Managing Provenance in Orchestrated Distributed Systems

Studiengang Medieninformatik

Bachelorarbeit

vorgelegt von

Dennis Appel
geb. in Lich

durchgefuehrt am
Fraunhofer IGD, Darmstadt

Referent der Arbeit:	Prof. Dr. Frank Kammer
Korreferent der Arbeit:	M.Sc. Ivo Senner
Betreuer am Fraunhofer IGD:	M.Sc. Ivo Senner

Friedberg, 2020

Abstract

English

Today, in the world of Big Data, Cloud Computing and Internet of Things, many datasets are produced by increasingly complex data processing pipelines and workflows, which are often executed by workflow management systems. However, the data produced by such pipelines and workflows may not always be correct. Incorrect data is not always discovered immediately, and the problems that caused the incorrect data can be hard to find. Provenance is a powerful tool, which users can use to assess the quality of the data produced by their workflows. The thesis presents a provenance management solution to be used in an orchestrated distributed workflow management system. The provenance solution aims to provide complete provenance coverage with minimal requirements to support as many applications as possible.

To determine the applications requirements, the target system and related provenance solutions are explored and evaluated. Furthermore, a prototype based on the requirements is implemented. Finally, simulated workflow based on a real world use case is used to point out the strengths and weaknesses of the presented provenance solution.

Deutsch

In der heutigen Welt von Big Data, Cloud Computing und Internet of Things werden viele Datenätze durch immer komplexer werdende Prozessierungs-Pipelines und Workflows erzeugt. Diese Pipelines und Workflows werden häufig innerhalb von Workflow-Management Systemen verwaltet und ausgeführt. Die produzierten Daten sind jedoch nicht immer fehlerfrei. Fehlerhafte Datenätze werden meist nicht sofort entdeckt, und die Probleme, die zu den Fehlerhaften Daten geführt haben, können schwer zu finden sein. Provenance ist ein mächtiges Werkzeug, welches Nutzer verwenden können um die Qualität der erzeugten Daten zu untersuchen. Diese Arbeit stellt eine Provenance Lösung für ein orchestriertes, verteiltes Workflow Management-System vor. Die vorgestellte Provenance Lösung zielt auf eine vollständige Abdeckung der Provenienz mit minimalen Anforderungen, um möglichst viele Anwendungen zu unterstützen.

Um die Anforderungen an die Anwendung zu ermitteln, werden verwandte Provenance Lösungen, sowie das Zielsystem untersucht und bewertet. Zudem wird ein Pro-

totyp basierend auf den ermittelten Anforderungen implementiert. Zuletzt werden die Stärken und Schwächen der vorgestellten Lösung anhand von einem simulierten Workflow, welcher auf einem realen Anwendungsfall basiert, ermittelt und aufgezeigt.

Eidesstattliche Erklärung

Ich erkläre, dass ich die eingereichte Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Friedberg, den 17. Januar 2020

Dennis Appel

Acknowledgements

The topic researched in this thesis is part of the “Marauder” project, a project by Fraunhofer IGD, in which a feature-rich orchestrated distributed workflow management system is developed.

I would like to use this opportunity to thank Prof. Dr. Frank Kammer for his support during my bachelor studies and while writing this thesis.

Furthermore, I want to thank Ivo Senner for introducing me to the Marauder project and advising me during this research.

Finally, special thanks go to my family and friends for always being there for me when I need them and for supporting me in every step I’ve taken so far. This would not have been possible without your support.

Contents

List of Figures

Chapter 1

Introduction

When working with data, one question is often left unanswered: "How was the dataset produced?". Before a dataset is made available, the data has to be processed. Depending on the type of data, these processes can vary between simple sensory output reads and highly complex post-processing pipelines. Information about these steps, the data and everyone involved is called *provenance*.

The *Word Wide Web Consortium* (W3C) specifies provenance as "information about entities, activities, and people involved in producing a piece of data or things, which can be used to form assessments about its quality, reliability or trustworthiness" [14]. Moreover, the W3C also provides *PROV-DM*, a generic provenance data model. The data model describes the different components, including the three core structures activities, agents and entities, as well as the relations between them.

Especially geospatial datasets are usually produced by processing pipelines, or workflows. The workflows are often managed by workflow management systems like Apache's *Nifi*[1]. However, such systems are often limiting users to predefined processes, or processes that are implemented using their specifcation. As a result, provenance generated by workflow management systems is often limited to single workflow executions. Hence, information on what happened to a dataset across potentially many different workflows needs to be collected manually.

The "Marauder" project seeks to develop an orchestrated, distributed workflow management system that allows users to connect generic processing, or even storage, components to its infrastructure. The developed system is from here on referenced as orchestration system. This is achieved by on the one hand providing a system-level connector component and on the other hand enforcing a comparatively open specification that allows developers to easily connect their components. System services offer additional functionality. One of the system services is a provenance service with the goal to generate high-quality provenance, independent of workflow boundaries, for as many different components as possible.

Aperture Tank[2], from here on referred to as tank, is a distributed database that stores

[1] https://nifi.apache.org/

[2] https://github.com/aperture-sh/tank

geospatial data and focuses on visualization and analysis. It is used as the final storage location for geospatial workflows. The database accepts GeoJSON features [5] that are prepared in various processing steps, before they are imported. During the database-import features with unknown fields or incorrect field formats, may be rejected. The rejected features are stored in an auxiliary database the *Exhauster*[3], from which they can be queried and corrected manually. Without provenance the only way to know what has been changed between a feature before and after it has been stored is a direct comparison, albeit without information about possible interim steps. Hence, possible problems within the processing pipelines are hard to find and solve. This problem is even more severe in distributed systems where data is shared between and manipulated by multiple services.

1.1 Motivation

For many datasets meta data is just as, if not more, important as the data itself. Provenance, in its essence, is a piece of meta data, that has seen different data models and has been applied to various different applications and systems, like file systems, workflow management tools, data catalogues, and more. With the growing amount of data, manual management of meta data such as provenance is next to impossible.

A workflow that prepares a GeoJSON feature-collection for the tank database is a good example for this. An initial feature-collection is often only a collection of geometries and their respective coordinates and contain no human readable data. Before they are imported into the tank they need to be prepared. During preparation the dataset gets split apart and features are prepared separately, before they are stored in the database as separate entities. The significance of provenance in such a workflow can easily be described with a simple example.

A user defines a workflow that contains several services, which when executed invoke a process that computes data, or in this case make up the processing pipeline used to prepare and store features in the tank database. The lengthy workflow is executed over night and on the next day, when the user wants to work with his newly generated data, only geometries and coordinates are stored in the database. To make matters worse, half of the data contained in the initial dataset is missing, but the workflow finished without an error. What happened? Since the user did not use a workflow management system that supports provenance, the only way to find out what exactly went wrong is to debug the entire workflow. If the user had used a system with provenance, he could easily reproduce the process chain that generated the incorrectly imported feature, which in return makes it easier to find the error. Furthermore, once the culprit has been found, provenance, depending on the implementation, can be used to find all other datasets that have been corrupted by the responsible service.

[3] https://github.com/aperture-sh/exhauster

1.2 Research Objective

The example in the earlier section shows how provenance can be used to make debugging workflows easier. However, providing detailed provenance is not always simple, especially when multiple different service definitions need to be supported. Hence, the goal of the thesis is to propose a provenance service that is able to generate quality provenance for an orchestrated and distributed workflow management system, independent of service definitions and workflow boundaries. Furthermore, the usefulness of the generated provenance for the system, its services and the produced data is assessed.

To do so an application is implemented that reads, handles and stores provenance data from services within the orchestrated system to generate provenance graphs. In addition, a queryable application programming interface (API) is implemented to retrieve and evaluate the stored information. Hence, the goal of this thesis is to answer the following questions:

- What information is required to generate sufficient and usable provenance information?
- How can the data be accessed via a queryable API
- How can the data be processed efficiently?
- How can provenance be used to increase the quality of the data generated by the application and the application itself?

1.3 Methodology

To study the generation and analysis of provenance in distributed systems the thesis proceeds as follows: Using the specification of a distributed orchestration system the requirements for a service that generates provenance for the system is developed. A message specification will be developed to ensure that the requirements of sufficient provenance based on PROV-DM are met.

Furthermore, an API, which offers methods to retrieve specific entities, activities or agents in addition to full graph representations, is implemented. Together they are used to evaluate the data model, the chosen technologies as well as the developed message specification. The API will be used to evaluate the implemented provenance data model and the value of the generated provenance for the orchestration system, its services and the generated data.

The shortly mentioned import pipeline for the Tank database offers an optimal use-case to implement and evaluate a provenance solution. Hence, scripts and tools will be implemented to simulate its workflow. The results will be used for evaluation.

1.4 Structure

First, the fundamental technologies and design patterns used throughout the thesis are explained. Next, in Chapter 3, the thesis analyzes and summarizes previous work in the field of provenance across different fields.

Chapter 4 introduces the underlying application design and its internally used communication framework before deriving the provenance services requirements from the application's specification. The chapter finishes by introducing the steps and services necessary to realize the Tank import use case.

In the following chapter, the implementation of each implemented component, the use case realization, the used technologies and lastly how the service and its component are deployed is explained. The implementation and its general concept is evaluated in Chapter 6.

Chapter 2

Fundamentals

2.1 Distributed Architectures and Orchestration

Steen and Tanenbaum define distributed systems as "a collection of autonomous computing elements that appears to its users as a single coherent system" that refer to two characteristic features of distributed systems [19]. The first characteristic, the autonomous computing elements, refer to computing nodes that can and will act independently, and hence are autonomous. The second characteristic refers to the outward appearance of a distributed system. Even if it consists of many independent nodes, distributed systems appear as a "single coherent system". Distributed systems are usually organized by a middleware. The tasks of a middleware in distributed systems are for example the management of communication, resources and accounting.

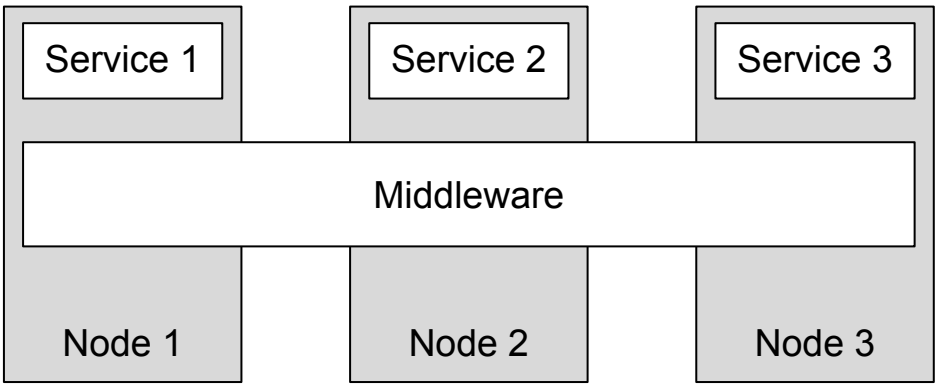

Figure 2.1: Abstract visualization of a distributed system across multiple nodes, which are connected to a shared middleware.

An orchestration service is the implementation of a centralized middleware that controls the workflow of a distributed application. Steen and Tanenbaum compare a centralized organisation to a client server pattern [19]. The server is a service that is implemented on a single node and the client requests a server to do something. To do so

the orchestration service needs to be aware of all nodes within the system. Awareness is gained through different means. One example is a handshake between services. A handshake between services is achieved by one service, for example the orchestration service, sending out a broadcast request to all connected nodes which then reply with which service they implement. If the orchestration service wants a specific service to be executed, the orchestration service takes on the role of the client and requests the target service, the server, to do something. The organization done by the orchestration service includes process and service execution in addition to the organizational tasks of a middleware as shown above.

2.2 Provenance

Provenance is generally defined as the heritage or origin of something. In computing provenance is a metadata component. Geospatial datasets commonly use ISO 19115 family for the description of meta data and provenance information. However, with provenance being not exclusive to geospatial data, other standards have been developed in different fields. Moreau et al. proposed the *Open Provenance Model* (OPM) with the means to create a shared provenance model [13]. Based on OPM, the *World Wide Web Consortium* (W3C) introduced the *PROV* family in 2013. The core of the PROV family is *PROV-DM*, a conceptual and flexible data model for provenance.

PROV-DM. W3C's provenance data model distinguishes between *PROV-DM Types* and *PROV-DM Relations*. Types describe objects, occurrences and responsibilities and the PROV-DM relations are relations between those types. A visualization of the core concept's types and relations can be found in Figure 2.3. The PROV-DM core concept consists of the following types:

- **Entities** are real and imaginary objects or things with fixed attributes.
- **Activities** are occurrences that use or generate entities over a certain period of time.
- **Agents** are things that are responsible for activities or other agents. Agents are not necessarily a person.

The core relations include:

- **WasGeneratedBy** is a relation between an entity and an activity that indicates which activity has generated a selected entity.
- **Used** is a relation between an activity and an entity that indicates which entities were used by an activity.
- **WasDerivedFrom** is a relation between two entities that indicates an entity's origin entity.

- **WasAttributedTo** is a relation between an entity and an agent that indicates which agent is responsible for a specific entity.
- **WasAssociatedWith** is a relation between an activity and an agent that indicates which agent is responsible for a specific activity.
- **ActedOnBehalfOf** is a relation between two agents that indicate which agent is responsible for another one in the specific context of an activity.

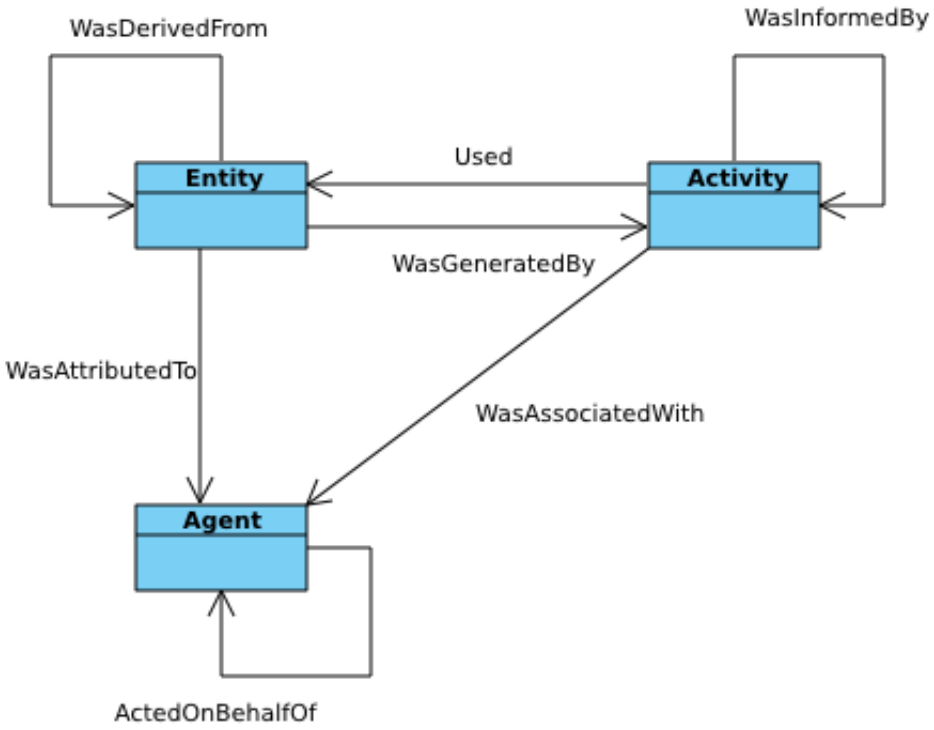

Figure 2.2: PROV-DM core structure in form of an UML [14].

2.3 Advanced Message Queuing Protocol and RabbitMQ

The Advance Message Queuing Protocol (AMQP) is an open internet protocol for business messaging between two parties. It consists of multiple layers that include abstract messaging formats and, on its lowest level, a binary transportation protocol between peers [1].

AMQP is widely used in the industry and has been implemented in multiple products[1], for example Microsoft's Windows Azure Service Bus, Apache Qpid and RabbitMQ. Naik shows that, while not being a global standard like the *HyperText Transfer*

[1] https://www.amqp.org/about/examples

Protocol (HTTP), AMQP is widely used in *Internet of Things* (IoT) and *Machine to Machine* (M2M) settings [16].

AMQP Model. The AMQP model specifies modular components that can be divided into three main types, namely *exchanges*, *message queues* and *bindings* [2], that are connected into processing chains.

- **Exchanges** receive and route messages to queues. Exchanges can be further divided into exchange types. The criteria for message routing is dependant on the exchange's type.
- **Message queues** store the messages that have been routed to them until they are consumed by a client application.
- **Bindings** are the relationship between a message queue and an exchange. Bindings include the criteria used by exchanges for routing to message queues.

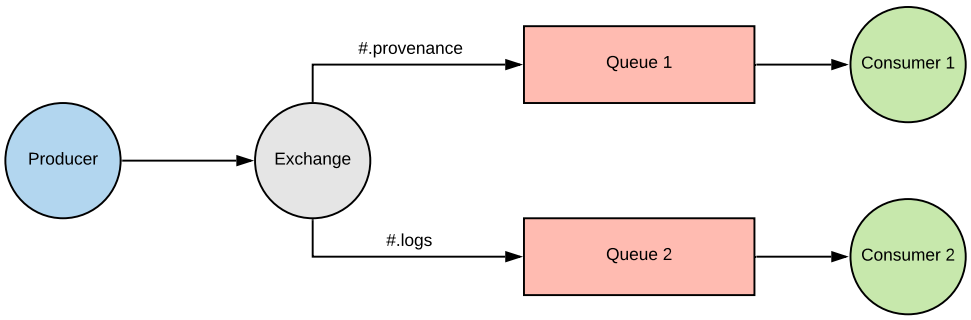

Figure 2.3: RabbitMQ communication model for a topic exchange.

Messaging in Topic-Exchanges. *Topic exchanges* are a specific exchange type that allows the use of binding keys to bind a message queue to the exchange. Messages sent to a topic exchange usually contain a so called routing key, an arbitrary list of words that is delimited by periods, for example "*system.log.warning*". A queue that binds to a topic exchange uses a binding key. Binding keys follow the same rules as routing keys. They can, however, also use the wildcards "*" to substitute exactly one word and "#" to substitute zero or more words. Binding keys are used by topic exchanges to route messages that match a bound queue's binding key to that specific message queue. For example, a queue with the binding key "*#.log.#*" receive any message of which the routing key contains the word "log", this includes routing keys like "*system.log.warning*", "*app.log*", or "*log.error*".

RabbitMQ. *RabbitMQ*[2] is a message broker by Pivotal that implements the AMQP specification. While AMQP version 1.0 became an ISO standard in 2014, RabbitMQ was originally developed, and still uses, AMQP version 0-9-1. However, support for version 1.0 can be achieved by using plugins[3].

RabbitMQ offers cloud and enterprise ready features, for example authentication and *Lightweight Directory Access Protocol* (LDAP) support, and is expandable using prebuilt or custom plugins. RabbitMQ supports multiple deployment strategies[4] for local- and distributed cloud deployment using clusters[5].

2.4 GraphQL

GraphQL[6] is an open-source query language and server-runtime for APIs developed and maintained by Facebook. GraphQL's development started 2012 as an internal project before it was made public in 2015. It was developed to replace *Representational state transfer* (REST) application interfaces in scenarios where the REST model would require too many queries to accumulate the required data, or when the data amount transferred would be unreasonably high due to unnecessary information contained within the data. Nogatz and Seipel [17] show that queries that would require multiple request-response cycles in a REST implementation can be concluded in a single cycle by using GraphQL instead.

Queries. GraphQL queries are basically a selection of objects and their attributes. An object's attribute is called a field while the object itself is a type. In the latest released specification, Facebook describes Queries as selection sets that "is primarily composed of fields" [9]. Due to selecting the fields within a query, a queries result has the same structure and shape as the query itself. Listing 2.1 and Listing 2.2 show an exemplary query and its result.

```
{
    student(id: "1000") {
        uid
        id
        name
        age
    }
}
```

Listing 2.1: Exemplary GraphQL query.

```
{
    "data": {
        "student": {
```

[2] https://www.rabbitmq.com/
[3] https://www.rabbitmq.com/plugins.html
[4] https://www.rabbitmq.com/download.html
[5] https://www.rabbitmq.com/clustering.html
[6] https://graphql.org/

```
            "uid": "0x1"
            "id": "1000"
            "name": "Max Mustermann",
            "age": 42
        }
    }
}
```

Listing 2.2: Result of the query in Listing 2.1.

Types and Fields. GraphQL queries are run against *GraphQL types*. Types are domain objects that contain *GraphQL fields*. For example, when describing a student object in GraphQL, the user would be the type and its fields would be its attributes such as name and age. Listing 2.3 shows the type and field definition for a student object. The exclamation mark means that a field is not nullable.

```
type Student {
    uid: ID!
    id: ID!
    name: String!
    age: Int!
}
```

Listing 2.3: Type and field definition for an arbitrary student type object.

The data type of a field is called a *scalar*. GraphQL provides scalars for primitive data types like strings or integers, however, custom scalar types can be implemented using the official libraries. The example in Listing 2.1 is a query on the type *Student* and its fields *Name* and *Age*, that are of the scalar types *String* and *Number* respectively. Furthermore, fields can have arguments that are used to further filter the data of a field.

It is to note that a field can also be used to describe relations between types. For example, the student type can have a field named *university* of type *University* that is used to connect the two types.

Schema. GraphQL's general design concept is that the underlying business domain is modeled as a graph. This is achieved by defining a *GraphQL schema* based on GraphQL's type system. Within a schema types and the fields of types as well as their relations are described. Additionally, a schema contains *query types*. Query types share similarities with the object types, but are used to define the entry point of a query. They also provide *resolver functions* that are used to accumulate the data that is required to construct the objects the GraphQL server returns. Due to the use of independent data acquisition through resolver functions, GraphQL is not bound to specific databases or backend technologies. Listing 2.4 shows a Go implementation of a simple query type.

```
"student": &graphql.Field{
    Type:        Student,
    Description: "Get student by id",
    Args: graphql.FieldConfigArgument{
        "id": &graphql.ArgumentConfig{
```

```
            Type: graphql.ID,
        },
    },
    Resolve: func(p graphql.ResolveParams) (interface{}, error) {
        \\ accumulate data for student object
        return resolveStudentQuery(p)
    },
}
```

Listing 2.4: Exemplary GraphQL query type implemented in Go.

2.5 Dgraph

Dgraph[7] is an open-source distributed graph database written in Go, which is developed by ex-google engineers. Graph databases is one of the NoSQL[8] database families. Other families include *Wide Column Stores*, *Document Stores* and *Key-Value stores*. Dgraph uses *Badger*, a key-value store, as persistent storage. Batra and Tyagi show that graph databases are significantly faster than relational database models for scenarios where highly relational data is queried [3].

Query Language. Dgraph's query language is *GraphQL+*, a derivative of Facebook's GraphQL. It shares the same design principles as GraphQL, and thus the fundamentals described in the previous section apply. One key difference to Facebook's GraphQL is, that GraphQL+ has been simplified and unnecessary features have been removed, while features required for database operations have been added. Listing 2.5 shows an exemplary query for comparison with the GraphQL query from the previous section. Notable differences are the function declaration in the query header instead of a simple variable binding. This is due to the different operations supported by Dgraph. For example when querying an user by the user ID, different query operations can be used. The *eq*-operation[9] looks for an exact match, while operations such as *anyofterms* can be used to find matching patterns like words in sentences.

```
{
    user(func: eq(id, "1000") {
        uid
        id
        name
        age
    }
}
```

Listing 2.5: GraphQL+ query on an arbitrary user node.

[7] https://dgraph.io/
[8] https://nosql-database.org/
[9] https://docs.dgraph.io/query-language/#indexing

Schema. Unlike many database models, Dgraph does not enforce a pre-defined structure in order to run queries or mutations. If a schema is not defined, the type of a field is inferred on the first mutation that adds it. Schemas can be added or modified at any time. This allows for a very dynamic and flexible schema. However, in order to efficiently index and work with the data in the database, applying a schema is highly recommended.

Within a schema, a field is assigned a data type and indices. The indices are used for different query operations. The earlier example makes use of the *eq*-operation on the *id* field. Therefore, the *id* field has to be given an index that supports this operation. Other operation specific indices are *reverse edges* and *count*. Listing 2.6 shows an exemplary schema that defines the fields used in this section's examples. Note that *uid* is not defined in the schema as it is a default field used by Dgraph and its values are assigned internally.

```
id: string @index(hash) .
name: string @index(exact, fulltext) @count .
age: int @index(int) .
```

Listing 2.6: Exemplary Dgraph schema.

Mutations. A mutation is an operation that inserts data into the database. Dgraph uses the *Resource Description Framework*[10] (RDF) language, specifically the *N-Quad*[11] format, for mutations.

The N-Quad uses triples in the form "`<subject> <predicate> <object> .`". Subjects are nodes in a graph and thus database objects. Predicates are a directed edge to the object — note that this is not the same object as a database object. The object is the target of the directed edge and can be a database object or a literal. A period is used as the delimiter. "`<0x01> <friend> <0x02> .`" is an exemplary triple in which the subject, a node in the database with the uid *0x01*, is linked to the object, another node in the database with the uid *0x02*, by the directed edge *friend*.

If a new database object is to be inserted, a *blank node* has to be used as the subject. A blank node has the form `_:IDENTIFIER`. Blank nodes can be used like variables within mutations. This allows a new nodes to be linked with each other. Note that the blank node identifier only exist during the execution of the mutation and thus cannot be used as parameters for subsequent queries or mutations. For those, the uid assigned by Dgraph has to be used.

Listing 2.7 shows a mutation in which two existing nodes, *0x01* and *0x02*, are linked by the *friend* edge, and two new nodes *_:new-1* and *_:new-2* are added. The blank node *_:new-1* is linked to the existing node *0x01* and the other blank node *_:new-2* is linked to *_:new-1* upon insertion.

```
<0x01> <friend> <0x02> .
<_:new-1> <id> "1003" .
<_:new-1> <name> "Max Mustermann" .
```

[10] https://www.w3.org/RDF/
[11] https://www.w3.org/TR/n-quads/

```
<_:new-1> <age> "20" .
<_:new-1> <friend> <0x01> .
<_:new-2> <id> "1004" .
<_:new-2> <name> "Max Muster-Mustermann" .
<_:new-2> <age> "22" .
<_:new-2> <friend> <_:new-1> .
```

Listing 2.7: Dgraph mutation with existing and new nodes.

Chapter 3

Related Work

Provenance is a well documented and researched topic that has been applied in many different fields for different use cases. In this chapter different provenance concepts, models or extensions to existing models are introduced. After giving a brief overview of provenance concepts for different use cases, modern provenance solutions used for geospatial data are introduced and shortly evaluated. At last this chapter introduces the workflow management system *Apache Nifi* to show how the proposed provenance solution for the orchestration system and the orchestration system itself differ from Apache's solution.

3.1 Provenance in Distributed Environments.

Gehani and Tariq introduce *Support for Provenance Auditing in Distributed Environments* (SPADE) [11], a provenance collection and management infrastructure built on top of a graph-based data model. It supports domain specific semantics in form of arbitrary annotations connected to nodes. Gehani and Tariq discuss different storage solutions and define the capabilities and requirements for provenance systems in distributed environments. Malik, Nistor and Gehani show how SPADE can be used to generate provenance by collecting provenance sub-graphs from different hosts to generate a complete provenance graph [12].

While SPADE seeks to provide a full solution to most provenance problems, their approach is monolithic and requires their own reporters to be connected to the applications that should generate provenance. However, their work clearly shows that a provenance infrastructure can be build using a graph-based data model, in their case OPM, and graph databases.

The provenance solution that is proposed in the thesis seeks to be able to generate provenance for as many different services as possible. Its provenance generation is designed so that it needs as little information as possible to generate provenance. Additionally, the provenance service is meant to fetch missing information on its own. This is in clear contrast to Gehani and Tariq's solution that relies on the reporters to

provide full provenance coverage of provenance components that are to be stored in their system.

3.2 Provenance-Aware Storage.

Provenance at its core is essentially meta data and can be applied to anything on which the basic concept of heritage can be applied on. This means that it can be used for more than service driven environments like workflow execution systems. Muniswamy-Reddy, Holland, Braun and Seltzer show that provenance is not only applicable in service driven environments, but also in a system level storage systems [15]. Zhao, Shou, Maliky and Raicu expand on the file system idea and introduce a distributed and provenance aware file system [22]. This is achieved by combining the distributed file system FusionFS with SPADE [11].

3.3 Semantic Provenance.

Sahoo, Sheth and Henson introduce semantic provenance [18]. Semantic provenance is an approach in which common models of provenance representation are extended with key concepts of semantic metadata with the goal to make provenance more useful in eScience. They argue that in order to be usable in eScience, provenance needs to be "software-interpretable and expressive". To achieve the combination of semantic metadata and provenance they add "domain knowledge and ontological underpinning" to the existing model. This results in a three-dimensional provenance model in which domain specific provenance objects, achieved by domain knowledge, are combined with semantic annotations like time and space. Sahoo et al. show that semantic provenance offers many advantages. However more modern provenance data models like PROV already support semantics[1]. Hence, a manual extension of provenance to support semantics is no longer required.

3.4 Geospatial Provenance.

Closa, Masó, Proß and Pons apply the W3C PROV model and other provenance models to geospatial data [7]. Their work shows that for pure geospatial use cases a single provenance model may not be enough. However, PROV's design make it the best available choice for most use cases. They also show that PROV is easily expandable and can serve as a base for a combined provenance model.

Garijo, Gil and Harth discuss the challenges of modeling geospatial provenance [10]. They define seven categories of questions that form the requirements for geospatial provenance. The categories are divided into three levels: dataset, object and property requirements for a single dataset and sets of datasets. Their levels are defined as follows:

- At the **dataset level** the entire dataset is considered a single provenance entity.

[1] https://www.w3.org/TR/prov-sem/#semantics

- At the **object level** the objects contained in the dataset are asserted.

- At the **property level** provenance is used to answer questions about a single object's properties, their attributes and the corresponding values.

The last category are requirements that cannot be categorized into the mentioned categories. Like Closa et al. [7] they base their provenance on the W3C model and expand it to fit their requirements. Doing so they again prove the extendability and viability of a W3C based provenance approach.

Yue and He define general considerations and geospatial considerations for provenance in modern infrastructures [21]. Their practical main considerations are the aggregation and availability of provenance in applications. The main focus of their work is the applicability of provenance for geospatial data. They share main ideas with Closa et al. [7] and Garijo, Gil and Harth [10] and define the requirements for geospatial provenance. Yue and Hes main requirements are space and time related information, the granularity and the scalability of the applied provenance model. Di, Yue, Ramapriyan and King [8] base their approach on Yue and He's publication [21] and extend it by defining general guidelines, key considerations and introduce potential solutions for future work in that field.

The approaches introduced in this paragraph prove that PROV is applicable for geospatial provenance. Moreover, the requirements that a provenance data model needs to fulfill to be a viable option for provenance in geospatial processing are usually more complex and require more consideration than provenance for non geospatial data. The orchestration system is able to execute both geospatial and non-geospatial workflow scenarios and thus the W3C PROV model is a suitable approach to a provenance solution for that system.

3.5 Retrieval and Storage of Provenance.

The W3C PROV-DM can be modeled as a graph. Graphs are usually highly relational data. If such data would be translated to common relational database models, the data would have to be divided into several tables. Hence, querying graph models from relational database models would result in multiple "JOIN" statements, which, in theory, is rather slow. Vicknair et al. compare a MySQL database to the graph database Neo4j [20]. Their results show that graph databases outperform relational databases in queries that return a structural representation of a graph. Their work also shows that due to the indexing mechanisms in graph databases, particularly Neo4j, graph databases can outperform relational database models when requesting single database entities that are queried by full text search. They further show that, while graph databases outperform relational databases in string based queries, relational databases are far superior in integer based single entity queries. In addition to their performance analysis, Vicknair et al. discuss security concerns when using graph databases as they usually do not provide multi user management.

Bryant shows how GraphQL can be used to retrieve heritage data [4]. While he does not work with provenance, provenance and heritage data share enough similarities to consider his approach. Bryant also discusses the advantages and limitations of using a GraphQL API together with a graph database. His main advantages are the standardized API that allows simple integration of third-party tools, the well documented GraphQL standard and the high-quality feedback provided by the API server. Furthermore, GraphQL can simplify the maintenance of internal and external data models as its design allows it to be used with any data model on the backend.

3.6 Provenance Quality.

Cheah and Plale show how provenance can be analysed with the means to increase its quality [6]. They argue that most applications that generate provenance do so passively and thus anomalies that may have occurred within those applications are also present in the resulting provenance representations. They define two dimensions of provenance quality, the completeness and the correctness. The correctness refers to the contextual integrity of provenance data and includes faulty provenance entries caused by errors in workflow execution or problems with the provenance generation itself. The completeness is the structural integrity of a provenance graph. Their work shows that evaluating the quality of provenance graphs is important to find and isolate potential problems with not only the provenance collection but possibly the application or workflows executed by the system.

The procedures Cheah and Plale introduce should be applicable to most provenance data models, including PROV. Due to the amount of different services that can and will generate provenance during execution within the orchestrated system, the quality of the generated provenance should be analysed frequently. Cheah and Plales approach appears to be suitable for this analysis.

3.7 Apache NiFi.

Apache NiFi[2] is a workflow management system. That has been originally developed by the *National Security Agency* (NSA). It is designed to control the data flow between services in distributed and non-distributed settings. NiFi supports data provenance to monitor and analyse the data flow.

While NiFi and the orchestration system share similarities like a distributed architecture and support for data provenance, NiFi lacks the ability to orchestrate services. Additionally services, or processors as they are called in NiFi, are required to be specifically implemented for NiFi. Whereas the core concept of the orchestration system is to be able to connect any service to the system without developers having to implement specific interfaces.

[2] https://nifi.apache.org/

NiFi handles provenance by tracking provenance events[3] which essentially are PROV activities. In contrast to the provenance solution developed in the thesis, NiFi does not handle provenance on an entity base. This means that instead of creating a new provenance entity which is connected to an activity, which has created this entity, NiFi tracks the changes to the input data during the workflow execution. As a result their provenance model is limited to and exists only in the context of its corresponding workflow. The provenance solution developed for the orchestrated system however aims to support provenance relations across multiple workflows or scenarios.

[3] https://nifi.apache.org/docs/nifi-docs/html/user-guide.html#data_provenance

Chapter 4

Concept

4.1 Application Design

The provenance service is designed to be used within a distributed and orchestrated application and hence the central concepts of the application need to be considered. The goal of this chapter is to describe the central concepts and requirements of the application. Note that the application described in this chapter is in a development stage. Thus, the concepts for both, the orchestrated application and the provenance service, are not final. However, the fundamental design principles will still apply.

Scenario. The main purpose of the application is to execute scenarios. *Scenarios* are a collection of tasks which, when executed, make up a process pipeline. In other words, a scenario is a predefined and repeatable workflow that consists of multiple execution steps and is created by an application user. Additionally, a scenario contains all information required to execute a task specifically for a scenario. The information can include command-line parameters, paths to executables, paths to in- and output files, or directories respectively, and more. Scenarios are given an unique ID, called *scenarioID*, by the orchestration application that is used internally to differentiate between them.

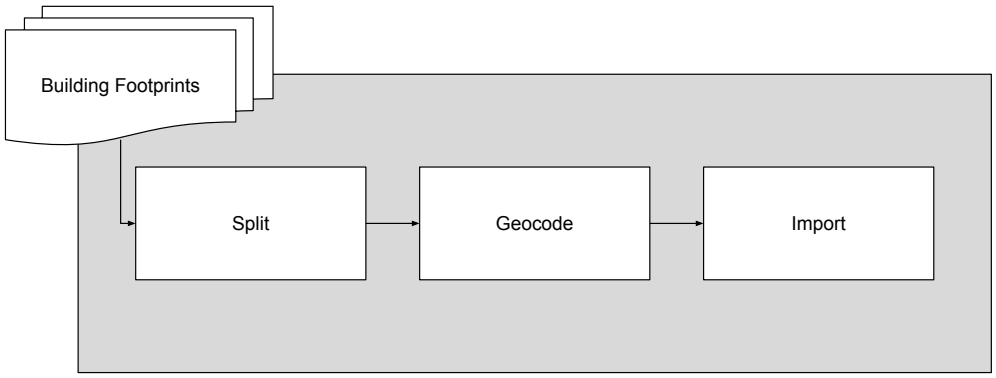

Figure 4.1: Abstract scenario of the tank-import use case.

Service. A *service* is an user defined, executable task within a scenario. Services usually, but not necessarily, use command-line tools that accept parameters and runtime arguments that are managed by a scenario. Each execution is unique, even if it is the same service and are called process. Services, just as scenarios, are given an unique ID called *serviceID*.

Process. A *process* is a single, unique runtime of a service within a scenario. Processes are also given unique IDs called *processID*.

System Services. System services are services provided by the application. They include services to generate and retrieve system logs and metrics, provenance, as well as administrative services that can be used to retrieve information on processes, scenarios, services and users.

Requirements. In order to execute and manage its corresponding services as well as system services, the application needs to meet the following requirements:

- The application needs an orchestration service that manages scenarios and services.
- The orchestrater needs to be able to communicate with the scenarios and services in order to manage them.
- The services need to be able to communicate with scenarios as well as other services.
- The application needs a defined, yet as generic as possible, message format to allow communication for as many differently designed services as possible.

4.2 Communication within the Application

The application, scenarios and services communicate by sending JSON encoded messages using the message broker RabbitMQ. Before the usage of RabbitMQ can be further described, the messages that need to be exchanged by RabbitMQ need to be considered. Messages exchanged within the application can be divided into the following basic types:

- *Status messages* are sent when a process spawned by a service, or a service itself changes its state, for example when it finished executing and changes to a completed state.

```
{
    "timestamp": "2019.12.01 12:12:12.000"
    "started": "2019.12.01 12:12:11.000"
    "status": "complete"
    "numResults": 42
}
```

- *Log messages* are further divided into different log-levels and can be both, process related or scenario related. Scenario related log messages are sent when for example the scenario moves to the next service, or when a service spawns a new process. Process logs can contain messages such as access notifications and error messages and are specific to a process.

```
{
    "timestamp": "2020-01-16 12:00:00",
    "message": "failed to parse feature <125caf57-2221-4a11-
        a66c-f01a7eba5ae1>",
}
```

- *System messages* are messages used for orchestration or handshakes between services. For example, in order to establish communication between a new service that is connected to the system infrastructure and other services, the new service sends out a system wide message to make other services aware of it.

```
{
    "timestamp": "2020-01-16 12:00:00",
    "status": "ready",
    "type": "database",
    "url": "http://192.168.0.42:8080",
}
```

In order to make the messages exchanged in RabbitMQ as slim as possible, the communication design relies heavily on RabbitMQ's topic exchanges. Topic exchanges allow the use of routing keys to route messages to specific queues connected to an exchange. Furthermore, every scenario is given its own dedicated topic exchange.

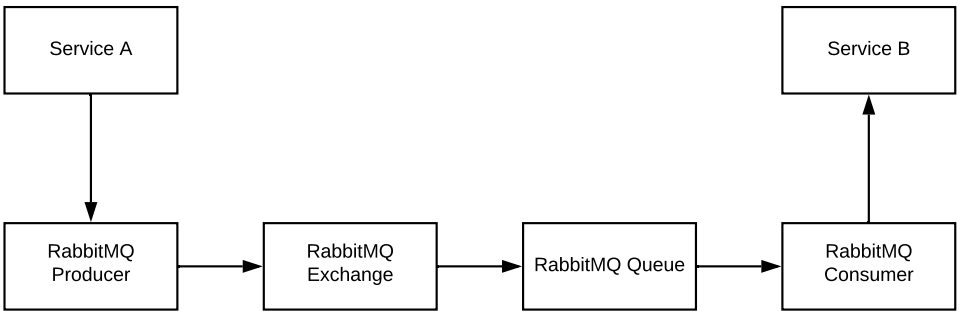

Figure 4.2: Communication between two services via the RabbitMQ message broker.

Using the message types above, a simple routing key scheme can be established. The first word of the key is used to differentiate between scenario related log or system messages and service messages. Note that, since a process is invoked by service, process messages always have the ID of its corresponding process as the first word of the routing

key. However, because every scenario has its own exchange, the scenario itself does not need to be part of the routing key. Thus, the first section can for example be *log*, *system*, or *serviceID*.

The second word is used to give more details about the message. Scenario related log messages for example use the second section of the routing key to specify the log level. Service messages use the second key to differentiate further between either process messages, or service log, or status messages. Examples for the second section are *status*, *log*, or *processID*. The third and following sections are analogue to the second one, with the difference that it is used for processes.

The following examples for routing keys make use of the pattern described above:

- *service42.status* contains a status messages sent by the service with the ID *service42*.
- *service42.process123.log.error* contains a log message, with the log level error, sent by the process with the ID *process123* that is invoked by the service with the ID *service42*.
- *service42.process123.status* contains a status messages sent by the process with the ID *process123* that is invoked by the service with the ID *service42*.

Message Format. Due to the use of one topic exchanges and routing keys, the body of the messages exchanged does not have many requirements. Furthermore, scenario, service and process information is not required within the messages, as they can be queried from system services by using the IDs found in the routing keys and the exchange name. As a result, most fields used within the message body are optional or service and scenario specific.

4.3 Provenance Service Architecture

The Provenance service is a system service that connects to scenario specific exchanges in order to generate and store provenance. This section further describes the provenance service's requirements and architecture. Given the application and communication design described in the previous section, the provenance service must fullfil the following requirements within the orchestration system:

- The service needs to be able to listen to scenario specific RabbitMQ exchanges.
- It needs to be able to establish or revoke connections to exchanges during runtime in case scenarios are added or removed.
- The provenance service needs to be able to listen and respond to global messages in order to provide information on how to retrieve the stored provenance.

Additionally, the service must meet the following performance and design goals:

- The provenance service needs to be able to handle high loads and scale when the application demands it.
- It needs to be able to generate provenance with as little information as possible.
- The service needs to store provenance efficiently.
- The service needs to provide API endpoints to allow other services or administrators to access the data.

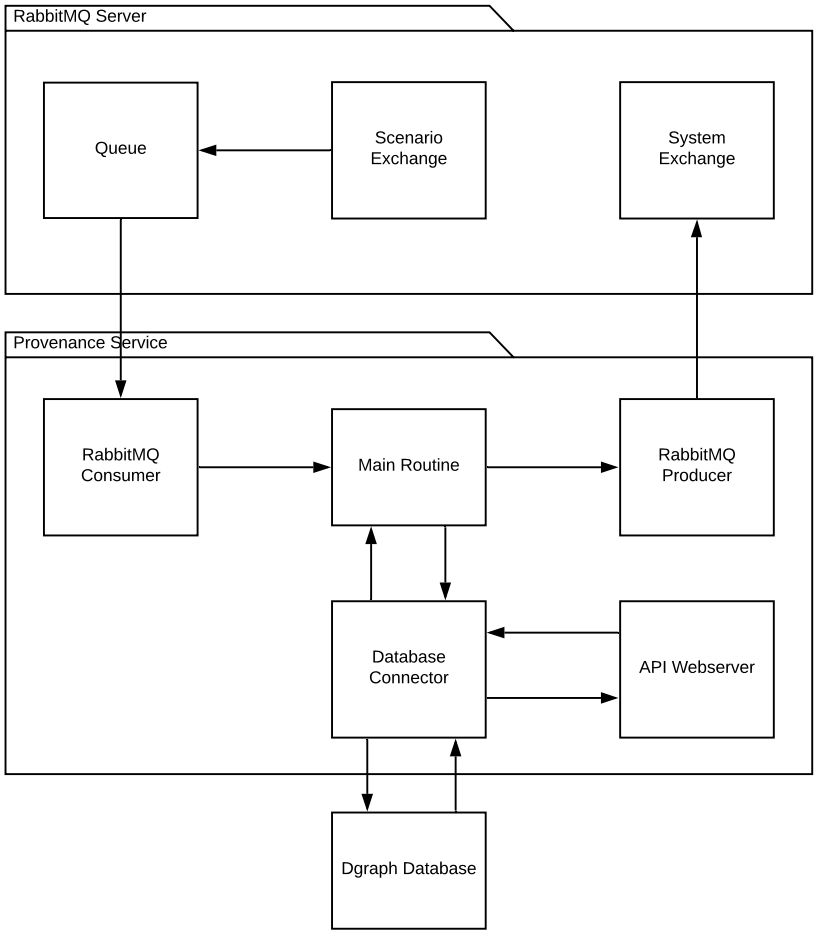

Figure 4.3: Provenance service architecture.

Provenance Messages. Services that want to provenance to be produced send out provenance specific messages. Thus, in addition to the existing message types, a fourth message type is introduced.: *Provenance Messages*. Provenance messages are routed by a dedicated routing key within a scenario's dedicated exchange. Expanding the

example earlier, Listing 4.1 shows an examplatory provenance message. Because every provenance message corresponds to exactly one provenance entity, provenance messages provide a single string for the output filed. However, the input field is defined as a list of inputs, because a single entity can be derived from multiple ancestors.

```
{
    "timestamp": "2019.12.01 12:12:12.000",
    "input": ["b4c09cf6-0b22-4de6-b8da-19a7c8158061"],
    "output: "ea46abad-0f34-4aac-be6e-197891d79e3d"
}
```

Listing 4.1: Example body of a provenance message.

Communication Components. The provenance service needs to be able to communicate with the orchestration system. This is achieved by connecting RabbitMQ consumers to the system exchange, an exchange dedicated to provisioning and system wide messages. Additionally, the service needs a RabbitMQ producer component to send log messages, metrics and general information such as the API's URL to the system. Each scenario has a dedicated exchange and hence the provenance service starts a consumer for each scenario exchange. The consumer listens to a queue that is bound to the exchange using a binding key that matches a provenance message's routing key.

Extracting and Storing Provenance. The message body of consumed messages alone does not contain enough information to generate sufficient on PROV-DM based provenance. The provenance service makes use of the system services to accumulate all further information required to build provenance based on the basic components of PROV-DM. As shown in section 2.2, the basic components of PROV-DM are activities, agents and entities. Considering the applications design as well as the basic definition of processes and services, this data model can be applied as follows.

By definition, a process is the actual execution of a task described by a service. Hence, they are considered an *activity* from a provenance perspective. In addition, the service is responsible for the execution, as it contains all necessary information. A service can thus assumed to be an *agent*. However, the service has been designed by an user, which makes the user an *agent* for whom a service acted on behalf of. The last component, *entities*, are simply the in- and output data that is used by, or generated by a process. The input data is the original entity and the output data is assumed to be its derivative. This also applied when multiple inputs get merged into one output or one input is split into multiple outputs.

Assuming that a process sends out a provenance message containing information on in- and outgoing data, provenance can be extracted relatively easy. The information on in- and output from within the message body can be used to cover the original entity and its derivative. As mentioned earlier, scenarios use routing keys that contain the service-ID and the process-ID when sending producing provenance messages. The sections on the routing key can be used to fill the remaining components. For example, a routing key *service42.process123.provenance* used for a provenance message on an exchange with

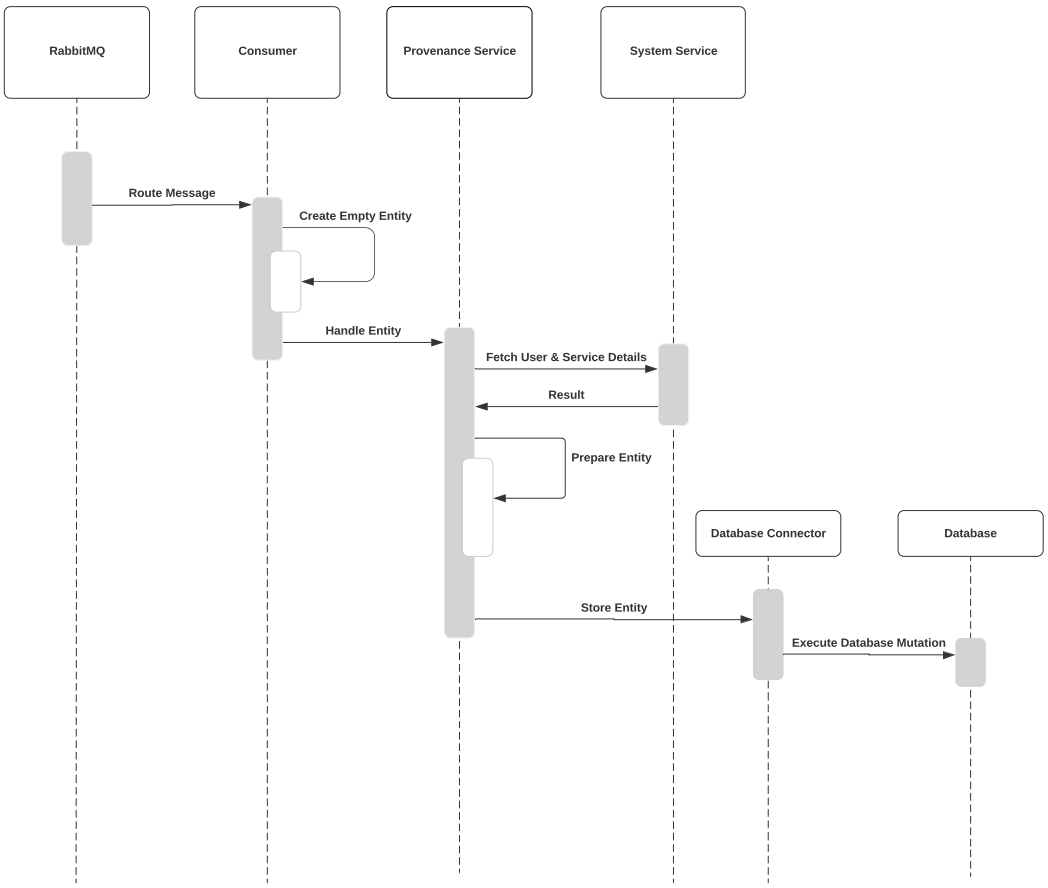

Figure 4.4: Sequence diagram for the provenance generation of a single provenance entity.

the ID *scenario1* can easily be broken down into activity related, process-ID, and agent related, service-ID, information. These IDs can be used to query system services for information on what exactly the process did, how long it ran, or even who a service was created by. With all information gathered, a provenance graph can be created. Figure 4.5 is a visualization of a simple provenance graph that represents the generation of a single provenance entity and all parties involved. This graph is then stored in a database.

Service Interface. In order to make stored provenance available to the application as well as other services a server based API is required. To be able to work with provenance both visually and with the provenance data, endpoints that offer a data driven representation of the graph are required. While the graph's root is usually an entity component, users might want to choose different components as root nodes depending

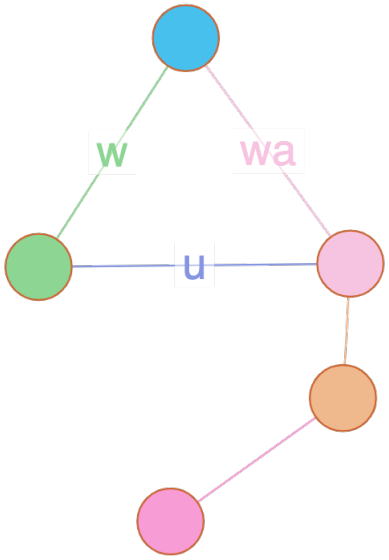

Figure 4.5: Provenance graph of a single entity (blue) and the directly involved parties, color coded as follows: ancestor entity (green), activity (pink), agent (orange), responsible agent (magenta).

on their use case. For example, one user wants to know which entity the predecessor of a derivative entity was, while another user wants to see all entities generated by the processes spawned by a specific service. Hence, the service interface needs to support graph representations with all three basic provenance components as root nodes. In addition to endpoints offering a graph representation based on an entity, the API offers endpoints to query detailed information on single provenance components such as agents. A suitable approach for a service interface like this is a queryable API.

Chapter 5

Implementation

Based on the observations in Section 4.3, a prototype is developed. The prototype is used to proof the concept and architecture of the provenance service and is thus used for evaluation.

It has to be noted that some system components are not implemented yet. This is due to the fact that the application, and thus orchestration system and system services are in a pre-development state. Some of the missing components, such as the system services that handle orchestration and provide detailed information on scenarios, services and processes, are required to generate provenance. However, these services are not part of the provenance service and thus won't be further discussed in the thesis. Some of the missing behaviour is replicated using auxiliary scripts or is hard-coded into the service. Their theoretical concepts and how they would be used if they were implemented have been described in the previous chapter.

The system can be divided into four main components. The first one is the communication component that creates and manages the RabbitMQ consumers and producers. The second component is the actual provenance service, which receives so called deliveries from the consumers. Deliveries are a wrapper for an unprepared provenance entity. The service first accumulates missing data and prepares the provenance entities to be stored in the database. Prepared provenance entities are batched and lastly stored in the database. The third component is the database connector, which manages the database session and offers functionalities used during the accumulation process and provenance retrieval. The final component is the API-Server. While the server is deployed in its own instance, it makes use of Go's package system to reuse the communication and database connector components.

5.1 Technologies

The service and the required RabbitMQ components are implemented in Go. Provenance generated by this service is stored in Dgraph, a distributed graph database. Instead of a commonly used REST-API, the provenance services uses GraphQL to support

queryable API endpoints. This section introduces the technologies used to implement the provenance service.

Go. *Go*[1] is an open source programming language developed by Google engineers. It follows *C's* basic syntax and borrows ideas from various other languages. The language is designed to be fast to work with, i.e., fast compile times, easy deployment and a rich standard library.

Go has been selected because of several reasons. It is subjectively easy to work with and allows quick prototyping. Moreover, it has very good support for event-based asynchronous applications and offers decent scalability. This is achieved with *goroutines*, which are lightweight threads that are managed by the Go runtime. To communicate between goroutines, Go uses so called channels. They are allocated like maps and arrays and act as a queue. There are different types of channels; buffered and unbuffered channels. Go applications usually make heavy use of structs. Structs can be used to bind attributes and functionalities to objects. While structs are in some way similar to classes found in object oriented languages, they are still different since Go uses a different approach. While still a comparatively young language, Go is mature enough to be consistently rising in popularity in developer surveys and actual production usage. Popular projects include Docker and Kubernetes.

Go offers all the tools necessary to build an event-based provenance service and an API-Server. Its big and well maintained standard library removes the need of frameworks and allows for a slim application without bloat.

Dgraph. Provenance is highly relational meta data that can be represented in form of a graph. Storing graphs in relational databases like PostgreSQL would require multiple join operations. This has a big performance impact when querying the database. Thus, graph databases are a very good choice for storing provenance.

The most commonly used graph database is Neo4j[2]. Dgraph, however, provides a fresh and interesting approach to graph databases. Even thought it is a relatively young database, Dgraph offers promising features with seemingly good scalability and performance.

GraphQL. In section 4.3, one of the mentioned requirements for the provenance service is to offer the user freedom when querying provenance data. He should be able to receive graph representations independent of the chosen provenance component as well as be able to extract only the necessary information on a single provenance object. GraphQL allows this feature. Additionally, while multiple ways a user can query provenance is supported, the used data model is static and thus the types and field are also static. The fact that types and fields are not dynamic allows a simple implementation of the GraphQL required functions.

[1] https://golang.org/
[2] https://neo4j.com/

5.2 Initialization

For the time being the service initialization is hard-coded into the service. However, assuming the system services were implemented, the hard-coded information such as URLs to system services and databases, as well as exchange names, would be provided by the orchestration system after a successful handshake.

Upon start-up, the service first fetches collects requirements such as service and database URLs and stores them in a configuration struct. Using the information contained in the configuration struct, connections to the database and RabbitMQ are established. Once a connection the RabbitMQ server is established, the required consumers and producers are initialized. Lastly the provenance service itself is invoked in form of a goroutine that receives messages from the consumers via a Go channel shared between the components.

5.3 Communication

The orchestration system and the connected services use RabbitMQ for communication. RabbitMQ is an implementation of the AMQP standard and hence any technology or library that implements this standard can be used to communicate within the system. The Go package AMQP[3] is used to implement the required consumers and producers.

While the provenance service uses multiple consumers to listen to multiple provenance sources at once, their behaviour is exactly the same, because they are just multiple instances of the same consumer. Thus, only a single implementation of a provenance consumer is needed. The only exception are the consumers used for orchestration or system messages, which are not further discussed.

The consumer described in this section assumes that provenance messages are sent using dedicated provenance routing-keys.

Provenance Consumer. The connection created during initialization is passed down to and shared between all provenance consumers. For each provenance source, or exchange used by a scenario, a consumer is initialized. While consumers share a connection, each consumer creates its own RabbitMQ channel to connect to queues. This is due to the fact that channels are not thread safe[4] and each instance of a consumer runs within its own goroutine and hence its own thread. It is to note that, while the statement about thread safety is for the official Java client, the same principles can be applied to any implementation of AMQP, including the one used for this implementation. Once a RabbitMQ channel has been created, the consumer declares its dedicated exchange and connects to a queue. Finally the queue is bound to the exchange by a binding key that matches the pattern used dedicated provenance routing keys. The pattern used to match any provenance message that is sent with a dedicated provenance routing-key is *#.provenance*. This pattern matches any routing-key that has the word *provenance* as

[3] https://github.com/streadway/amqp
[4] https://www.rabbitmq.com/api-guide.html

its last section, for example *service42.process123.provenance*, *service42.provenance* and *provenance*.

Messages received by the RabbitMQ consumer are used to initialize structs that represent provenance entities. To do so, an empty struct is initialized and some of fields, like the entities ID, its ancestors ID, and all other fields that can be filled with the information contained in a provenance message are filled. The entity struct uses a nested structure that later on can be easily parsed to the JSON, which is required for the database mutation. An exemplary JSON representation can be found in Appendix A. Each nested object represents a provenance component and the names of each nested object represent the relation between the nested object and its higher level component. Note, that the *UID* field of each object contained in the struct is initialized with strings in the format `_:<ID>`. The UID field is used to connect related provenance objects and will be further explained later. The entity struct is then forwarded to the provenance service using a shared Go channel. All consumers use the same channel to forward their messages.

5.4 Provenance Generation

Due to the use of a message broker, the provenance service is required to work asynchronously and event based. It is thus implemented in form of a goroutine that receives deliveries from the communication components. However, because of the database's import handling, more in Section 5.5, incoming deliveries need to be handled in batches. Thus, an auxiliary goroutine is implemented. This goroutine, as seen in Listing 5.1, forwards deliveries to the main goroutine, or, if no new delivery has been received within a certain time window, uses the *commit* channel to tell the main goroutine to write to the database before the batch is full.

The auxiliary goroutine makes use of Go's select statement. Select statements are used to handle multiple operations based on channels. By design select statements block until a case can be run. The select statement used consists of two cases. The first one is run when a new entity (delivery) is forwarded by a consumer, while the second case is run automatically after a timer expires and is used to prematurely commit a batch if no new entity has been received within a certain time window. Due to the usage of an infinite loop, the select block is repeated whenever one of the cases is run. This allows a permanent stream of entities to the main goroutine. If the second case is run, the execution is blocked until a new entity is received.

```
go func() {
    for {
        select {
        case derivative := <-tracer.deliveries:
            derivatives <- derivative
        case <-time.After(batchTimeout):
            commit <- struct{}{}
            derivative := <-tracer.deliveries
            derivatives <- derivative
        }
```

```
        }
    }()
```

Listing 5.1: Gateway goroutine used for forwarding entities and committing if no entity has been received in a certain time window.

Before the deliveries forwarded to the actual provenance service can be used as provenance, additional information needs to be accumulated. To do so, the information within the exchange name and the routing-key contained in the delivery is used to fetch the required information from system services. As described in Section 4.2, each section of a routing-key, and the exchange name itself, are unique IDs used within the orchestration system to identify processes, scenarios and services. After accumulating all necessary information, the provenance object is then prepared for storage within the database.

Accumulating Information. System services offer API endpoints, which can be used to fetch the required information using unique IDs contained in the routing keys. Using the unique IDs HTTP requests are sent to the endpoints for service and user information. The services return JSON objects that contain the requested information. Listing 5.2 shows an exemplary response to a request about details of a service. The information returned by the services are then used to fill the missing fields in the entity struct.

```
{
        "id": "125caf57-2221-4a11-a66c-f01a7eba5ae1",
        "name": "Split Feature-Collection",
        "description": "Splits a Feature Collection into single
           features",
        "type": "service",
        "createdBy": "2c2755df-ca17-4a01-a4a5-bfd91b3e18f8"
}
```

Listing 5.2: Exemplary response body of a GET-request to a system service to retrieve information about a service.

Storage Preparation. Once the information on the provenance components activity, agents and entities is accumulated, the provenance object can be prepared for storage. When a provenance object is inserted into the database, two cases need to be considered. Firstly, a provenance component, namely activity, agent or entity, is not present in the database. In this case the initial string values of the UID fields are used to later identify the nodes. Secondly, if any component is present within the database, its initial string value is replaced with the UID used within the database. Only the strings of present components are replaced, while the not present ones keep their initial string values. In order to determine whether a component is present in the database, the database is queried using the database connector component. Listing 5.3 shows the query used to retrieve the UIDs of all possible provenance components, if it returns an empty value, the component is not yet present. It is to note that, in addition to the three main

components, an auxiliary component *supervisor* is used within the preparation steps. This is to differentiate between the acting agent, the agent who is responsible for the process, and the supervising agent, the agent whom the acting agent acts on behalf of. The retrieved database UIDs are cached in a simple map to avoid unnecessary repeated database transactions. The map uses the IDs from the orchestration system as key and the database's UID as value. If the query result was empty, the initial string value is stored instead. In case that an ID is present as a key in the cache, the value mapped to the ID is always used, even if it is the initial string value.

```
QueryAllUIDsByID = `
    query All($entity: string, $activity: string, $agent: string,
        $supervisor: string) {
        entity(func: eq(id, $entity)) {
            uid
        }
        activity(func: eq(id, $activity)) {
            uid
        }
        agent(func: eq(id, $agent)) {
            uid
        }
        supervisor(func: eq(id, $supervisor)) {
            uid
        }
    }`
```

Listing 5.3: Dgraph query to retrieve internally used UIDs by IDs that have been assigned by the orchestration system.

5.5 Provenance Storage

After all node IDs are set, the prepared provenance object can be stored in the database. The provenance service uses the graph database Dgraph to store provenance objects. While Dgraph supports concurrent read operations, concurrent writes can be problematic. When two or more transactions want to mutate the same database entry, for example, two operations want to connect a new activity node to an existing agent, only one of those operations succeeds while the others fail. However, handling all provenance objects sequentially, as in creating a single database transaction that fills the ID field of present components, is also not beneficial due to it resulting in multiple small gRPC requests per object and hence slowing it down further. Thus, to increase the write performance of the database, the provenance service batches all incoming provenance objects. This is achieved by appending every new provenance entity to a list of entities that is kept until the batch is committed.

Batches are committed when a predetermined batch size is reached, or the commit signal is sent by the gateway goroutine, because no new provenance message has been consumed within a certain time window. When a batch is committed, its collected entities are converted to a list of JSON objects and embedded in the mutation field

of a Dgraph transaction. Using the database connector component the transaction is executed.

Batch Imports. The initial string values of a provenance component's UID consist of the components name followed by a number, for example *_:entity-1*. The number indicates the current size of the batch and is reset together with the batch list when the limit is reached. These initial string values are variables used to reference nodes within a batch. This is possible due to how Dgraph, and specifically RDF, handles nodes. Before going into detail, it needs to be considered how Dgraph handles mutations. Every mutation run in Dgraph uses the *RDF N-Quad* format. If a mutation is not in the RDF format, like the JSON mutations used for the provenance service, Dgraph first converts the mutation into RDF before proceeding. To draw edges RDF uses a node identifier followed by the edge's predicate and another node identifier, for example:

```
<0x01> <name> "Alice" .
```

If the identifier is unknown, or simple not present, a blank node has to be used, which is later replaced by the internally assigned UID. Blank nodes are essentially variables that only exist during the lifecycle of a single database mutation.

When inserting a batch of objects into the database, there are multiple possible scenarios where blank nodes are required. However, by using the cache map described in Section 5.4, the initial string values can easily be used to draw edges between new nodes within a batch, because they follow the RDF specification for blank nodes. For example, if the key with the ID *process123* is mapped to the value *_:activity123*, and an entity contains a nested activity object with the same ID, the value of its UID field is overwritten with the value from the cache, in this case *_:activity123*. This allows Dgraph to draw an edge to that specific node and avoids duplicate nodes.

Listing 5.4 shows the routine that builds and executes the database mutation. It accepts a batch of prepared provenance entity object that get marshaled into a list of JSON encoded entities, as shown in appendix A. The resulting list is then bound to a Dgraph transaction's mutation field and executed. The UID's assigned by Dgraph are returned.

```
func (c *Client) RunMutation(mutation *[]*util.Entity) (*api.Assigned
    , error) {

    \\ initialize new database transaction and discard it when the
        function returns
    txn := c.conn.NewTxn()
    defer txn.Discard(context.Background())

    \\ convert the mutation (list of entities) to JSON
    payload, err := json.Marshal(mutation)

    ...

    \\ initialized new dgraph mutation object
    \\ CommitNow is used to indicate that no other mutations or
        queries are part of this transaction
```

```
    mu := &api.Mutation{
            CommitNow: true,
    }

    \\ bind the JSON payload to the mutation and execute it
    mu.SetJson = payload
    assigned, err := txn.Mutate(context.Background(), mu)

    ...

    return assigned, nil
}
```

Listing 5.4: Method for committing a batched Dgraph mutation.

5.6 GraphQL Integration.

In order to retrieve provenance from the database, an independent GraphQL API server is developed. Just like the actual provenance service, the API server uses a single producer and consumer to listen and respond to system messages for orchestration and service awareness. The following section further explains the server-sided GraphQL integration using GraphQL's own Go implementation.

To make use of GraphQL, a web server needs to be implemented first. This is achieved using the http package of Go's standard library. Go's web server uses handler functions to register functions to specific URLs. For example, a URL *http://api-host/api* would be routed to the handler function that is bound to */api*. A single exposed endpoint is enough for complete provenance retrieval using GraphQL.

GraphQL integration into the API is achieved, by binding a GraphQL schema to an API's handler function. The schema contains the configuration of both query and mutation objects. Since the API is read-only, only the query configuration is implemented. The configuration of mutation objects will not be further described in this section.

GraphQL Schema. A GraphQL schema is an object that consists of the complete configuration of mutation and query objects. GraphQL objects use fields to bind information to specific keys. The query object used to realise the provenance API uses four fields. One field for each provenance component: activity, agent and entity and a field for an experimental workaround used to retrieve a predefined graph structure, the workaround query is further explained in Section 6.6. Listing 5.5 shows an exemplary implementation of a single field query object.

```
graphql.ObjectConfig{
    Name: "Query",
    Fields: graphql.Fields{
        "entity": &graphql.Field{
            Type:        entityType,
            Description: "Get entity by id",
            Args: graphql.FieldConfigArgument{
                "id": &graphql.ArgumentConfig{
```

```
                    Type: graphql.String,
                },
            },
            Resolve: func(p graphql.ResolveParams)
                (interface{}, error) {
                return resolveQueryEntity(db, p)
            },
        },
    },
}
```

Listing 5.5: GraphQL field object for a provenance entity.

GraphQL Type. In order to define the properties of each GraphQL field, a type object is implemented for each provenance component. Using the figure above as an example, the *entity* field is of type *entityType*.

```
var entityType = graphql.NewObject(
   graphql.ObjectConfig{
       Name:        "Entity",
       Description: "Provenance Entity Object",
       Fields: graphql.Fields{
           "uid": &graphql.Field{
               Type: graphql.String,
           },
           "id": &graphql.Field{
               Type: graphql.String,
           },
           ...
       },
   },
)
```

Listing 5.6: Type object used to describe provenance entities.

All three provenance component fields follow the same schema and only some field names are different. However, the graph field greatly differs. Instead of simple key value pairs, such as *id*, or *name*, the workaround types's uses a single field that is required to be complete graph representations encoded as JSON. Thus, the scalar used for the field needs to be a custom scalar. Custom scalars require a serializer and parser for literals and values to be implemented specifically for that scalar. Go usually decodes JSON objects into structs for manipulation. However, it also offers an additional JSON type called *json.RawMessage*. This type is an alias for byte arrays and can be used to store JSON encoded strings. Therefore, when encoding, or marshalling, a string into JSON using the Go standard library, the result is of type json.Rawmessage.

With that in mind, the parsers and serializer for custom scalars are relatively easy to implement. In case of serialization the value, or a pointer to the value is returned if the type is either a json.RawMessage, or a pointer to one. For parsing, the value is cast to json.RawMessage, or a pointer to one.

GraphQL Arguments. The arguments used for each field is usually the unique ID that has been distributed by the orchestration system. However, arguments may also contain properties that refine queries. For example, a property that defines a graph's depth can be declared that can be later on used by resolvers to fine tune a graph representation.

GraphQL Resolvers. Each GraphQL type has its own resolver function. Additionally, each field of a type that represents a provenance relation has a resolver function too. The resolver functions query and return single database entries, and all their properties, by IDs and UIDs, thus a resolver for an activity object only returns one single activity object. While the IDs are arguments given to the resolvers, UIDs are given to nested resolvers by their parent resolver. When querying nested objects from GraphQL, the GraphQL server builds a chain of resolvers, which after execution connects the separate results to a single structure. As described in Section 2.4, the body of a GraphQL query contains the field and the field's properties that should be returned by the query. An implementation of a filter function to only return the requested fields is therefore not required as it is handled by the GraphQL server itself. The resolver used for the workaround query object simply returns the result of the database query.

5.7 Use Case Scenario Realization

In order to test and evaluate the provenance service and its capabilities, the tank import use case is implemented. The use case consists of two scenarios. The first scenario is used to prepare and import a feature-collection, and the second to correct features that have been rejected by the database during the import step. Each scenario consists of several steps or services. This section explains how these services are implemented.

Because of the missing system components, IDs and other required information is hardcoded into the messages that are consumed by the provenance service. While parts of the messages are hardcoded and mocked, the services used by the scenario are actually implemented. The input data for the scenario is provided by Microsoft's *USBuildingFootprints* GitHub repository[5] .

Scenario Trigger. In the fully implemented orchestration system, this scenario would be triggered by an event. This means that once the scenario has been orchestrated by the system, and therefore initialized, no user input is required to start the service chain. Instead, once an input file is present, the scenario would be notified and made aware of the files location. For the exemplary realization of this use case however, the services of each scenario is started manually. Due to the limitations the implemented services always have files as both in- and output. The file names used are unique IDs and are therefore suitable to be used as identification for provenance components.

[5] https://github.com/microsoft/USBuildingFootprints

Splitting the Feature Collection The first processing service is a simple shell script that splits a single feature-collection into multiple separated features. This step is necessary because it is the used import format by the tank database. Additionally, the geocoding service also supports the format, which is why this service is executed first. It utilizes MapBox's *tippecanoe-json-tool*[6] to split feature collections into separated features. To simulate a parallel workflow, the script is given a GeoJSON file as input and, after splitting the feature-collection, puts out multiple single feature files. The resulting files can then be further processed independently. The tool is executed as follows:

```
tippecanoe-json-tool INPUT_FILE > OUTPUT_FILE
```

Reverse Geocoding. The features contained in the feature-collection, and thus also the separated features, contain only the coordinates and the geometry of a given feature. Before the features can be stored, they therefore need to be given properties. This is achieved by reverse geocoding. Reverse geocoding is the process of assigning human readable properties, such as postal codes, street names, and more, to coordinates.

The geocoding service is implemented as a one-line shell script that passes the input files to an already existing NodeJS script, which sends request to a geocoding server and sets the properties of a GeoJSON feature accordingly. For the actual geocoding an internally hosted instance of *Nomantim*[7] is used. It then outputs a new file with the geocoded feature.

Import. The import service is also realized as a one-line shell script that calls the curl command. It accepts an input file that contains one or more seperated features and sends them via a POST request to the Tank database. The database then returns the internally distributed IDs. These IDs will be further used for identification and are thus also stored in the resulting output file. The curl command used is as follows:

```
curl -s -X POST -H "Content-Type: application/json" -d FEATURE
    TANK_URL > OUTPUT_FILE
```

Import Error Correction. Import errors are handled in a separate scenario. Like the import scenario, this scenario is started manually, whereas in the fully developed system it would be triggered by an event invoked by the completion of the import scenario.

In this scenario a one-line shell script polls the exhauster database for rejected features and stores them in an output file. The features downloaded are in the feature-collection format and hence need to split again by the same method as described earlier. After splitting, the features are then sanitized using a python script. In the script predetermined fields are corrected. Currently, the correction is purely exemplary and only

[6] https://github.com/mapbox/tippecanoe
[7] http://nominatim.org/

checks the postcode field. The postcode usually contains a string but needs to be converted to an integer during the database import. If the value contains anything but numbers, this is prevent and thus, in the exemplary script, anything that is not a number is removed from the field. If a rejected feature has a correct postcode field, it can be assumed that the problem is something else and is thus discarded to simplify the implementation. The remaining, sanitized features are deleted from the exhauster and lastly re-imported into the Tank database. In an advanced use case, this scenario would trigger itself again if features have been re-imported. The service implementation can be described with the following code:

```
regex = r"^([1-9]*?):"
with open(input_file_path) as input_file, open(output_file_path, 'a
    +') as output_file:
    feature = json.load(input_file)
    post_code = feature['properties']['postcode']
    matches = re.search(regex, post_code)
    if matches:
        feature['properties']['postcode'] = matches.group(1)
        json.dump(feature, output_file)
```

5.8 Deployment

The compiled executables of the provenance service and the API web server are very small. Additionally, since configuration is in theory handled by the orchestration system, no configuration files are required. As explained earlier, since the system is not available, the configuration required for further testing and evaluation is hardcoded instead. Due to the executable's sizes and the unproblematic configuration, the service as well as the API web server can easily be deployed in form of Docker images. Moreover, from "scratch" Docker images can be used, because the resulting executables contain Go's runtime and thus the base image does not have to contain a Go runtime or other requirements. This results in rather small Docker images.

Because Dgraph, RabbitMQ and the applications required for the tank use case provide Docker images, docker-compose can be used for deployment. Note, that this is only applicable for testing and evaluating the provenance service and its components. A full deployment of the entire orchestration ecosystem would require more complex tools such as *Ansible*[8], and would consist of multiple cloud computing nodes.

[8] https://www.ansible.com/

Chapter 6

Evaluation

In this chapter the concept, the implementation and the used technologies are evaluated. The evaluation is based on a simulation of the tank data import use case. The use case is simulated to avoid unecessarily long processing times and avoid workarounds that would be necessary because the orcehstration service is not yet implemented.

The evaluation is done with datasets containing $k = 10$, $k = 1000$ and $k = 10000$ features. All three datasets are subsets of the building footprints of Hawaii[1] ($k = 252891$ features). The tests were executed on a MacBook Pro 13" (2017 model) with 16GB memory a 3,1 GHz dual-core Intel Core i5. During the tests all required applications were run locally. This includes a RabbitMQ server, a Dgraph database, a web server to simulate HTTP requests to fetch information from system services, the provenance service and its corresponding API web server.

The graph visualizations are generated by Dgraph's management interface *Ratel*. Due to limited configurability of the admin interface, the color coding of the graphs could not be standardized. Furthermore, the name of some edges could not be hidden, which results in the first letters of the edge names appearing on some drawn edges.

6.1 Use Case

The tank import use case makes use of multiple services to prepare features before finally storing them in the tank database. The use case consists of two separate scenarios. The first scenario is the preparation and initial data import, the second one is the correction and reimport of rejected features. While the use case is divided into two scenarios, the resulting provenance is linked. This is because the initial inputs for the correction scenario are a subset of the output of the last step before the actual data import, the geocoding service. In this section the resulting provenance, its quality, the services performance and challenges related to the use case are discussed and evaluated.

Simulated Services. As pointed out earlier, the evaluation is based on a simulation of the scenarios instead of the execution results of the actual services. From a provenance

[1] https://github.com/microsoft/USBuildingFootprints

perspective, the use case can be realized in a single one-time executed scenario. This is possible because of how the data generated by the scenarios is connected. Furthermore, the reason the use case is split into two scenarios in the first place is that the correction scenario is allowed to run asynchronously. However, in order to simulate each service and their in- and outputs need to be considered.

Section 5.7 describes the services used by each scenario and their respective implementation. The services are file based, and thus instead of sending the results through the message broker, the results are collected and stored in single or multiple files. Services with multiple outputs generally indicate parallelisation, as one process of the following service is invoked per output. In a simulated setting the parallelisation can be realised without the need of creating multiple files.

The services are simply simulated by sending provenance messages that contain the IDs generated by the previous simulated service as input and new random generated IDs as output. This is possible because from a provenance perspective the data contained in the files can be completely ignored and only the IDs contained in the provenance messages are important as they are used to link the different provenance components. Additionally, during provenance generation the service does not differentiate between an activity that maps a single entity to a single new entity, or an activity that maps a single entity to multiple new entities or vice versa.

The only exception to the file based workflow is the actual import into the tank database and its fan-out to the exhauster. However, after inserting data into the database, the tank returns the internally assigned ID. If the scenario would not fan-out and thus result the import of a single feature at the end of each parallel process chain, a complete mapping of one feature to one entry in the tank database could not be guaranteed. Moreover, the tank also assigns IDs to rejected features, but does not return them. It is to note that currently the tank only provides the ID of rejected features in its logs. For the purpose of this simulation it is assumed that the IDs of rejected features are returned in the request. Which can be further used to connect the features contained in the feature-collection that is returned by the exhauster to the features prior to the actual tank import.

To simulate the fan-out to the exhauster, only 90% of the results of the geocoding step are imported. The remaining results are assumed to be incorrect and thus in the exhauster. They are later used during the exhauster scenario. It is assumed that not all incorrect features can be sanitized. Thus, further 50% of the remaining features are discarded before the simulation of the sanitization service in the exhauster scenario.

Even though the actual services are not used in the evaluation, exemplary results of each service that is simulated can be found in the appendix.

6.2 Provenance Generation.

Provenance generation is a very important aspect of the provenance service. To reduce the amount of database requests required to store provenance in the database, new

database entries are batched. An exemplary, for mutation prepared, provenance entity that is contained in a batch can be found in Listing A.1 in the appendix.

The following discusses the service's performance during the generation of provenance based on the simulated use case. It is to note that the simulated provenance messages are all send without any artificial delay. Hence, there is no delay in the provenance stream that could cause inconsistencies in the performance measurements.

First the simulation runtimes are evaluated. To do so simulations with dataset sizes of $k = 10$, with $n = 10$ executions, $k = 1000$, with $n = 10$ executions, and $k = 10000$, with $n = 5$ executions, are run. Due to the size of the last simulation, the database has been wiped before each of its five executions. Table 6.1 shows the average duration for the entire simulation, the average lifetime of a single batch within a simulation, as well as the amount of batches required to build the corresponding provenance graph and the amount of new database entries per simulation. A batch lifetime is defined as the time between its creation and its commitment to the database. The batch size for all tests is $j = 1000$.

Simulation Input Size	10	1000	10000
Batches per Simulation	1	4	33
Avg. Simulation Duration [s]	0.1763	7.0707	69.9893
Avg. Batch Lifetime [ms]	176.3931	1730.7859	2120.8885
New Database Entries per Simulation	33	3201	32001

Table 6.1: Average duration of simulations and the corresponding batches.

The results show that the average time it takes to completely generate provenance for a simulation directly correlates with the amount of batches the simulation has been divided into and their respective lifetimes. Furthermore, the total runtime of the $k = 10$ is equal to the time it took to complete the batch, because the entire simulation fits within a single one. The average batch lifetimes for $k = 1000$ and $k = 10000$ are roughly equal. It is to note that the fourth batch of the $k = 1000$ simulation only contains 201 provenance entities and is hence four-times faster than the previous three batches. While the $k = 10000$ simulation also contains an incomplete batch as its last batch, the total amount of batches is far higher. Thus, the incomplete batch has almost no impact on the average. When excluding the last batch from the measurements of the $k = 1000$ simulation, the average batch lifetimes are equal. The exact amount of data imported into the database is a result of the 10% and 50% feature reduction during the scenarios.

Batch Time Spread. The previous results show that batches are computed comparatively slow. Within the life time of a batch at least two HTTP requests are made to fetch the missing data, for example an agent's name, its type and a description in case of the agent being a service. Additionally, the Dgraph database is queried for every involved component. To avoid multiple queries on the same component, the query results are cached. However, the cache only exists on a per batch basis and is thus cleared after a batch is committed. The data fetched is then processed and used for a database

mutation. Table 6.2 shows the average ($n = 33$) time spread within a batch life time. The results are measured from the beginning to the end of each computation step.

The time spread shows that 99.6% of the time is spent at the database mutation. The time spent for computation and HTTP requests in insignificant and needs no further evaluation. This indicates a problem with the write-performance of Dgraph and will be further evaluated in Section 6.5.

Total	2092.0503ms
Database Mutation	2085.6917ms
HTTP Requests	0.2380ms

Table 6.2: Average time spread of processing times within a transaction batch ($n = 33$). Each batch contains $k = 1000$ provenance entities

6.3 Provenance Retrieval.

The provenance service is a composite service consisting of the generation service and a web server that serves a GraphQL API. The GraphQL queries are passed to the API as a string within the query parameter, for example: `localhost/graphql?query={entity(id:"ba75ac00-09ae-49d4-b20f-48e4ec03e75c"){uid,id,creationDate}}` To test the GraphQL API a simple query, Listing 6.1, that returns the IDs of all provenance components that are directly involved in the generation of a single new provenance entity is run against the GraphQL API. The query is run $n = 10$ times to calculate the average request time. The average will be used as reference in comparison with a GraphQL type that uses the same query but ignores GraphQL's resolve chains, and the same query run directly against the database. All queries have been executed after the $k = 1000$ simulation has been executed. Therefore at the time of each query 32001 database entries are present. Network delay is not considered in the evaluation.

```
{
    entity(id:"ba75ac00-09ae-49d4-b20f-48e4ec03e75c") {
        id,
        wasDerivedFrom {
            id
        },
        wasGeneratedBy {
            id,
            wasAssociatedWith {
                id,
                actedOnBehalfOf {
                    id
                }
            },
            used {
                id
            }
        }
    }
```

```
}
```

Listing 6.1: GraphQL Query to retrieve an entity and all provenance components that were directly involved in the creation of it.

The GraphQL type that ignores the GraphQL intended resolve chains uses the entire query from Listing 6.1 within a single resolver. It therefore cannot be customized and is limited to a single graph structure. While the structure can be used to represent all subgraphs that consist of a root entity and its directly related provenance components, other graph structure cannot be represented. As a result the query is merely a workaround to test GraphQL's performance impact. The workaround query can be called as follows:

```
{
    workaround(id:"ba75ac00-09ae-49d4-b20f-48e4ec03e75c") {
        json
    }
}
```

Table 6.3 shows the average request time for each query. The request time for the GraphQL queries are determined by measuring the time between the start and the end of the GraphQL execution directly on the server. To determine the duration of the query execution against Dgraph directly, Dgraph's debug information contained in the returned results are used. The request time averages indicate that using GraphQL the intended way has a relatively big performance impact with a factor of about 3.6 compared to the workaround query. The time difference between the request when using the query natively against the database and the workaround query is insignificant. This is most likely due to the fact that the workaround result is not parsed at all.

The performance impact observed is caused by GraphQL calling a resolver for every traversed edge (indicated by nested structures in the query). This results in a total of six executed queries for the reference query versus a single query against the graph database when using the workaround or querying directly against Dgraph. The impact grows more significant when the root query returns more than a single entity and is discussed further in Section 6.6.

Query	**Avg. Duration in ms**
GraphQL Intended	36.4082
GraphQL Workaround	10.4978
Against Database	9.8023

Table 6.3: Query performance for single entities and graph representations with different roots.

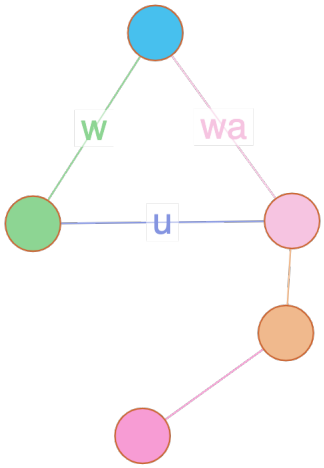

Figure 6.1: Visualization of the query result from Listing 6.1. The blue colored node is the root entity, the green node is the entity the root entity derived from. The generating process is colored in pink and the responsible parties in orange and magenta.

6.4 Provenance Quality.

The structural and contextual integrity are the two most important criteria when determining the quality of the generated provenance. Structural integrity is assessed by traversing the provenance graphs of a provenance component to its furthest ancestor. An analysis of the contextual integrity is omitted. This is due to contextual errors usually being a product of misbehaving process executions. In addition, due to the design of the use case, contextual integrity can be implied if a provenance graph is complete.

First the integrity of a provenance graph that corresponds to the splitting service is evaluated. To do so an entity generated by a process that has been invoked by the service is selected as the graph's root. While the activity that represents the process could be selected as root, the entity is chosen due to it being the most common starting point when traversing provenance trees. An exemplary use case for such a selection is as follows:

A dataset used by a process is found to be incorrect. To find the cause of its incorrectness a user traverses the provenance tree to the activity that has generated the entity. The process that correlates to the activity is assessed and a problem is found that caused incorrect data to be generated. To determine all incorrect datasets, all entities that have been generated by that process are traversed to. This example can be taken even further. Additionally, to the entities generated by the process, all processes and their resulting entities that have been invoked by the responsible service can be determined, as well as who is responsible for creating the service.

Figure 6.2 shows the resulting provenance graph for such a use case. Note that in order to traverse to the generated entities from an activity, the corresponding edge "wasGeneratedBy" needs to be indexed as a reverse edge to allow bidirectional graph

traversal. The resulting graph contains all provenance components that are involved in the generation of the root entity. In addition, entities that are created by the same process can also be found in the result. Therefore, the resulting graph can be considered complete.

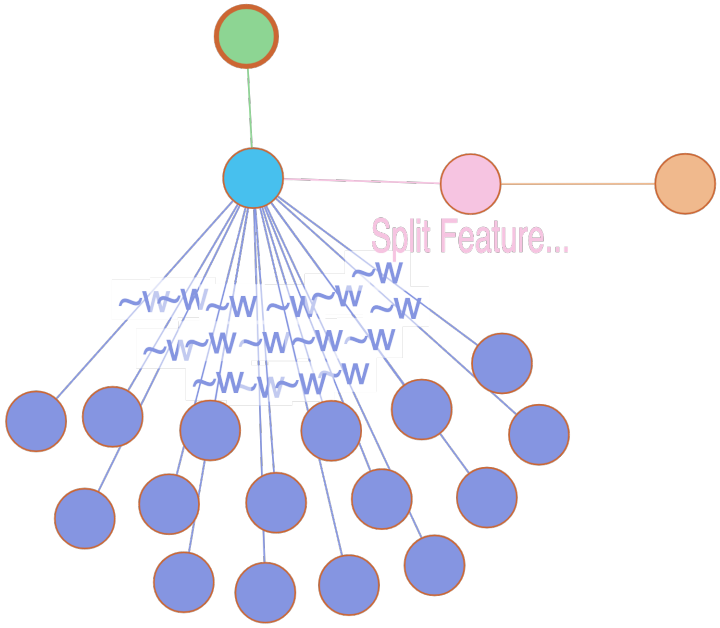

Figure 6.2: Graph containing the root entity, the activity that generated it and its responsible parties, as well as all other entities that are generated by the same activity. Color code: Root entity (green), activity(light blue), responsible service (pink), user responsible for service (orange), other by the activity created entities (dark blue).

The second graph is the result of a query for an use case in which the user wants to determine the whole ancestry chain of a feature that has been successfully imported by the tank import scenario. The root entity is the entity that correlates to an entry in the tank database.

Starting from the root, the query traverses the involved parties, including activities, and the responsible agents. The result, which is visualized in Figure 6.3, shows the complete provenance graph of a successfully imported feature. The graph was generated by using a recursive query which can be found in Appendix B. The resulting graph contains all edges and connects the nodes correctly and thus considered complete.

The next graph is the provenance graph for a feature that has been rejected by the tank database, corrected by the exhauster scenario and has lastly been reimported. To generate the graph another recursive query is used, which can also be found in the appendix. The graph, shown in Figure 6.4, is essentially the union of two sub-graphs. The first sub-graph (blue border) is an incomplete graph of the tank-import scenario. It is considered incomplete, because the last service of the tank scenario, the

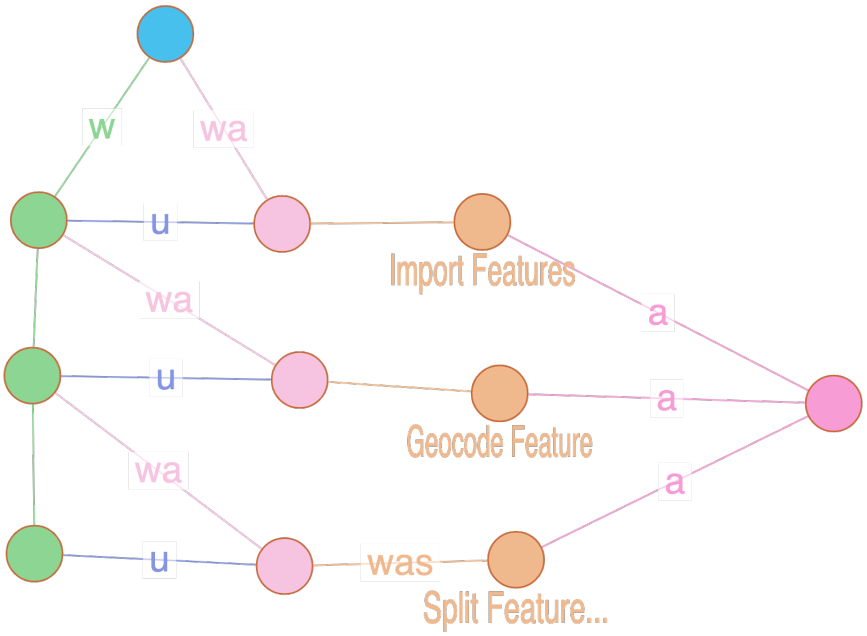

Figure 6.3: Provenance graph of an imported feature. Color code: Root entity (blue), ancestor entities (green), generating process (pink), responsible service (orange), user responsible for the services (magenta)

import is missing. The second sub-graph (green border) is the complete graph of the exhauster scenario. As a result, the graph contains provenance information about two independently running, but through data flow connected, scenarios.

Even though the tank-import sub-graph is incomplete, the resulting provenance graph is still structural complete and correct, because the graph is the exact representation of the events shared by the processes that are invoked by the services of each scenario. This can be further proven by combining both queries. The resulting graph, see Figure A.1, is connected graph of rejected graph and the graph of the successfully imported feature.

Missing Provenance Information. The problem that some information is missing, is caused by the way the simulated services handle these situations. The provenance service is designed as a passive service. As such it is not aware of features that simply go missing due to a process not sharing information on, for example, rejected imports. Another situation which results in an incomplete provenance graph is the feature sanitization. Before rejected features can be corrected, they first need to be split into single features again, because the exhauster provides feature collections instead of single features. Assuming the provenance chain starts at the feature-collection, no dataset can go missing yet, this however changes during sanitization. The automated sanitization script assumes that, for example, a given field is in incorrectly formatted. If the field tested by the script is correct, it can be assumed that the problem was something dif-

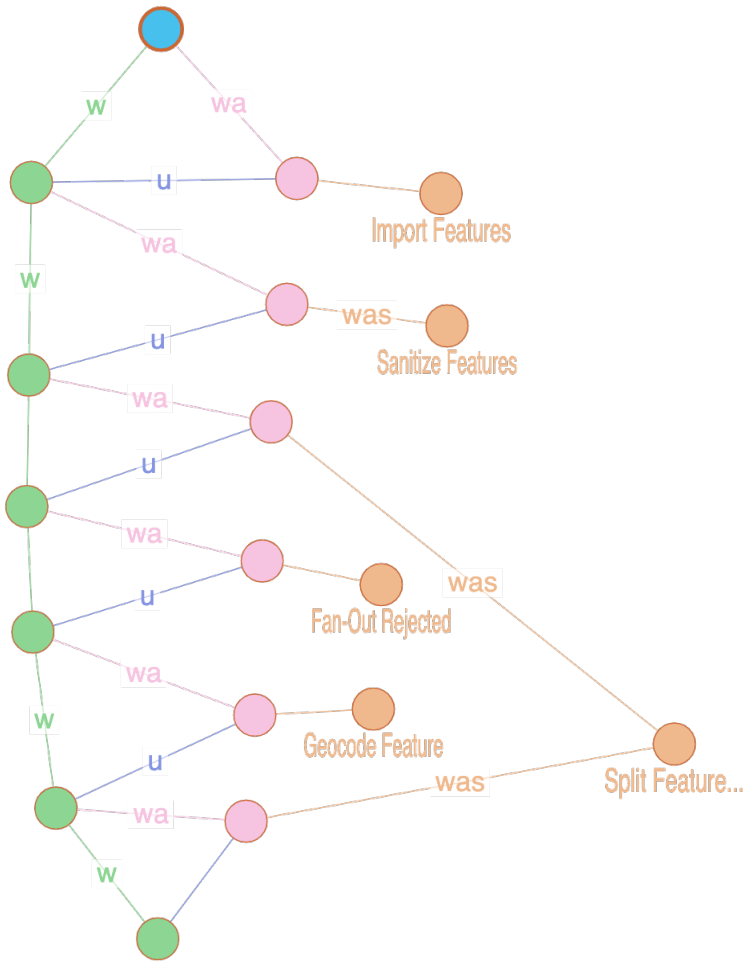

Figure 6.4: Provenance graph of a corrected feature. Color code: Root entity (blue), ancestor entities (green), generating process (pink), responsible service (orange)

ferent. The feature is therefore discarded and remains in the exhauster database, where it can be corrected manually.

Possible Solutions. Currently the provenance service is unable to automatically identify and solve the issue at hand. A possible solution is to expand provenance messages and solve the problem on the service level. To do so, a new field *success* is introduced, which indicates the outcome of the process that did or did not generate a new provenance entity. If a process generated a new entity successfully, the provenance message is analogue to the current model. In the other scenario, that an entity was not generated successfully, a provenance message with both in- and output IDs would still be sent and thus provenance could be generated. However, the new field would indicate that the process responsible for that entity was not successful. The information on the

outcome could be stored within the "wasGeneratedBy" edge. The resulting provenance graphs are always complete and contain information even on provenance entities that would normally not be recorded. Using Figure 6.4 as an example, the missing information would be inserted between the activities that correlate to the service "Geocode Feature" and "Fan-Out Rejected". While this solution generates better provenance for those cases, it requires more overhead from the user responsible for the service creation.

Another solution is therefore to make the provenance service aware of the expected outcome of each process. This would also require more user overhead, but instead of changing the service's execution or implementing the new message format, the information on what is expected can be contained in the service description. To do so, the service description is expanded, including in- and output cardinality. Using the sanitization service as an example, a process invoked by that service expects one input file and generates exactly one output file. It can be assumed that a new entity could not be generated, if no output information is provided by the process. In that case a placeholder entity is created by the provenance service, that makes use of the proposal above to store this information in an edge.

While the second proposal works on all services that generate exactly one output, even if multiple inputs are provided, it won't work for services that have an undefined amount of outputs. For that matter, further solutions need to be tested and evaluated. This, however, is subject to future work.

6.5 Database Performance

The use-case evaluation in the earlier section shows that Dgraph's query performance is more than suitable for relatively big provenance representations. However, the write performance is clearly a bottleneck. In this section the cause of the problem as well as potential solutions are discussed.

Concurrency. Many databases allow concurrent read and write operations. Dgraph does, for the most part, not. When two database transactions are executed concurrently, there are two cases that need to be considered. The first and best case, shown in Listing 6.2, is that both transactions write to independent nodes using the "wasDerivedFrom" edge. If that is the case, both transactions succeed and the database has been written to concurrently.

```
// first mutation
{
    "uid": "_:b4c09cf6-0b22-4de6-b8da-19a7c8158061",
    "id": "b4c09cf6-0b22-4de6-b8da-19a7c8158061"
    "creationDate": "2020-01-16 02:08:56.837612 +0100 CET m
        =+0.019593012",
    "wasDerivedFrom": [
        {
            "uid": "0x123",
        }
    ]
```

```
}

// second mutation
{
    "uid": "_:ea46abad-0f34-4aac-be6e-197891d79e3d",
    "id": "ea46abad-0f34-4aac-be6e-197891d79e3d",
    "creationDate": "2020-01-16 02:08:56.837612 +0100 CET m
       =+0.019593012",
    "wasDerivedFrom": [
        {
            "uid": "0x456",
        }
    ]
}
```

Listing 6.2: Exemplary mutations contained in concurrent Dgraph transactions, that would result in successful concurrent writes.

However, this cannot be guaranteed. If two or more transactions try to write to the same node, all but one transaction are rejected. The rejected transactions can just be executed again, however, this could lead to incomplete provenance graphs. This is due to the way Dgraph creates edges. Within a batch, all connected blank nodes have been assigned by the preparation steps. A rejected database transaction might include provenance components that are now available in the database, but due to them missing before the transaction was executed, they are still referenced by a blank node instead of a Dgraph assigned UID. Therefore, all entities within a rejected transaction are required to redo all preparation steps that fetch related provenance components. Hence, concurrent write-transactions need to be avoided.

Potential Solutions. Before solutions can be discussed, the environment in which the provenance service is deployed needs to be considered. As part of one of the system services, the provenance service needs to guarantee that all dedicated messages sent can be handled. However, it does not need to guarantee that a provenance graph for a scenario is available the moment it concludes. Therefore, the actual database transactions can be done asynchronously and independent of the actual message handling.

Currently the relatively slow writing speed could lead to problems. The performance measurements show that more than 99.6% of the time is spent on the database transaction. This could potentially cause a considerable backlog of provenance messages, that cannot be persisted in a timely manner. In a worst case scenario this could lead to a dataloss. Thus, provenance messages need to be prepared and persisted as fast as possible.

An easy solution is to use a different database that supports concurrent writes. While this solves the write performance problem, query times become considerably longer depending on the replacement database. A potential replacement database is Neo4j[2]. Another advantage of keeping Dgraph as the storage solution is its easy scalability and deployability in distributed environments.

[2] https://neo4j.com/

The second solution is to use a centralized persistent cache as an auxiliary storage, in which prepared entities are stored and steadily transferred to the Dgraph database. In its current implementation, the provenance services handles provenance messages sequentially by a single routine. Provenance entities are prepared by requesting data from the database, a system service, and the key-value cache used to store the blank nodes, before they are added to a batch. A centralized cache allows messages to be prepared concurrently, as they are not directly stored in the actual database at the end of the preparation steps. Instead of a single routine that prepares provenance entities based on the consumed messages, one goroutine can be invoked per message. The prepared entities created by each goroutine are then stored in the centralized cache instead of the database. To avoid concurrent writes, which might fail, a single goroutine collects the prepared entities from the cache and commits them to the database. To do so, the cache needs to act like a queue, the first entity that is written to the cache needs to be the first one to be read from the cache.

The provenance service does not need to guarantee that provenance is available as soon as a scenario is finished, but it needs to guarantee that a complete provenance graph is generated. The solutions proposed can solve the problem, however they both need to be thoroughly tested and evaluated.

6.6 GraphQL API

GraphQL is a great tool and depending on the use case a superior replacement to REST APIs. Although the provenance generated can be queried using the GraphQL API, it is not without its downsides or workarounds. GraphQL excels when the user can define the structure of the data. This is easily achieved for retrieving single provenance components. Retrieving graph representations, however, is a bigger problem. While the implemented API offers this functionality, it is simply a workaround.

Essentially in its current implementation, a user has two ways to retrieve a graph representation. The first one is simply using the workaround root query and the other is building its own structure starting at an activity, agent or entity root query. When using the workaround root query, a query, what is essentially a static, predefined query, is run against the Dgraph database and its output is simply passed through the GraphQL server to the client. This makes best use of the benefits of a graph database, but GraphQL's biggest selling point, its customisable query structure, is being discarded. Although the second option to query data is the preferred way, it greatly sacrifices performance for customisability.

Independent of the database used when resolving a nested GraphQL query, each resolver would generate a new database request. The problem becomes even more severe when working with lists. This is due to how resolvers get passed down in GraphQL. When a top level resolver results in a list of objects, the lower level resolvers are not made aware of the list. Instead a new resolver is invoked for each item in the list. This is known as the *N+1 Problem*.

N+1 Problem. The N+1 problem is a problem unique to GraphQL and is caused by its resolver concept and can lead to poor query performance independent of the chosen backend database. The query in Figure 6.5 returns a list with all activities and their respective fields *id* and *description*, and the entities used and their field *id*, by each activity, that have been associated with the agent with the ID *agent1*.

```
{
    query{
        activity(WasAssociatedWith.id="agent1") {
            id
            description
            used {
                id
            }
        }
    }
}
```

Listing 6.3: Query to fetch all activities and their used entities, that are associated with a certain agent.

In Dgraph, or any graph database for that matter, this would require a single database round-trip. However, due to the N+1 problem, this would result in several full round trips; one for all activities and one more for each activity, hence $n+1$ queries (amount of activities returned by the root query plus the root query). Listing 6.4 shows the chain of queries that would be executed if the query returned two activities with the respective IDs *activity1* and *activity2*. Figure 6.5 shows the corresponding pseudo graph. It is color coded to indicate the different queries generated by GraphQL.

```
SELECT *
FROM activities
WHERE agent_id = "agent1"

SELECT *
FROM entities
WHERE wasgeneratedby_id = "activity1"

SELECT *
FROM entities
WHERE wasgeneratedby_id = "activity2"
```

Listing 6.4: Pseudo queries to fetch all activities and their used entities that are associated with a certain agent.

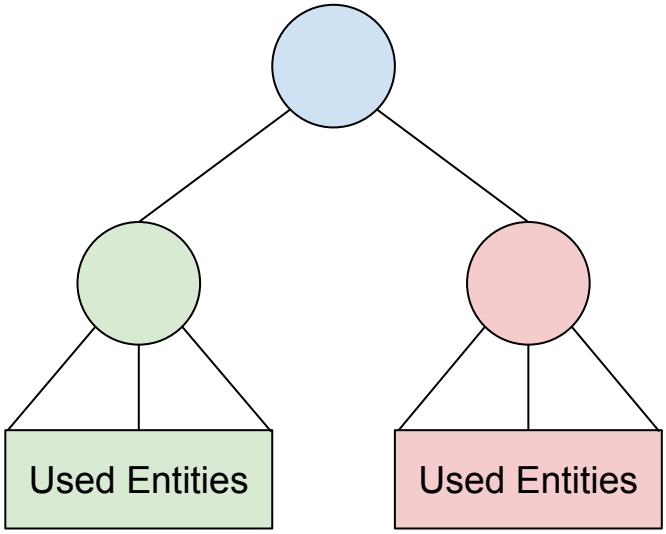

Figure 6.5: Graph representation of Listing 6.3

Solutions. The most common approach to the N+1 problem is to use Facebook's DataLoader[3], or community developed implementations for other languages. The basic principle of the DataLoader is to batch and cache resolver executions. As a result, the amount of full database run trips can be drastically reduced. While this is a big improvement for relational databases, graph databases still would suffer from performance hits compared to native queries to the database, because they could usually resolve the GraphQL query in a single database request.

Recently graph database developers started to acknowledge GraphQL's native way of querying data. Neo4j released a GraphQL plugin[4] that seeks to allow full integration of the GraphQL standard to applications that want to make use of both Neo4j and GraphQL. Dgraph uses its own derivative of GraphQL, GraphQL+. While it is based on the standard, it is not compatible and thus requires workarounds such as the one described earlier. This has been a big talking point in the community and thus has led to the Dgraph team starting its development on their own implementation of the full GraphQL API server that can be integrated into the Dgraph cluster. Their efforts are currently available in beta[5].

The technologies by Dgraph and Neo4j offer a solution to the problem but are specific to their respective database. Moreover, the Dgraph approach does not allow developers to implement their own API, but instead forces usage of their own server. While this should be suitable for most use cases, there are still situations where a custom implementation is preferred. Whether the data loader pattern, or a first party solution offered by the database developers is better in the specific use case of a provenance service for distributed system requires more testing and cannot be answered at the

[3] https://github.com/graphql/dataloader
[4] https://neo4j.com/developer/graphql/
[5] https://graphql.dgraph.io/

given time. However, a solution, in which the full benefits of a graph database can be combined with the capabilities of a GraphQL server, is preferred.

6.7 Generating Provenance without Dedicated Messages

For the realisation of the tank-import use case dedicated provenance messages have been used. Provenance messages are generally the preferred way to generate provenance for a scenario. However, there may be cases in which a service and therefore its invoked processes cannot provide dedicated provenance messages, even though the user that created the service wishes that provenance is generated. This is, from a provenance perspective, the most challenging situation. In this section different ways to generate provenance without dedicated provenance messages are discussed.

Retroactive Provenance Generation. Currently the service generates provenance by combining the information on a process, contained in dedicated provenance messages, with information queried from system services of the orchestration system. The IDs required for the queries are extracted from a message's routing key. As per specification, a message send by a process must contain the ID of the process and thus the ID can also be used for the extraction of process information. This is possible due to the orchestration system's concept. The main purpose of the system is to manage and execute scenarios. In order to execute the services, that make up the scenarios, processes need to be invoked. Furthermore, the processes need to exchange in- and output data, while at the same time not being aware of each other. Thus, the data exchanged must be handled by the orchestration system and hence the orchestration system must also be aware of process related data.

The information on processes within the system service is essentially the same information contained in provenance messages. The key difference is that the way the information is handed to the system services by a process can vary. For instance, the script used to split feature-collections for the tank import use case is an example for a file based service. The script is called with input and output paths as arguments, which therefore have to be known beforehand. Note that the output paths can be paths to directories or files. It can be assumed that those are assigned by the orchestration service. Therefore the input path, the output path and the exact time the process has been invoked are known by and available to system services and the remaining information, for example the time of completion, possible errors or execution reports, is added upon completion.


```
{
    "id": "2c2755df-ca17-4a01-a4a5-bfd91b3e18f8",
    "instanceOf": "125caf57-2221-4a11-a66c-f01a7eba5ae1",
    "startDate": "2020-01-16 02:08:56.837612 +0100 CET m
        =+0.019593012",
    "endDate": "2020-01-16 02:08:56.837612 +0100 CET m=+0.019593012",
    "input": "b4c09cf6-0b22-4de6-b8da-19a7c8158061"
    "output": "ea46abad-0f34-4aac-be6e-197891d79e3d",
```

```
    "status": "complete",
}
```

Listing 6.5: Exemplary process information used for retroactive provenance generation.

In order to combine multiple messages to a single provenance message, the provenance service would need to store the message fragments and track the execution state of processes to ensure full message integrity. However, this can be avoided. It can be assumed that at the time a process' last message fragment has been consumed, all information on that process is available to the orchestration system, including file paths, timestamps and possible state reports. Thus instead of tracking and building full messages on its own, the provenance service only needs to wait for messages that indicate a process has been completed. After such a message is consumed, the information that would usually be contained in provenance messages can be queried from system services by the ID contained in the routing key.

Log-Messages. Provenance generated from provenance messages and retroactively from system service share the assumption that all data required for a complete provenance presentation of a process and its in- and outputs is made available by a process. However, it is possible that this is not the case. For example a process invoked by a service designed to import a batch of data into a database might not have an actual output relevant to the scenario and because outputs are not mandatory this information can be omitted. This would result in incomplete provenance information, because even if detailed provenance for each object in the batch is available before the import, after the import the entire batch would be linked to the same derivative entity, namely the entire database.

Processes like this could still send out log messages. The log messages can be filtered for messages that can be used for provenance using regular expressions. This however implies that the provenance service is aware of the log message format used by a service. Using the tank-import service as an example, a process invoked by the service could publish out a log message, as seen in Listing 6.6, once a feature has been successfully imported. Depending on the service definition, log messages can be handled like full provenance messages.

```
{
    "timestamp": "2020-01-16 12:00:00",
    "message": "feature imported, id: 125caf57-2221-4a11-a66c-
        f01a7eba5ae1",
}
```

Listing 6.6: Log message sent by a process on successful database import.

In some cases a combination of a log-message based and a retroactive provenance generation may be required. The listing above only contains the ID of the feature within the tank database. Therefore, in order to generate complete provenance, process information needs to be extracted using the retroactive method described earlier. The extracted information then needs to be combined with the content of the log-message. Even if the orchestration system is unaware of all objects that have been created by a

process, it must be aware of the input dataset given to a process. This is implied by the orchestration system's design.

Mixed Message Scenarios. Provenance messages are sent by a process and thus the type of message that is supposed to be used for provenance is defined by the invoking service. As a result a scenario could potentially make use of dedicated provenance messages, log messages and retroactive provenance generation within a single scenario execution. While challenging, it is certainly possible to generate provenance in a scenario like this. However, in order to produce meaningful provenance, the service needs to be aware of the different provenance generation method, that needs to be used per service. Otherwise, provenance information may be lost, which results in an incomplete graph. This can easily be achieved by introducing two new fields to the service definition. The first field *provenanceType* is used to define the method for provenance generation and the second field *provenancePattern* is used to filter provenance messages if the defined generation method is "log message".

Providing full provenance coverage might not always be possible. However, combining dedicated provenance messages and the concepts introduced in this section should cover most scenarios. The realisation and evaluation of the concepts is subject to future work.

Chapter 7

Conclusion

This thesis focused on the concept and the implementation of a provenance service for an orchestrated distributed workflow management system. To determine a suitable data model, as well as the general requirements, for the service, different provenance use cases and implementations have been investigated. The investigation has shown that provenance is commonly used in various environments. The W3C proposed PROV-DM is commonly a used data model for geospatial data. Moreover, the investigation shows that a PROV based data model can easily be expanded to fit most requirements.

To determine the requirements for the provenance service the specifications of the target system have been reviewed. Based on the requirements a general concept and application architecture has been developed, which has been implemented as a prototypical provenance service.

The implementation has been evaluated using a simulated scenario based on a real world use case. While the evaluation has proven that the provenance service's concept is able to generate provenance for the simulated scenario, limitations of the current concept could also be observed. The main limitation has been Dgraph's write performance. Even with batched mutations, the database's write speed has been relatively slow. The evaluation has also shown that the GraphQL query language is a potent tool when working with provenance graphs. However, the GraphQL server itself has had a negativ performance impact when working with a graph database. Potential solutions to both problems have been proposed and discussed.

In addition to storage performance, the provenance graphs generated by the service have been evaluated. The evaluation of the graphs has shown that the service is able to create graphs independent of the executed scenarios. It also has also shown that the quality of the provenance graphs is in a direct relation to the amount of data sent to the provenance service. This became apparent in the provenance graphs generated by the scenario responsible for correcting rejected features as the resulting graphs did not contain the activity responsible for the import. This was the result of the service not providing provenance messages for rejected features. While it is not the responsibility of the provenance service to insert data that is assumed to be missing, potential ways to indicate failed process executions have been proposed.

Finally, the prototype is only able to generate provenance by consuming dedicated provenance messages. While this is the easiest and probably most efficient way to generate provenance, other creation methods have been introduced.

In conclusion, it can be said that the provenance service developed in this thesis is a suitable foundation to expand on. The limitations that have been observed do not indicate problems with the concept, because they, for the most part, can be directly associated with the chosen technologies. The provenance solution developed in this thesis should be seen as the first steps to complete provenance solution and can be used for future developments.

Appendix A

Provenance Representations

A.1 Dgraph Mutation


```
{
  "uid": "_:ea46abad-0f34-4aac-be6e-197891d79e3d",
  "id": "ea46abad-0f34-4aac-be6e-197891d79e3d",
  "creationDate": "2020-01-16 02:08:56.837612 +0100 CET m
     =+0.019593012",
  "wasDerivedFrom": [
      {
          "uid": "_:b4c09cf6-0b22-4de6-b8da-19a7c8158061",
          "id": "b4c09cf6-0b22-4de6-b8da-19a7c8158061"
      }
  ],
  "wasGeneratedBy": [
      {
          "uid": "_:22c2b109-31bf-419c-a26e-6d14e03ef4c5",
          "id": "22c2b109-31bf-419c-a26e-6d14e03ef4c5",
          "startDate": "2020-01-16 02:08:56.837423 +0100 CET m
             =+0.019403581",
          "endDate": "2020-01-16 02:08:56.837612 +0100 CET m
             =+0.019593012",
          "wasAssociatedWith": [
              {
                  "uid": "_:125caf57-2221-4a11-a66c-f01a7eba5ae1",
                  "id": "125caf57-2221-4a11-a66c-f01a7eba5ae1",
                  "name": "Split Feature-Collection",
                  "description": "Splits a Feature Collection into
                     single features",
                  "type": "service",
                  "actedOnBehalfOf": [
                      {
                          "uid": "_:2c2755df-ca17-4a01-a4a5-
                             bfd91b3e18f8",
                          "id": "2c2755df-ca17-4a01-a4a5-bfd91b3e18f8",
                          "name": "Max Mustermann",
                          "type": "user"
                      }
```

```
                    ]
                }
            ],
            "used": [
                {
                    "uid": "_:b4c09cf6-0b22-4de6-b8da-19a7c8158061",
                    "id": "b4c09cf6-0b22-4de6-b8da-19a7c8158061"
                }
            ]
        }
    ]
}
```

Listing A.1: Prepared database mutation for a new entity and all in its creation involved parties.

A.2 Combined Provenance Graph

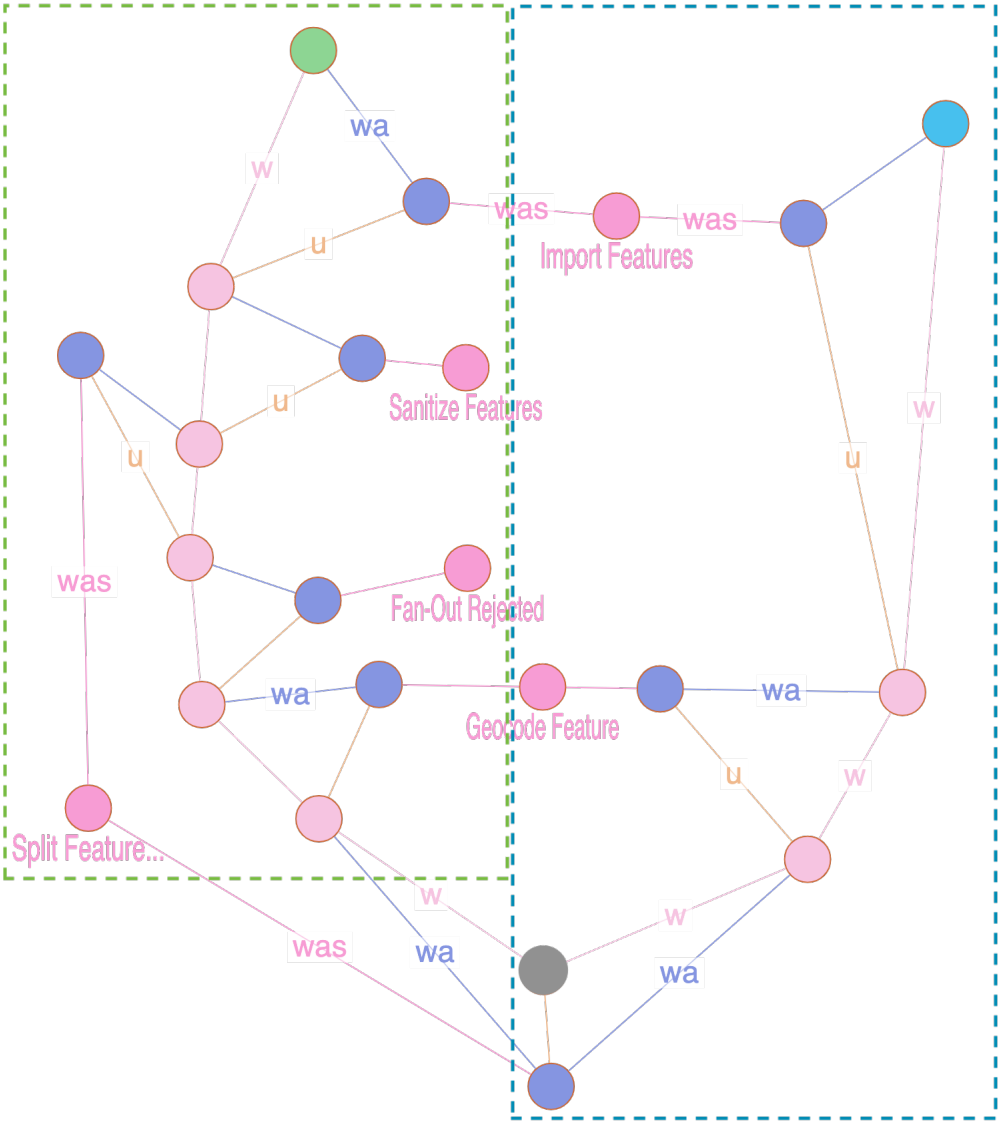

Figure A.1: Combined graph of the complete processing trees of a rejected feature (green border) and a successfully imported feature (blue border). The sub-graphs can be found in the Figures 6.3 and 6.4. The color coding is as follows: rejected root (green), imported root (blue), ancestor entities (pink), involved activites (blue), responsible services (magenta), origin entity (grey).

Appendix B

Dgraph

B.1 Queries

```
{
    rejected(func: eq(id, "93dc9f1e-f656-4796-acb1-9a9ca39e6b4b"))
        @recurse(depth: 10, loop:true) {
        id
        creationDate
        startDate
        endDate
        name
        wasDerivedFrom
        wasGeneratedBy
        used
        wasAssociatedWith
    }
}
```

Listing B.1: Recursive query to generate full provenance graph of a rejected entity.

```
{
    imported(func: eq(id, "cb0b735b-db57-4ef6-b76f-37b471c1988d"))
        @recurse(depth: 10, loop:true) {
        id
        creationDate
        startDate
        endDate
        name
        wasDerivedFrom
        wasGeneratedBy
        used
        wasAssociatedWith
    }
}
```

Listing B.2: Recursive query to generate full provenance graph of a successfully imported entity.

```
{
    query(func: eq(id, "05e46b01-7b16-4c9d-b567-b1791bf82814")) {
        id
        creationDate
        wasDerivedFrom {
            id
        }
        wasGeneratedBy {
            id
            startDate
            endDate
            used {
                id
                creationDate
            }
            wasAssociatedWith {
                id
                name
                actedOnBehalfOf {
                    id
                    name
                }
            }
        }
    }
}
```

Listing B.3: Query to generate a provenance graph of an entity and all parties that were involved in its creation.

B.2 Schema

```
<actedOnBehalfOf>: uid @reverse .
<creationDate>: string .
<description>: string .
<endDate>: string .
<id>: string @index(exact) @upsert .
<name>: string .
<startDate>: string .
<type>: string .
<uri>: string .
<used>: uid @reverse .
<wasAssociatedWith>: uid @reverse .
<wasDerivedFrom>: uid @reverse .
<wasGeneratedBy>: uid @reverse .
```

Listing B.4: Dgraph database schema.

Appendix C

Tank Import Use-Case Results


```
{
    "type":"FeatureCollection",
    "features":
    [
        {
            "type":"Feature",
            "geometry":{
                "type":"Polygon",
                "coordinates":[
                    [
                        [-155.773005,19.108122],
                        [-155.772896,19.108141],
                        [-155.772916,19.108241],
                        [-155.773025,19.108222],
                        [-155.773005,19.108122]
                    ]
                ]
            },
            "properties":{}
        }
    ]
}
```


Listing C.1: Input feature-collection.


```
{
    "type":"Feature",
    "geometry":{
        "type":
        "Polygon",
        "coordinates":[
            [
                [-155.773005,19.108122],
                [-155.772896,19.108141],
                [-155.772916,19.108241],
                [-155.773025,19.108222],
                [-155.773005,19.108122]
            ]
```

```
        ]
    },
    "properties":{}
}
```

Listing C.2: Split feature returned by the splitting service.

```
{
    "type":"Feature",
    "geometry":{
        "type":
        "Polygon",
        "coordinates":[
            [
                [-155.773005,19.108122],
                [-155.772896,19.108141],
                [-155.772916,19.108241],
                [-155.773025,19.108222],
                [-155.773005,19.108122]
            ]
        ]
    },
    "properties":{
        "id":45621189,
        "postcode":"96737",
        "county":"Hawaii County",
        "road":"Aloha Boulevard",
        "area":132.08039884712187,
        "lat":19.1081815,
        "lon":-155.7729605
    }
}
```

Listing C.3: Geocoded feature generated by the geocoding service.

```
{"msg": "feature imported", "id": "2ef9a0ac-651c-4330-80dc-da44683c194d"}
```

Listing C.4: Response by the tank upon a successful import.

```
Fan out to Exhauster - {"status": "accepted", "id": "5
    e215086060b8f5e95ff5628"}
```

Listing C.5: Log message produced by the tank upon feature rejection.

Appendix D

Docker

D.1 Dockerfile

```
FROM golang:alpine AS build

RUN apk update && apk add --no-cache git ca-certificates

WORKDIR /tracer

COPY ./go.mod ./go.sum ./
RUN GOPROXY=https://proxy.golang.org go mod download

COPY ./ ./

RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 \
        go build -ldflags="-w -s" \
        -installsuffix "static" \
        -o /go/bin/tracer /tracer/cmd/tracer

FROM alpine:latest

ENV DEPLOYMENT_ENVIRONMENT "PROD"
ENV DATABASE_URL ""
ENV BROKER_URL ""
ENV BROKER_USER ""
ENV BROKER_PASSWORD ""
ENV BATCH_SIZE_LIMIT ""
ENV BATCH_TIMEOUT ""

COPY --from=build /go/bin/tracer /go/bin/tracer

CMD "/go/bin/tracer"
```

D.2 Docker-Compose

```
version: "3.2"
services:
  zero:
```

```
    image: dgraph/dgraph:v1.0.18
    volumes:
      - /tmp/data:/dgraph
    ports:
      - 5080:5080
      - 6080:6080
    restart: on-failure
    command: dgraph zero --my=zero:5080

  alpha:
    image: dgraph/dgraph:v1.0.18
    volumes:
      - /tmp/data:/dgraph
    ports:
      - 8080:8080
      - 9080:9080
    restart: on-failure
    command: dgraph alpha --my=alpha:7080 --lru_mb=2048 --zero=zero:5080

  ratel:
    image: dgraph/dgraph:v1.0.18
    ports:
      - 8000:8000
    command: dgraph-ratel

  rabbitmq:
    image: "rabbitmq:3-management"
    ports:
      - "5672:5672"
      - "15672:15672"
    volumes:
      - "rabbitmq_data:/data"

volumes:
  rabbitmq_data:
```

Acronyms

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CRUD	Create, read, update and delete
HTTP	Hyper Text Transfer Protocol
IoT	Internet of Things
LDAP	Lightweight Directory Access Protocol
M2M	Machine to Machine
OPM	Open Provenance Model
RDF	Resource Description Framework
REST	Representational State Transfer
SPADE	Support for Provenance Auditing in Distributed Environments
W3C	Word Wide Web Consortium

Bibliography

[1] ISO/IEC 19464:2014 information technology — Advanced Message Queuing Protocol (AMQP) v1.0 specification. Technical report, May 2014. (page 7)

[2] AMQP Working Group 0-9-1. AMQP advanced message queuing protocol; version 0-9-1. Technical report, November 2008. (page 8)

[3] S. Batra and C. Tyagi. Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2):509–512, may 2012. (page 11)

[4] M. Bryant. Graphql for archival metadata: An overview of the ehri graphql api. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2225–2230, Dec 2017. doi:10.1109/BigData.2017.8258173. (page 18)

[5] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub. The GeoJSON Format. RFC 7946, RFC Editor, August 2016. URL: https://tools.ietf.org/rfc/rfc7946.txt. (page 2)

[6] Y. Cheah and B. Plale. Provenance analysis: Towards quality provenance. In *2012 IEEE 8th International Conference on E-Science*, pages 1–8, Oct 2012. doi:10.1109/eScience.2012.6404480. (page 18)

[7] G. Closa, J. Masó, B. Proß, and X. Pons. W3c prov to describe provenance at the dataset, feature and attribute levels in a distributed environment. *Computers, Environment and Urban Systems*, 64:103 – 117, July 2017. URL: http://www.sciencedirect.com/science/article/pii/S0198971517300558, doi:https://doi.org/10.1016/j.compenvurbsys.2017.01.008. (pages 16 and 17)

[8] L. Di, P. Yue, H. K. Ramapriyan, and R. L. King. Geoscience data provenance: An overview. *IEEE Transactions on Geoscience and Remote Sensing*, 51(11):5065–5072, Nov 2013. doi:10.1109/TGRS.2013.2242478. (page 17)

[9] Facebook. Graphql specification. Technical report, June 2018. URL: https://graphql.github.io/graphql-spec/June2018/. (page 9)

[10] D. Garijo, Y. Gil, and A. Harth. Challenges in modeling geospatial provenance. In *Proceedings of the fifth international provenance and annotation Workshop (IPAW), Cologne, Germany, June 9*, volume 13, page 2014, June 2014. (pages 16 and 17)

[11] A. Gehani and D. Tariq. Spade: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*, pages 101–120. Springer-Verlag New York, Inc., 2012. (pages 15 and 16)

[12] T. Malik, L. Nistor, and A. Gehani. Tracking and sketching distributed data provenance. In *2010 IEEE Sixth International Conference on e-Science*, pages 190–197, Dec 2010. doi:10.1109/eScience.2010.51. (page 15)

[13] L. Moreau et al. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, June 2011. URL: https://eprints.soton.ac.uk/271449/. (page 6)

[14] L. Moreau and P. Missier. PROV-DM: The PROV Data Model. W3C recommendation, W3C, April 2013. URL: http://www.w3.org/TR/2013/REC-prov-dm-20130430/. (pages 1 and 7)

[15] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I.Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56, 2006. (page 16)

[16] N. Naik. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, Oct 2017. doi:10.1109/SysEng.2017.8088251. (page 8)

[17] F. Nogatz and D. Seipel. Implementing graphql as a query language for deductive databases in swi-prolog using dcgs, quasi quotations, and dicts. *Electronic Proceedings in Theoretical Computer Science*, 234:42–56, Jan 2017. URL: http://dx.doi.org/10.4204/EPTCS.234.4, doi:10.4204/eptcs.234.4. (page 9)

[18] S. S. Sahoo, A. Sheth, and C. Henson. Semantic provenance for escience: Managing the deluge of scientific data. *IEEE Internet Computing*, 12(4):46–54, July 2008. doi:10.1109/MIC.2008.86. (page 16)

[19] M. van Steen and A. S. Tanenbaum. *Distributed Systems*. Maarten van Steen, third edition, December 2018. (page 5)

[20] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, New York, NY, USA, april 2010. Association for Computing Machinery. doi:10.1145/1900008.1900067. (page 17)

[21] P. Yue and Lianlian He. Geospatial data provenance in cyberinfrastructure. In *2009 17th International Conference on Geoinformatics*, pages 1–4, Aug 2009. doi:10.1109/GEOINFORMATICS.2009.5293509. (page 17)

[22] D. Zhao, C. Shou, T. Maliky, and I. Raicu. Distributed data provenance for large-scale data-intensive computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, Sep. 2013. doi:10.1109/CLUSTER.2013.6702685. (page 16)