gets stored together was set to the dataset id. Inspiration was also drawn from the setup Netflix used for their licensing server (Figure 13) (Avery and Reta 2017). A common choice for row id in event stores can be the aggregation, or view, that is interested in the data. According to Avery and Reta, the data structure should come from the aggregation. For us, the aggregates correspond most closely to the dataset ids. A limit in Table Storage is that there can only be one additional key to the partition key. To ensure unique identifiers, a combined key of aggregate id and version (Section 6.4.5.1) was used.

.

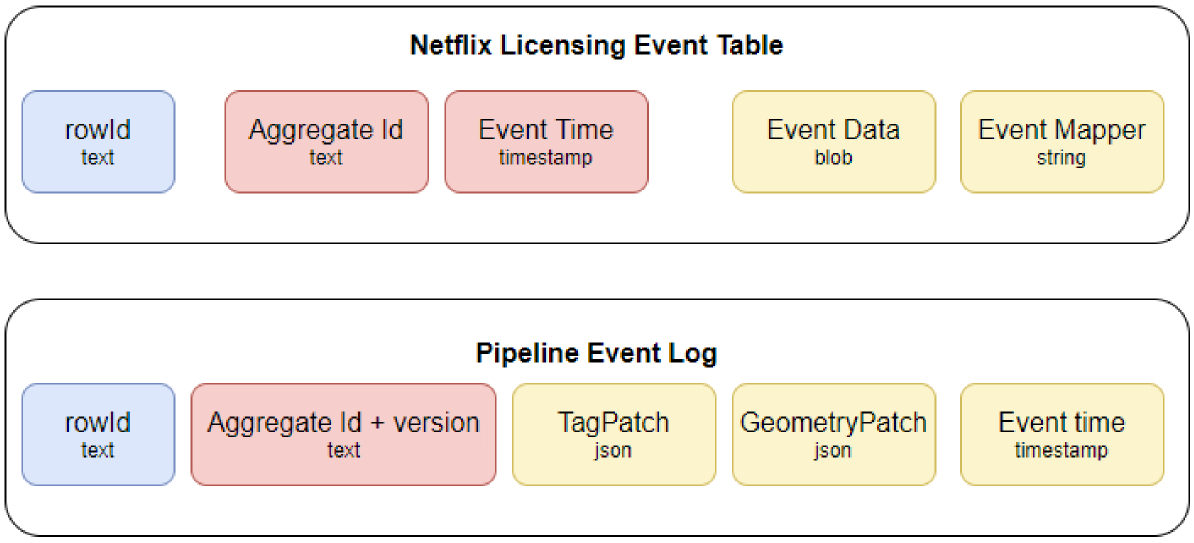


Figure 13 - Example of setup of event table, with partition key, row keys and data columns (Netflix setup based on Avery and Reta 2017)

6.5 Cloud setup

To make the above solution work, some resources had to be set up in the Azure cloud. Deploying to Azure is grouped into resources and resource groups.

Resources are any single service that are available in the cloud solution. Examples range from databases, to virtual machines, to function apps. Depending on the resource, these can be categorized under Platforms-as-a-Service (PaaS), Software-as-a-Service (SaaS) or serverless computing/Functions-as-a-Service (FaaS).

For our setup, the serverless computing services were used for all processing purposes. This was due to the need for elastic scaling, compared to the more rigid structures offered through virtual machines. The initial setup was also simpler, as there was less need for defining endpoints, processing power and storage space ahead of implementation.

The Event Pipeline and the API was deployed as two separate Function Apps, setup with HTTP triggers. This meant that the services would be ready to run and would do so any time someone with the right credentials sent a GET request to the relevant HTTP endpoint.

A resource group represents a container that groups related resources together. Under the hood, the resource group holds the metadata for the grouping, and provides a way to control and monitor the connected resources together. All resources belong to one and only one resource group. A common strategy for grouping is based on resource lifecycles. If the resources are created, updated, and deleted together and their structure is dependent on one another, they should be grouped together. Here, the resources for the Event Sourcing pipeline, the event storage and the API were grouped in a resource group, while the database containing the test dataset was hosted in another. This was decided because the test dataset existed separately of this project and was also accessed from other work. A structure like this encourages good development practices related to decoupling code and the separation of concerns described by Dijkstra (1976).

6.5.1 Creating an API

A vital part of making the system useable was creating an API that allowed external solutions to make reads on the event log without having to adapt to its specific structure. A core principle should be that the underlying data structure should have as little impact as possible.

For this reason, we wanted to return objects and not events when we got read requests from the solution. The core principle, and what was implemented, was that every object was a sum of all the diff events related to that object. The object got created, then changed and then changed again. At one point the tags updated as well. What the end user usually would be interested in would be the current object state at some point in time.

6.5.1.1 Separating objects

To process the events for each object, it was decided to group the events by object aggregate ids. While this required an extra pass over the events, it was decided that the simplicity of this solution and the ability to reduce latency by reducing the number of read requests was worth it. Hence, the first step was fetching all events related to the dataset in question.

Language Integrated Query (most known as LINQ) is a query tool built into the C# language that offers functional programming patterns. It provides numerous capabilities for sorts, groups, searches and similar for collections such as lists. For this purpose, its *GroupBy* function made it simple to go through the events for one object at a time, building the dataset.

Another way to achieve a similar data structure that was considered, was looping over the list of events. Each event could have then been mapped to a Dictionary structure with aggregate ids as keys and lists of events as values. In the end, the resulting structure would be similar.

6.6 Test setup

After the system was implemented and set up properly in Azure, it was necessary to test how performant it was. As the main performance criteria was scalability and throughput, it was decided to run the system on increasing workloads in terms of tuples handled, until max throughput was found. When maximum capacity was found, a run time analysis for different loads would determine how the system scaled. Ideally, a solution with parallel execution of tasks would scale sub-linearly as input got larger.

6.6.1 Runtime analysis

Implementing a timer function within the orchestration function did not work due to the constant restarts of the orchestrator, and the recommended method for recording the execution time of a solution was to inspect the execution history within the Microsoft Azure Storage Explorer interface (Gillum et al., 2019). When measuring the time elapsed, there was a number of points that could be defined as the starting and stopping points (Figure 14). The most relevant for total performance would be measuring the time from the user sends a request and until the user has access to the requested data. However, this period includes network latency and the time Azure uses to initiate the process. As this thesis is focused on the pipeline itself, it was decided to measure time elapsed from the startup of the Orchestrator function (t_0), until the orchestrator was finished with all its subprocesses (t_1).

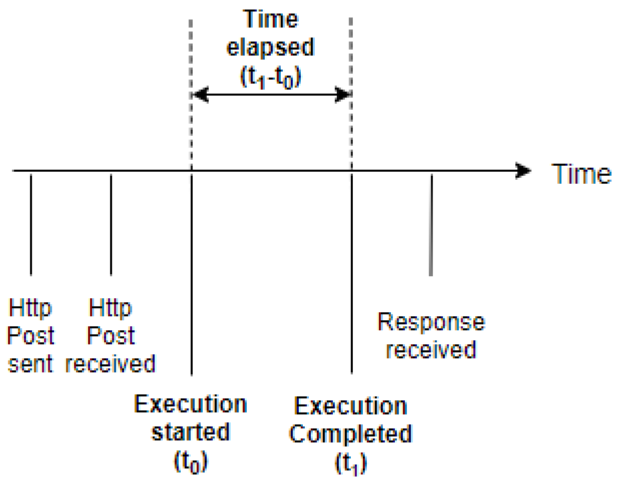

Time elapsed (t_1-t_0)
Time
Http Post sent
Http Post received
Execution started (t_0)
Execution Completed (t_1)
Response received

Figure 14 - Selecting a time span to measure

The table *DurableFunctionsHubHistory* in Azure is the event store for the orchestrator function and is used to keep track of different functions starting and stopping. By using the timestamp of the events *ExecutionStarted* and *ExecutionCompleted* we could find the time elapsed during the execution of the pipeline.

6.6.2 Number of runs and different datasets

We define one run as an execution from start until end. While timing the function from t_0 to t_1 reduces some unknown variables, it was vital to find the average time over multiple runs to reduce uncertainty.

A variable that could affect runtime was the data being used, as geometries vary in size and analysis complexity. It was decided to do half the runs with an update from 2012 to 2013 and the other half from 2013 to 2014. This could give some insight into whether different data impacted runtime in a meaningful way.

6.6.3 Parallel runs

As discussed in Section 5.8.1 about Function Orchestration, Azure Durable Functions are orchestrators that themselves are functions. A main benefit of this is the ability to create sub orchestrations that can handle parts of a larger workflow (Baldini, Cheng, et al., 2017). To test the possibility of including such a strategy for handling datasets too large to handle in one run, we ran a simple test by scheduling multiple runs concurrently on the largest test set to see whether a parallel execution of multiple pipelines could scale even further.

6.6.4 The uncertainty of serverless

As discussed in Section 5.8, a core principle of serverless architectures is the delegation of hardware and middleware responsibilities to the cloud provider. When testing the performance of the pipeline, it can be difficult to control unknown variables related to how Microsoft deploys and runs the ordered services. As shown by Lynn et al. (2017) and Garcia Lopez et al. (2019) serverless orchestrators in general and Azure in particular is still not mature in terms of reliable run times.

6.6.5 Function timeouts

In some cases, a function stalls and is not able to finish its execution within five minutes. If this happens, Azure terminates the function. As the pipeline implemented did not contain any fallback solution for this, runs that took more than five minutes was discarded from the runtime results.

7 Results

7.1 Runtime

The raw event data exported from Azure were further grouped and analyzed using Microsoft Excel. By calculating the difference in timestamps between the *ExecutionStarted* and *ExecutionCompleted* events, time elapsed (*tElapsed*) for each run is found. The final results are presented in Table 2, grouped by dataset (*ySerie*) and number of objects (*nObjects*).

Dataset years	ySerie	nObjects	tElapsed	Runs	Avg tElapsed
2012-2013	1	50	00:51,5	5	00:10,3
2012-2013	1	100	01:23,1	5	00:16,6
2012-2013	1	150	01:48,5	5	00:21,7
2012-2013	1	200	02:20,4	5	00:28,1
2013-2014	2	50	01:08,1	5	00:13,6
2013-2014	2	100	01:07,2	5	00:13,4
2013-2014	2	150	01:59,6	5	00:23,9
2013-2014	2	200	02:00,7	5	00:24,1
2013-2014 *(In parallel)*	3	200	04:43,2	5	00:56,6

Table 2 - Runtime results by size of dataset and dataset used

A sample set of raw data exported from Azure is included in the Appendix (Table 3).

Figure 15 shows the results as a bar chart. The first column in each grouping describe average runtime when creating events from 2012 to 2013, the second 2013 to 2014 and the third the combined average of all runs. When the number of objects increase, we can see that the average runtime increases from about 12 seconds on 50 objects to about 26 seconds for 200 objects.

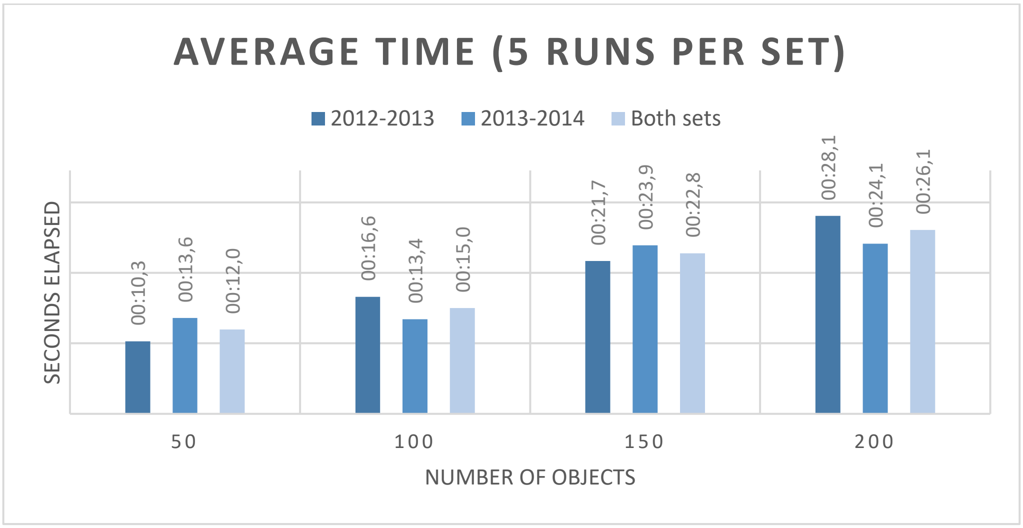


Figure 15 – Runtime results by size of dataset and dataset used

7.1.1 Reruns due to timeout

Due to timeouts (Section 6.6.5), four runs that all took place right after each other that all lasted over 5 minutes were discarded from the results and the relevant runs was run again.

7.2 Parallel runs

We included runs in parallel (Table 2, last row), to test whether multiple pipelines running concurrently could be a way to increase throughput.

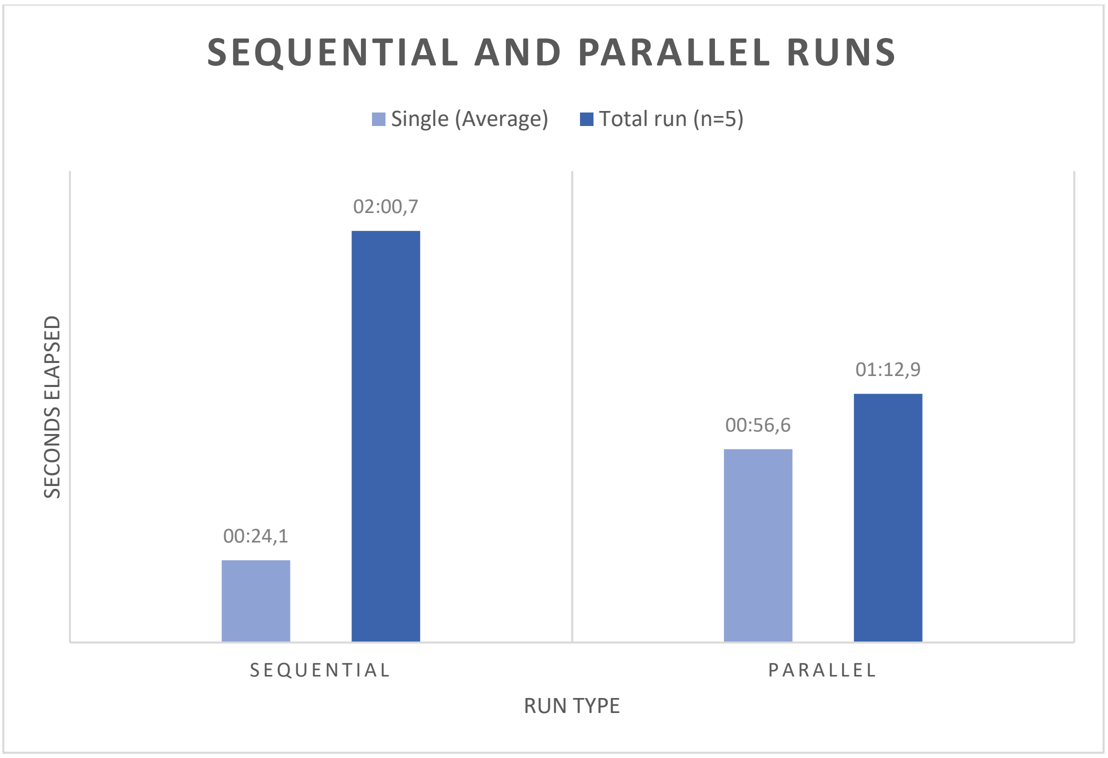


Figure 16 – Comparison between sequential runs and parallel runs. Each average runtime is the mean over 5 runs. The total run is the summation of single runs for the sequential run, and the time from first execution start to last execution completion for the parallel run.

Each run is executing significantly slower when run in parallel, almost 57 seconds compared to 24 seconds. However, the total run completes faster in parallel. 1:12,9 minute is over 45 seconds faster than the sequential run.

There was some delay between the start of runs while executing the parallel setup. This was partially due to waiting for a response to each call before sending the next one, and partially due to the delay in Azure between receiving a request and starting execution (Figure 17). The sequential runs are assumed to start up as soon as the previous run is done. Parallel runs start up with a delay from the one before. All runs vary in duration.

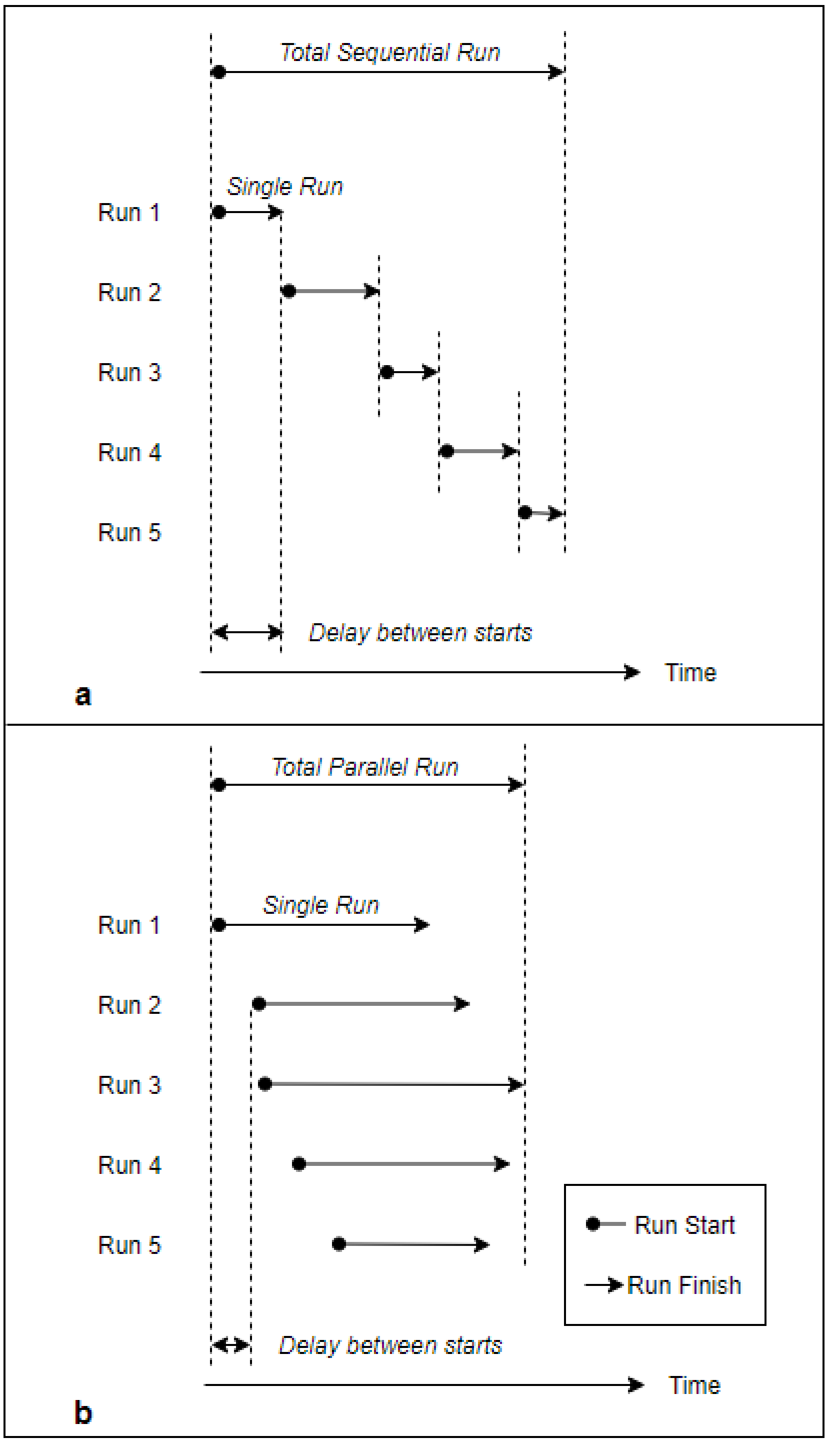


Figure 17 - Illustration of timing sequential (a) and parallel (b) runs.

7.2.1 Maximal runtime for parallel operations

The worst-case scenario of total run time in a parallel setup would be if the last run started, were the one taking the longest time. Then the total execution time would be the time of the longest run, plus the sum of all start delays.

$$t_{max,parallel} = \max(t_{run\,1}, t_{run\,2}, \dots, t_{run\,n}) + \sum_{i=1}^{n-1} t_{delay,n}\ .$$

In the test run with parallel runs, the sum of delays was 12,6 seconds, while the largest t_{run} was 1:01 minutes.

$$t_{max,parallel} = 61.0\ seconds + 12.6\ seconds = 73{,}6\ seconds$$

This was only slightly larger than the t_{total} observed, which was 72,9 seconds.

7.3 Costs of processing in the cloud

Figure 18 describes the costs associated with hosting the Azure Functions, Table Storage and Durable Orchestrator throughout the project.

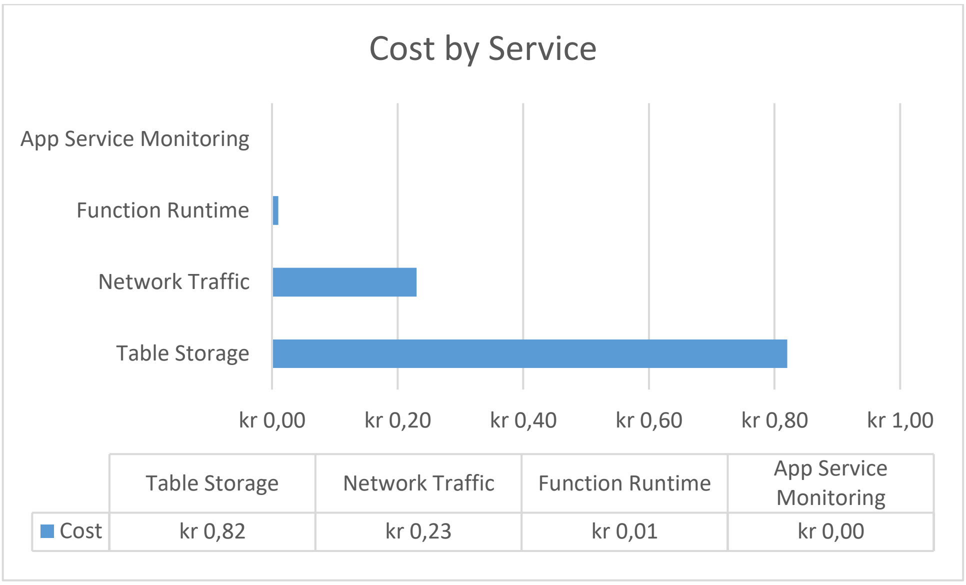


	Table Storage	Network Traffic	Function Runtime	App Service Monitoring
■ Cost	kr 0,82	kr 0,23	kr 0,01	kr 0,00

Figure 18 - Cost breakdown over project duration. All services represent the sum of traffic throughout development and testing

The costs associated with prototyping and testing the application can be considered negligible, with a total cost of 1.06 Norwegian Kroner (kr). Over 77 % of this, 0.82 kr, was from the Table Storage. Costs were fetched directly from the cost breakdown available in the Azure admin panel.

In line with the serverless architecture of the solution, the billing model for all services used was usage-based, with no underlying, monthly cost.

8 Discussion

Let us start by looking back to the main research questions presented in the introduction.

1. Is a cloud-based implementation of a diff-based event generation pipeline a viable solution for producing an event-stream from traditional, bulk-updated data?
2. How can such a pipeline be implemented in a modern, cloud-based, computing environment?
3. How does such a pipeline perform in terms of scalability and throughput?
4. How does an event generation pipeline for versioned geospatial data fit into a larger software architecture in terms of integration of data consumers?

8.1 The cloud-based pipeline implementation

The serverless paradigm of cloud computing seems especially well suited for the task of turning bulk-updated geospatial data into an event-stream. The recent advances in cloud-based computing enables us to scale up and down processing capacity as needed, and billing structures allow the system to scale to zero when no processing is needed. Handling events that represent a change to a single feature within a large dataset enables aggressive parallelization of the processing of these.

The challenges with cost found by others (Eivy, 2017) was not encountered in this early-phase implementation. While not representative of production-level loads, the low accumulated costs after some amount of load put on the system throughout development and testing is promising.

8.2 Scalability and throughput

The main challenge to overcome for this cloud-based system to be a viable implementation for the event generation pipeline is its scalability. Can it handle the large amounts of event generation needed for updating a large geospatial dataset from one version to another?

While the solution worked well for processing low- to medium numbers of features, larger processes (500-1000 features) stalled and did not run successfully. Three possible culprits could be:

- Orchestrator out of memory

When the orchestrator was run locally on a cloud emulator for test purposes, running out of memory was the main challenge. The local emulator runs all processes on the same computer and does not fully represent how resource allocation works in the cloud but having a problem on an emulator might mean it eventually becomes a problem in the cloud application as well.

- Run history takes longer and longer to process

As more functions are called, tasks awaited and others completed, the history table used by the orchestrator function to find its current state grows. This is the same challenge that any Event Sourcing system encounters at some point without snapshotting (Discussed in Section 9.3). As this initially was a likely candidate, the system was tested with an option enabled called Extended Sessions (Gillum, 2019b) that mitigates this issue by not turning off the orchestrator as aggressively when many small functions are called. As this changed little in terms of system performance, it is less likely that this is the issue.

- Large geometries block the pipeline

Tracing the execution logs, the bottleneck appears to be when fanning out *EventCreator* functions. Here, the difference (diff) generation for the update events are run. The diffing algorithm used for finding the differences between geometries was *GeomDiff*, made by Sveen. He commented that "preliminary results indicate that the *GeomDiff* algorithm degrades significantly with large (1000+ vertices) geometries."

Although this represents a relatively small portion of the geometries, their compute time can be many orders of magnitude longer than the rest. In a pipeline architecture with parallelization, this problem is initially not significant, but as the total number of these geometries get closer to the max parallel capacity, we see a significant slowdown (Figure 19).

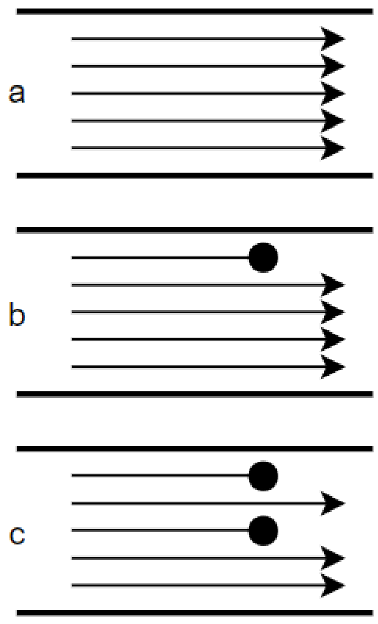


Figure 19 - Slow processing of some events cause throughput issues

A makeshift solution for timing out long-lived processes and falling back to representing the *Update* with a *Delete* and *Create* did help, but not enough to overcome the problem. A potential direction is to predict what geometries might be difficult to create patches for and use the fallback solution directly. Finding a way of creating an update event that contains the delete-create update representation would be better than creating two separate events, as this does not accurately model what is happening.

The results presented in this master thesis are not enough to conclude whether the orchestrator running out of memory, slow processing of some events or other factors is the hindrance for further scaling. If the parallel test used different data sets for each of its runs, we could have gathered more information about the issue. This way, the probability of it encountering problematic objects would increase and its longest run would likely slow.

8.2.1 Parallelization not as aggressive as hoped

The blocking of the pipeline by slow processes would not be a problem if the "pipe" itself was wide enough. One of the core assumptions made when selecting an architecture for parallel processing of the events was that this could be scaled further than what was the result. While the solution is closely following the official documentation for implementing the fan-out, fan-in pattern, some coding error might have limited the number of parallel processes. However, when searching for a solution online, others seem to have encountered similar scalability issues for processing heavy workloads. When encountering this kind of load with varying processing requirements, some of the underlying virtual machines might run on peak load while others are idling, and the load balancer might not realize that it needs to spin up more resources. Official documentation suggests that up to 20 VMs might be used at the same time (Gillum, 2019b), but even for hour-long runs we seldom registered more than 10 running at the same time.

One of the common suggestions for a work-around is limiting the number of processes each VM is delegated (Gillum, 2019b). However, taking into consideration numbers of VMs and how much each should process seems counter intuitive to the serverless workflow that the setup suggests it can deliver. Further work is required to find the root cause of this.

Another bottleneck that might be to blame is available memory in the orchestration function. While this is not something that we were able to decipher from the logs or live monitoring of the runs, when running similar tests on the local cloud emulator provided by Azure this was the main reason for slowdown. This might not be the case when running in the cloud, but it could be worth investigating.

The solution is running fine for smaller workloads. A reasonable approach is to split up an update of a database into multiple smaller runs. One way this could be achieved is by wrapping a new orchestration function around the existing one, delegating what part each should do. As there is no double-billing when running multiple levels of orchestration, the extra cost would be minimal, and the fact that Durable Functions is functions themselves allow for this approach Function orchestrationSection 5.8.1). As orchestration-within-orchestration is not easily done in all cloud platforms, a different approach would need to be used if implementing this extra orchestration step elsewhere (Baldini, Cheng, et al., 2017; Garcia Lopez et al., 2019). The results of running parallel runs compared to sequential ones indicate that there is an opportunity for better overall capacity by following this route. However, this was not tested in-depth and more work is needed to find an optimal structure for this.

8.2.2 Synthetic and real-life datasets

Ideally, any solution should be tested in a variety of scenarios, on a variety of different datasets (Theodoridis et al., 1999). However, for testing out the validity of a concept, it is normally not feasible to create a synthetic dataset that can cover such a variety.

As the system was tested on an adapted, real-life dataset it already has some resiliency to variation of data. During development, it was discovered that some features changed from one OGC Simple Features type to another (e.g. *LineString* to *Polygon*), something the *GeometryDiffer* library could not create a patch for. It is challenging to anticipate challenges such as this when creating a synthetic dataset that should model the challenges of real-world data.

8.2.3 Reliability

Four runs during testing had to be discarded due to function timeouts (Section 7.1.1). As these four happened right after each other and none of the other runs with identical parameters were affected similarly, there might be something happening server side that has not been accounted for. During the period these tests were done, the relevant Azure server region that was used (Western Europe) was experiencing up to a 775% increase in traffic due to the Corona virus epidemic in Europe (Microsoft, 2020). This might have caused some instabilities in the service, although this is not certain.

8.3 Similar work

Zhou et al. (2004) proposed a solution not too different from the one outlined in this thesis. An "event-based incremental updating (E-BIU)" system handles updates for spatiotemporal databases.

A main difference is that the paper looks more into the consumer-side of events, using them to update a database, rather than focusing on the production of events. "It is assumed that the event information can be collected and transferred to the database system according to certain form designed previously." This master thesis has focused on the generation of events that such a system can use for updates. Another key difference is that the implementation by Zhou et al. was done before cloud computing became widely available, and hence is implemented with a more traditional data structure and database setup in mind.

An interesting point is made about a type of change event, reappearance. Something exists, and then disappears before once again appearing. The example given is a river drying up during a drought, then reappearing after the drought is over.

A similar pattern was encountered in our own test data, but in a way that not necessarily represents real-world change. OSM is a crowd-sourced dataset, and we found examples of datasets that were registered, removed, and then reappeared at its original position. Because of this, the data is removed, then later reapplied. We can see the same in our specific transformation of the set, where the data was treated as fully updated versions every year. Every set not updated in a year was treated as deleted. Some models for events include reappearance (Worboys, 2005), while other models omit them (Claramunt & Thériault, 1995). The reappearance event might be more relevant for geospatial events than for other uses and should be considered for future work in the field.

Although serverless architectures are new and still in active development, it has already been used within the scientific community in general and for geocomputation, due to its ability to scale rapidly and parallelize larger tasks.

An example is satellite imagery analysis pipeline to evaluate surface reflectance of lakes (Figure 20) (John et al., 2019).

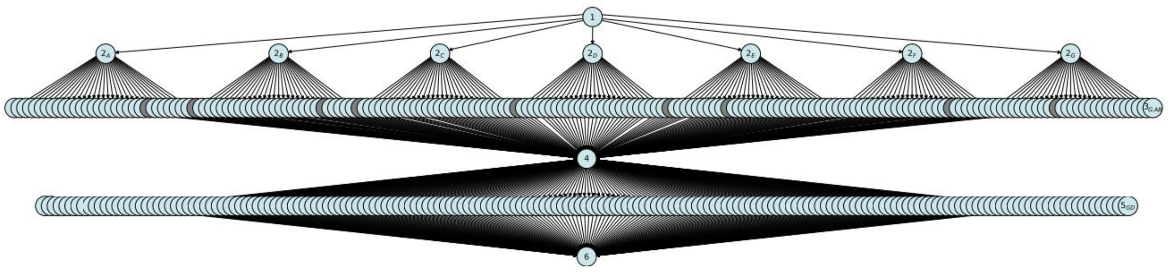


Figure 20 - Pipeline flow of highly parallelized processing of satellite imagery (John et al., 2019)

By subdividing raster images and processing different parts of the images in parallel, large throughput is found. Note that in Figure 20, in the second row from the top, we can see an example of sub orchestration for dividing up the work.

Recent work on Event Sourcing is more and more focusing on implementations in the cloud. Not only is cloud computing in general increasing in popularity, the functional, event-driven nature of serverless architectures fit well with Event Sourcing. Betts et al. (2012) represent the intentions of Microsoft of supporting this direction and implementing the Durable Orchestrator as an event-sourced system further signify this.

8.4 Other main takeaways

8.4.1 Debugging Durable Functions

A core challenge today when developing a solution with a Durable Functions orchestrator, is debugging code.

When the program encounters a runtime error, the program stops its execution, which is expected. However, as the orchestrator function is durable and keeps track of its execution in persistent storage, rerunning the code after updating it picks up where the previous run was and possibly tries to execute the same, faulty code from earlier instead of the new code. To remove impact from one run into the next the code either must run until execution is finished, or the user must manually clear out a *WorkItems* queue as well as storage tables keeping track of running instances. This is design as intended by the Durable Functions team, but a cumbersome process. Although good development practices and support tools might catch many bugs before runtime, some will invariably show up.

9 Conclusion and future work

There have been done work earlier in the overlap between geocomputation and Event Sourcing. Interestingly, it seems like the idea of events for updating state is more established within work on geospatial data than elsewhere. A lot of work done in the 90's and 00's on temporal GIS solutions are relevant for Event Sourcing. There is a lot of overlap between work on Event Sourcing and the serverless paradigm within cloud computing, and as shown in Section 8.3 there have been promising results in moving geocomputation to the cloud.

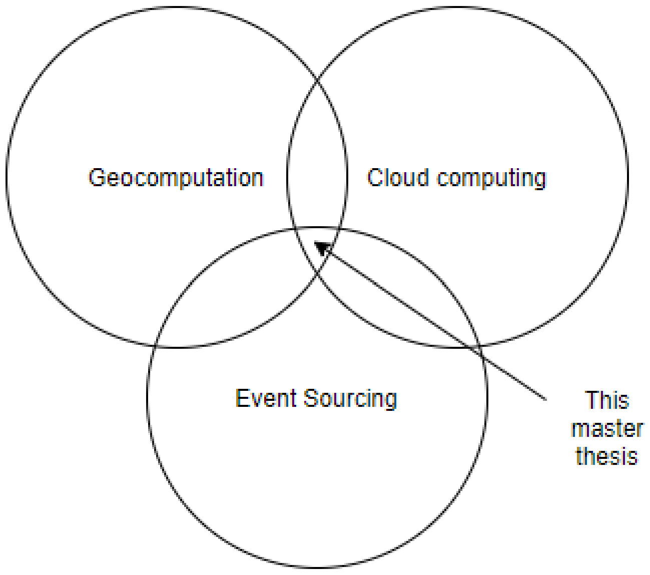


Figure 21 - The scope of this thesis

This thesis has experimented in the crossing point between these three technologies and discovered some possibilities. Further work should reveal interesting results, as the combination has many synergies and similarities.

9.1 Schema changes

How should future updates to the event structure and schema be handled (Overeem et al., 2017)? A core principle of Event Sourcing is that data should not be overwritten, it should be appended as new events. If we update events that happened in the past, the data is no longer immutable. A system for notifying consumers that a previous event has changed needs to be implemented, and these consumers need logic to deal with this. Depending on the use-case it might not be possible. The promise of no invalid caches is broken, so now the system needs a way to handle this. The benefits related to security and reproducibility is no longer there. This means that we cannot overwrite earlier events when converting to a new schema, the history is still what it always was (Young, 2017).

A proposed solution by Young, and in use at Netflix (Avery & Reta, 2017), is versioning. Although a large topic, the core principle is that the events have some version tracker or other metadata on themselves, so the system knows how to handle this type of event. When reading an older version that lacks a value, a placeholder might be added.

9.2 A configurable pipeline

To make future adjustments to the pipeline and to use it for different use cases, it was necessary for it to be adaptive to change and configurable. The pipeline was made basing the modelling of geospatial features on open standards such as OGC Simple Features and building the pipeline with modular functions with clear responsibilities and defined data objects for input and output between them. It should be possible to configure the pipeline for use of different function implementations and for different use cases. Making changes to the overall cloud architecture and the orchestrator function might be more challenging.

9.3 Snapshots

For the current implementation, there is little wrong in storing event-like structures as object lifelines (Stage Two). However, a more mature solution might include snapshotting, to avoid having to play through all events connected to an aggregate id. Figure 22a illustrates replaying events without snapshots and Figure 22b illustrates that fewer steps are necessary if we use snapshots. Since we are storing differences that can be applied in reverse by undoing patches, we can traverse the events backwards from a snapshot if this is easier, illustrated by Figure 22c.

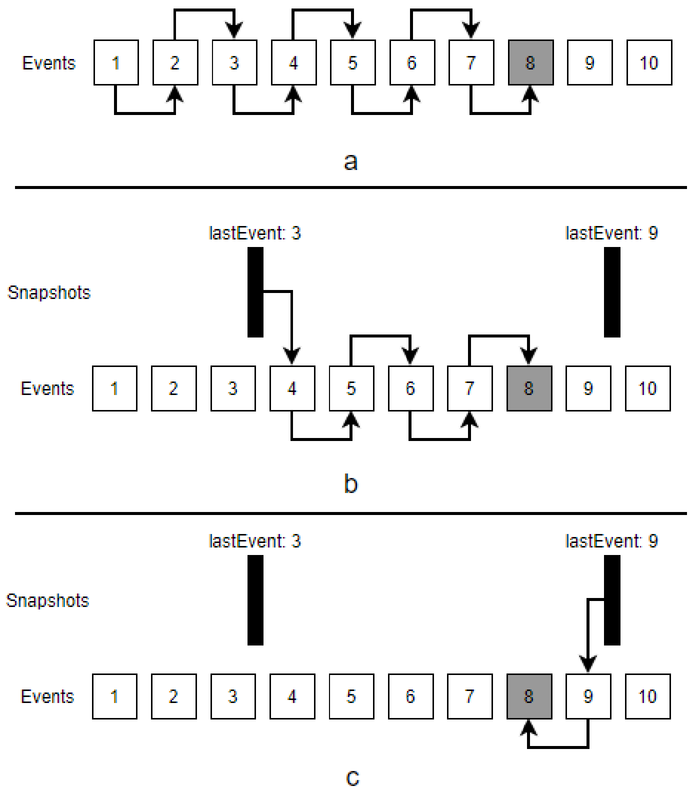


Figure 22 - Reproducing an earlier state: (a) without snapshots, (b) with snapshots, (c) with snapshots and difference-based events

9.4 Including geospatial references in events

One could reasonably imagine that a consumer of the event stream is only interested in a subset of the events within an aggregate. One way of handling this could be including a geospatial reference of some kind to easily filter based on location. Bounding boxes for geometries could be fetched from incoming data or could be generated as part of the event generation. The highly parallelized nature of the processing means that this should not slow down the generation much, although some slow-down of ingestion is to be expected. Including metadata of relevant areas or creating new aggregates is another way of handling this but might not be as flexible for future use cases.

9.5 A variety of input and output

An event-sourced, cloud-based system should lend itself well to integration with Internet-of-Things devices and sensor streams. This is because we already have tried representing more static objects in a way that is much more like the usual data streams coming from these sources. Possible future work in mixing real event data and event-sourced static data could be interesting. An architecture such as the one outlined in this thesis is a good starting point for such a system. We already have implemented two outputs, or read models, in the form of a persistent storage of events with an API that delivers traditional data structures and an event stream (Figure 23). A future version might include more than one input, by integrating more write models. The current is based on differences between versions in databases; an input event-stream should be possible.

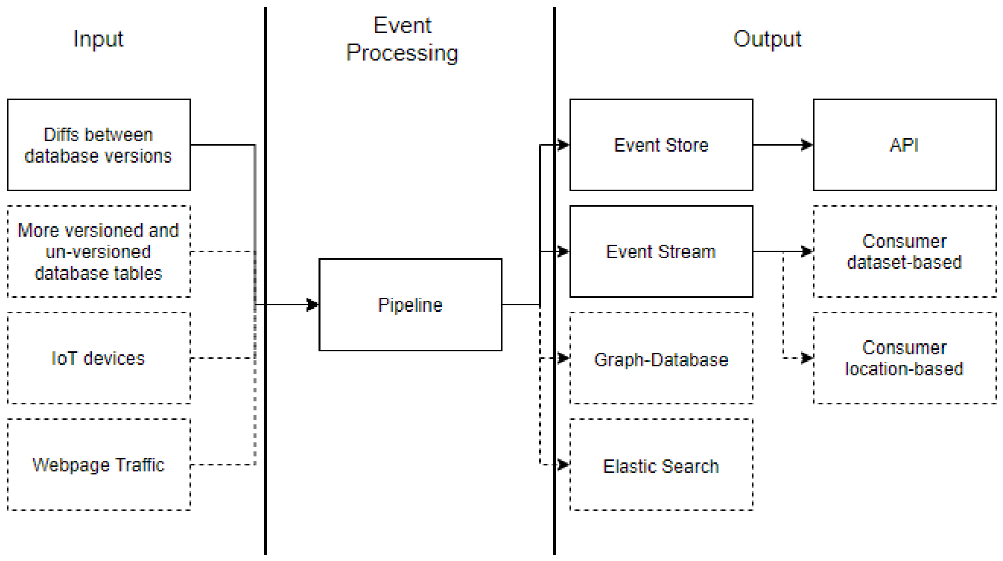


Figure 23 - Possible future integrations on write and read-side of pipeline

9.6 Introducing Domain-driven design

In an ideal world, the data model could be moved closer to a domain-driven based design. What are the creates, updates, and deletes representing? Is it a new road being paved, a river changing its course or simply new, more correct measurements? This is difficult to achieve without changes to the reporting of data. However, the flexibility and usability of the system would increase if such context could be included, either through richer input or more advanced event creation.

9.7 In conclusion

This master thesis has shown the potential of Event Sourcing updates from one version to another of traditional, bulk-updated data. Implementing the pipeline in a cloud environment as a serverless application with an orchestrator responsible for handling data flow shows promising initial results, but further work is needed to scale the setup sufficiently to larger datasets. A possible direction is introducing the idea of a master orchestrator delegating work to sub orchestrations.

By implementing an API that can turn events back into traditional snapshots, the system is able to serve traditional consumers of geospatial data on their terms. However, the event stream written to an event grid allows for new ways of consuming geospatial data in the form of dynamic indexing, location- and dataset-based consumers that update when necessary, and more. Event Sourcing static data also creates opportunities in mixing them with Internet-of-Things (IoT) event data and other event-based systems.

10 Bibliography

Armstrong, M. P., & Marciano, R. (1995). Massively parallel processing of spatial statistics. *International Journal of Geographical Information Systems, 9*(2), 169–189. https://doi.org/10.1080/02693799508902032

Avery, P., & Reta, R. (2017). *Scaling Event Sourcing for Netflix Downloads*. https://www.infoq.com/presentations/netflix-scale-event-sourcing/ Presentation Aug 09, 2017. (Accessed March 16, 2020).

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., & Suter, P. (2017). Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Singapore, Springer. https://doi.org/10.1007/978-981-10-5026-8_1

Baldini, I., Cheng, P., Fink, S. J., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P., & Tardieu, O. (2017). The serverless trilemma: Function composition for serverless computing. *Onward! 2017 - Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Co-Located with SPLASH 2017*, pp. 89–103. https://doi.org/10.1145/3133850.3133855

Betts, D., Domínguez, J., Melnik, G., Simonazzi, F., & Subramanian, M. (2012). *Exploring CQRS and Event Sourcing - A journey into high scalability, availability, and maintainability with Windows Azure*.

Brewer, E. A. (2000, July). Towards Robust Distributed Systems. In: *PODC (Vol.7)*, 1–12. http://awoc.wolski.fi/dlib/big-data/Brewer_podc_keynote_2000.pdf

Bryan, P., Nottingham, M. (2013). JavaScript Object Notation (JSON) Patch. In *RFC 6902 (Proposed Standard)*.

Claramunt, C., & Thériault, M. (1995). *Managing Time in GIS: An Event-Oriented Approach*. https://doi.org/10.1007/978-1-4471-3033-8_2

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM, 51*(1), 107-113. https://dl.acm.org/doi/abs/10.1145/1327452.13274

Debski, A., Szczepanik, B., Malawski, M., Spahr, S., & Muthig, D. (2018). A scalable, reactive architecture for cloud applications. *IEEE Software*, *35*(2), 62–71. https://doi.org/10.1109/MS.2017.265095722

Dijkstra, E.W. (1976). A Discipline of Programming. In *A Discipline of Programming (Vol. 1). Englewood Cliffs: prentice-hall.*

Dillon, T., Wu, C., & Chang, E. (2010). Cloud computing: Issues and challenges. *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*. https://doi.org/10.1109/AINA.2010.187

Eivy, A. (2017). Be Wary of the Economics of "Serverless" Cloud Computing. *IEEE Cloud Computing*. https://doi.org/10.1109/MCC.2017.32

Elmasri, R., & Navathe, S. B. (2016). Fundamentals of Database Systems Seventh Edition. Hoboken, NJ, Pearson

Enes, J., Expósito, R. R., & Touriño, J. (2020). Real-time resource scaling platform for Big Data workloads on serverless environments. *Future Generation Computer Systems, 105*, 361–379. https://doi.org/10.1016/j.future.2019.11.037

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015, March). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)* (pp. 171-172). IEEE. https://ieeexplore.ieee.org/abstract/document/7095802

Fowler, M. (2005). *Event Sourcing*. 18. https://martinfowler.com/eaaDev/EventSourcing.html. Accessed February 15, 2020.

Gaede, V. (1995). Optimal redundancy in spatial database systems. In *Egenhofer M.J., Herring J.R. (eds) Advances in Spatial Databases. SSD 1995. Lecture Notes in Computer Science, vol 951*. Berlin, Heidelberg, Springer

Gahegan, M. (2017). Geocomputation. In *International Encyclopedia of Geography: People, the Earth, Environment and Technology* (pp. 1–5). John Wiley & Sons, Ltd. https://doi.org/10.1002/9781118786352.wbieg0625

Garcia-Molina, H., & Salem, K. (1987). Sagas. *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD International conference on Management of data. December 1987 Pages 249–259 https://doi.org/10.1145/38713.38742*

Garcia Lopez, P., Sanchez-Artigas, M., Paris, G., Barcelona Pons, D., Ruiz Ollobarren, A., & Arroyo Pinto, D. (2019). Comparison of FaaS orchestration systems. *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, 109–114. https://doi.org/10.1109/UCC-Companion.2018.00049

Gillum, C., Chu, A., & Gailey, G. (2019). *Durable Orchestrations - Azure Functions*. Microsoft Docs. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-orchestrations?tabs=csharp#orchestrator-code-constraints. Accessed March 12, 2020.

Gillum, C. et al. (2019a). *Durable Functions Overview - Azure | Microsoft Docs*. Microsoft Docs. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp. Accessed March 11, 2020.

Gillum, C. et al. (2019b). *Performance and scale in Durable Functions - Azure*. Microsoft Docs. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-perf-and-scale#extended-sessions. Accessed March 12, 2020.

Git. (2020). https://git-scm.com/ Accessed March 25, 2020.

Goodchild, M. F., Yuan, M., & Cova, T. J. (2007). Towards a general theory of geographic representation in GIS. *International Journal of Geographical Information Science, 21*(3), 239–260. https://doi.org/10.1080/13658810600965271

Haklay, M., & Weber, P. (2008). OpenStreetMap: User-Generated Street Maps. Pervasive Computing. *IEEE Pervasive Computing, 7*(4), 12–18. https://doi.org/10.1109/MPRV.2008.80

Heath, M. (2017). *Azure Tables–What are They Good For?* SoundCode Blog. https://markheath.net/post/azure-tables-what-are-they-good-for

Hohpe, G. (2006). *Programming Without a Call Stack-Event-driven Architectures*. https://www.enterpriseintegrationpatterns.com/docs/EDA.pdf, Accessed March 16, 2020.

Hubbard, D., & Sutton, M. (2010). *Top Threats to Cloud Computing V1.0.* http://www.cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf

Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal*, *32*(2), 98–107.

John, A., Ausmees, K., Muenzen, K., Kuhn, C., & Tan, A. (2019). Sweep: Accelerating scientific research through scalable serverless workflows. *UCC 2019 Companion - Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 43–50. https://doi.org/10.1145/3368235.3368839

JTS. (2020). *JTS Topology Suite*. https://sourceforge.net/projects/jts-topo-suite/

Kratzke, N. (2018). A brief history of cloud application architectures. *Applied Sciences (Switzerland)*, *8*(8), 1–26. https://doi.org/10.3390/app8081368

Lynn, T., Rosati, P., Lejeune, A., & Emeakaroha, V. (2017). A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*. https://doi.org/10.1109/CloudCom.2017.15

Marten. (n.d.). *Polyglot Persistence for .NET Systems using the Rock Solid PostgreSQL Database*. Accessed March 26, 2020, from https://martendb.io/

McGrath, G., & Brenner, P. R. (2017). Serverless Computing: Design, Implementation, and Performance. *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, 405–410. https://doi.org/10.1109/ICDCSW.2017.36

Merkel, D. (2014). Docker : Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, *2014*(239), 2–7. https://dl.acm.org/toc/linux/2014/2014/239

Meyer, B. (1988). Eiffel: A language and environment for software engineering. *Journal of Systems and Software 1988;8: pp.199-246*. https://doi.org/10.1016/0164-1212(88)90022-2

Michelson, B. M., & Seybold, P. (2011). *Event-Driven Architecture Overview* (Vol. 2). http://elementallinks.com/el-reports/EventDrivenArchitectureOverview_ElementalLinks_Feb2011.pdf. Accessed March 26, 2020

Microsoft. (2020). *Update #2 on Microsoft cloud services continuity*. Azure Blog and Updates. https://azure.microsoft.com/en-us/blog/update-2-on-microsoft-cloud-services-continuity/ Accessed March 29, 2020

Milewski, B. (2014, December 25). 3. *Pure Functions, Laziness, I/O, and Monads - Basics of Haskell.* School of Haskell. https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io. Accessed March 26, 2020.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., & Schwarz, P. (1992). ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems (TODS), 17*(1), 94–162. https://doi.org/10.1145/128765.128770

Mourelatos, A. P. D. (1978). *Events, processes, and states*. *Linguist Philos 2,* pp. 415–434. https://doi.org/10.1007/BF00149015

Musa Elbushra, M. A., & Lindström, J. B. (2014). Eventual Consistent Databases: State of the Art. In *Open Journal of Databases (OJDB)* (Vol. 1, Issue 1). https://www.ronpub.com/OJDB-v1i1n03_Elbushra.pdf

NTS. (2007). *GitHub - NetTopologySuite*. https://github.com/nettopologysuite/nettopologysuite Accessed March 5, 2020

OGC. (2010). OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture. OGC 06-103r4. Version: 1.2.1/2011 *Open Geospatial Consortium, Inc*, 93. http://portal.opengeospatial.org/files/?artifact_id=25355E+Implementation+Standard+for+Geographic+information+-+Simple+feature+access#1

OGC. (2020). *Open Geospatial Consortium*. https://www.ogc.org/ Accessed February 20, 2020

Overeem, M., Spoor, M., & Jansen, S. (2017). The dark side of event sourcing: Managing data conversion. *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 193–204. https://doi.org/10.1109/SANER.2017.7884621

Peuquet, D. J. (2002). Representations of Space and Time. In *The Guildford Press*. New York, NY, Guildford Press. ISBN 9781572307735

Peuquet, D. J. (2017). Representation: Time. In *International Encyclopedia of Geography: People, the Earth, Environment and Technology* (pp. 1–18). John Wiley & Sons, Ltd. https://doi.org/10.1002/9781118786352.wbieg1067

Pfeiffer, D. U., Robinson, T. P., Stevenson, M., Stevens, K. B., Rogers, D. J., & Clements, A. C. A. (2008). Spatial Analysis in Epidemiology. In *Spatial Analysis in Epidemiology*. https://doi.org/10.1093/acprof:oso/9780198509882.001.0001

Redux-Saga. (n.d.). https://redux-saga.js.org/ Accessed March 16, 2020

Roberts, J. (2019). *Don't Code Tired | Understanding Azure Durable Functions - Part 8: The Fan Out/Fan In Pattern*. Don't Code Tired Blog. https://dontcodetired.com/blog/post/Understanding-Azure-Durable-Functions-Part-8-The-Fan-OutFan-In-Pattern Retrieved March 16, 2020

Ruparelia, N. B. (2010). The history of version control. *ACM SIGSOFT Software Engineering Notes*. 35 (1), pp.5-9. https://doi.org/10.1145/1668862.1668876

Ryan, M. D. (2013). Cloud computing security: The scientific challenge, and a survey of solutions. *Journal of Systems and Software*, *86*(9), 2263–2268. https://doi.org/10.1016/j.jss.2012.12.025

Savage, N. (2018). Going Serverless. *Communications of the ACM*, *61*(2), 15–16. https://doi.org/10.1145/3171583

Schneider, M. (2017). Spatial Database. In *International Encyclopedia of Geography: People, the Earth, Environment and Technology* (pp. 1–13). John Wiley & Sons, Ltd. https://doi.org/10.1002/9781118786352.wbieg0670

Shekhar, S., & Cugler, D. C. (2017). Parallel computing. In *International Encyclopedia of Geography*. John Wiley & Sons, Ltd. https://doi.org/10.1201/9781351074612-3

Spillner, J. (2017). Practical tooling for serverless computing. *UCC 2017 - Proceedings of The10th International Conference on Utility and Cloud Computing*, 185–186. https://doi.org/10.1145/3147213.3149452

Spinellis, D. (2012). Git. *IEEE Software*, *29*(3), 100–101. https://dl.acm.org/doi/10.1109/MS.2012.61

Stolze, K. (2003). SQL/MM Spatial: The Standard to Manage Spatial Data in Relational Database Systems. *Proceedings of the 10th Conference on Database Systems for Business, Technology and Web (BTW)*. pp. 247–264.

Sugumaran, R., & Armstrong, M. P. (2017). Cloud Computing. In *International Encyclopedia of Geography: People, the Earth, Environment and Technology* (pp. 1–4). John Wiley & Sons, Ltd. https://doi.org/10.1002/9781118786352.wbieg1017

Theodoridis, Y., Silva, J. R. O., & Nascimento, M. A. (1999). On the generation of spatiotemporal datasets. *In: Güting R.H., Papadias D., Lochovsky F. (eds) Advances in Spatial Databases. SSD 1999. Lecture Notes in Computer Science, vol 1651, pp 147-164.Berlin, Heidelberg, Springer • Online ISBN 978-3-540-48482-0* https://doi.org/10.1007/3-540-48482-5_11

Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uta, A., & Iosup, A. (2018). Serverless is More: From PaaS to Present Cloud Computing. *IEEE Internet Computing*, *22*(5), 8–17. https://doi.org/10.1109/MIC.2018.053681358

van Westen, C. J., Castellanos, E., & Kuriakose, S. L. (2008). Spatial data for landslide susceptibility, hazard, and vulnerability assessment: An overview. *Engineering Geology*. https://doi.org/10.1016/j.enggeo.2008.03.010

Varghese, B., & Buyya, R. (2018). Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*. Vol 79, Part 3, Pages 849-861 https://doi.org/10.1016/j.future.2017.09.020

Wang, Y. (2017). Deck.gl: Large-scale Web-based Visual Analytics Made Easy. *The IEEE Workshop on Visualization in Practice*. http://arxiv.org/abs/1910.08865

Worboys, M. F. (2005). Event-oriented approaches to geographic phenomena. *International Journal of Geographical Information Science, 19*(1), 1–28. https://doi.org/10.1080/13658810412331280167

Worboys, M. F., & Duckham, M. (2004). GIS: A Computing Perspective Second Edition. *CRC Press*. Taylor and Francis Group, Florida

Young, G. (2014). CQRS and Event Sourcing. At the Conference *Code on the Beach 2014, Florida. Saturday, August, 2014* https://www.youtube.com/watch?v=JHGkaShoyNs

Young, G. (2017). Why versioning. In *Versioning in an Event Sourced System*. https://leanpub.com/esversioning/read#leanpub-auto-why-version. Accessed March 19, 2020

Zhou, X. G., Chen, J., Jiang, J., Zhu, J. J., & Li, Z. L. (2004). Event-based incremental updating of spatio-temporal database. *Journal of Central South University of Technology (English Edition)*, *11*(2), 192–198. https://doi.org/10.1007/s11771-004-0040-3

11 Appendix

11.1 SQL Queries

11.1.1 Get updates from table1 to table2

SELECT table1.id, table1.version, table1.tags,

```
ST_AsBinary(table1.geom), extract(epoch from table1.ts),
table2.version, table2.tags,
ST_AsBinary(table2.geom), extract(epoch from table2.ts)
FROM {table1} AS table1
INNER JOIN {table2} AS table2
ON table1.id = table2.id
WHERE table1.geom IS NOT NULL and table2.geom IS NOT NULL
LIMIT {parameters.Tuples}
```

11.1.2 Get creates from table1 to table2

SELECT table1.id, table1.version, table1.tags,

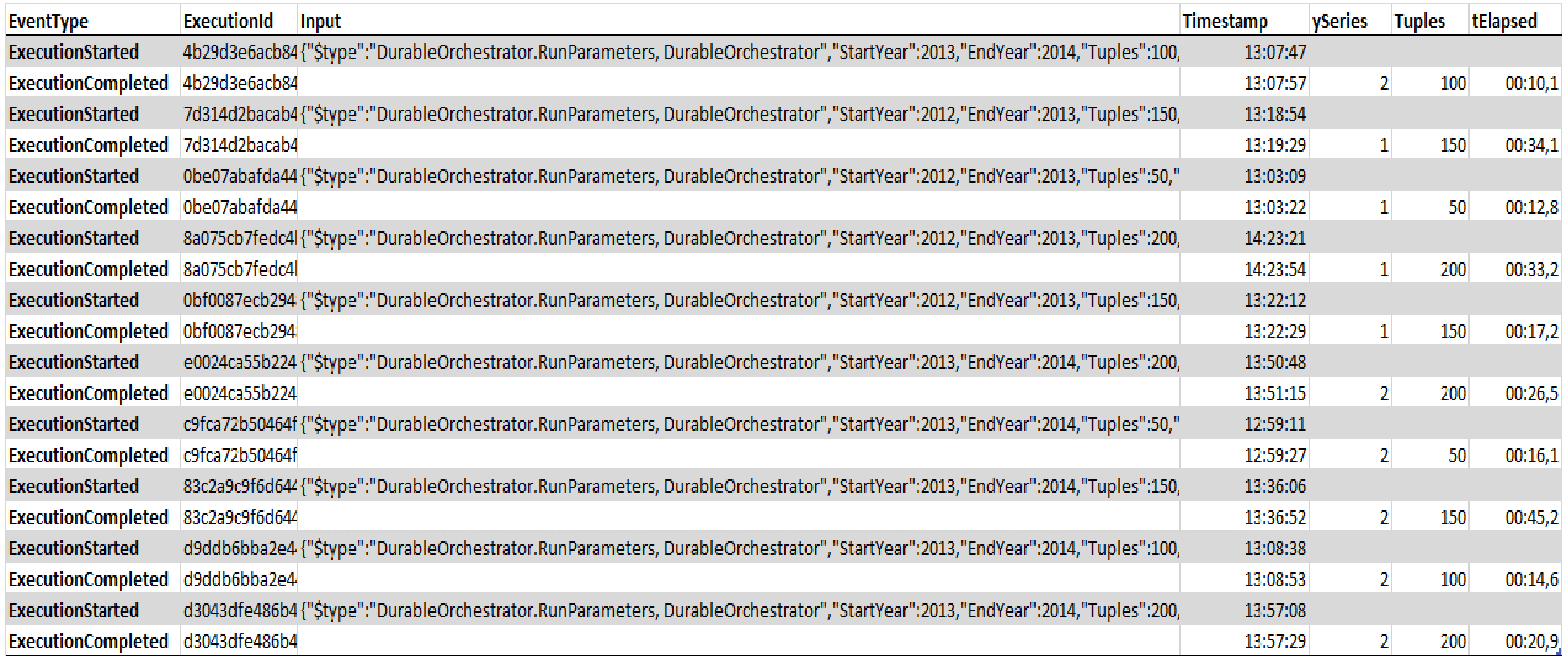
```
ST_AsBinary(table1.geom), extract(epoch from table1.ts),
table2.version, table2.tags,
ST_AsBinary(table2.geom), extract(epoch from table2.ts)
FROM {table1} AS table1
RIGHT JOIN {table2} AS table2
ON table1.id = table2.id
WHERE table1.geom IS NOT NULL and table2.geom IS NOT NULL
ST_GeometryType(table1.geom) = ST_GeometryType(table2.geom)
LIMIT {parameters.Tuples}
```

11.1.3 Get deletes from table1 to table2

SELECT table1.id, table1.version, table1.tags,

```
ST_AsBinary(table1.geom), extract(epoch from table1.ts),
table2.version, table2.tags,
ST_AsBinary(table2.geom), extract(epoch from table2.ts)
FROM {table1} AS table1
LEFT JOIN {table2} AS table2
ON table1.id = table2.id
WHERE table1.geom IS NOT NULL and table2.geom IS NOT NULL
AND ST_GeometryType(table1.geom) = ST_GeometryType(table2.geom)
LIMIT {parameters.Tuples}
```

11.2 Sample Raw Data exported from *DurableFunctionsHubHistory*



EventType	ExecutionId	Input	Timestamp	ySeries	Tuples	tElapsed
ExecutionStarted	4b29d3e6acb84	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2013,"EndYear":2014,"Tuples":100,	13:07:47			
ExecutionCompleted	4b29d3e6acb84		13:07:57	2	100	00:10,1
ExecutionStarted	7d314d2bacab4	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2012,"EndYear":2013,"Tuples":150,	13:18:54			
ExecutionCompleted	7d314d2bacab4		13:19:29	1	150	00:34,1
ExecutionStarted	0be07abafda44	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2012,"EndYear":2013,"Tuples":50,"	13:03:09			
ExecutionCompleted	0be07abafda44		13:03:22	1	50	00:12,8
ExecutionStarted	8a075cb7fedc4l	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2012,"EndYear":2013,"Tuples":200,	14:23:21			
ExecutionCompleted	8a075cb7fedc4l		14:23:54	1	200	00:33,2
ExecutionStarted	0bf0087ecb294	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2012,"EndYear":2013,"Tuples":150,	13:22:12			
ExecutionCompleted	0bf0087ecb294		13:22:29	1	150	00:17,2
ExecutionStarted	e0024ca55b224	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2013,"EndYear":2014,"Tuples":200,	13:50:48			
ExecutionCompleted	e0024ca55b224		13:51:15	2	200	00:26,5
ExecutionStarted	c9fca72b50464f	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2013,"EndYear":2014,"Tuples":50,"	12:59:11			
ExecutionCompleted	c9fca72b50464f		12:59:27	2	50	00:16,1
ExecutionStarted	83c2a9c9f6d644	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2013,"EndYear":2014,"Tuples":150,	13:36:06			
ExecutionCompleted	83c2a9c9f6d644		13:36:52	2	150	00:45,2
ExecutionStarted	d9ddb6bba2e4	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2013,"EndYear":2014,"Tuples":100,	13:08:38			
ExecutionCompleted	d9ddb6bba2e4		13:08:53	2	100	00:14,6
ExecutionStarted	d3043dfe486b4	{"$type":"DurableOrchestrator.RunParameters, DurableOrchestrator","StartYear":2013,"EndYear":2014,"Tuples":200,	13:57:08			
ExecutionCompleted	d3043dfe486b4		13:57:29	2	200	00:20,9

Table 3 - Sample Raw Data exported from DurableFunctionsHubHistory