Migrating Microservices to Graph Database

Master of Science Thesis
University of Turku
Department of Computing
Software Engineering
2021
Tuomo Virolainen

Supervisor:
MSc. (Tech) Sampsa Rauti

UNIVERSITY OF TURKU
Department of Computing

TUOMO VIROLAINEN: Migrating Microservices to Graph Database

Master of Science Thesis, 82 p., 7 app. p.
Software Engineering
February 2021

Microservice architecture is a popular approach to structuring web backend services. Another emerging trend, after a period of hibernation, is utilizing modern graph database management systems for managing complex, richly connected data. The two approaches have rarely been used in tandem, as microservices emphasize modularization and decoupling of services, while graph data models favor data integration.
In this study, literature on microservices and graph databases is reviewed and a synthesis between the two paradigms is presented. Based on the theoretical discussion, a software architecture combining the two elements is formulated and implemented using microservices serving content metadata at Yleisradio, the Finnish national broadcasting company. The architecture design follows the Design Science Research Process model.
Finally, the renewed system is evaluated using quantitative and qualitative metrics. The performance of the system is measured using automated API queries and load tests. The new system was compared to an earlier version based on a PostgreSQL database. The tests gave slight indication that the renewed system performed better for complex queries, where a large number of relations were traversed, but worse in terms of throughput under heavy load. Based on the these findings, a number of performance-enhancing optimizations to the system are introduced. Observations and perpectives are also gathered in a project retrospective session.
It is concluded that the resulting architecture holds promise for managing complex data rich in relations in a safe manner. In it, the different domains of the knowledge graph are decoupled into distinct named graphs managed by different microservices.

Keywords: microservices, graph databases, knowledge graphs, RDF, SPARQL, AWS, Amazon Neptune, broadcasting, media

Contents

1 Introduction **1**

1.1 Context of the thesis 2

1.2 Methodology 2

1.3 Research questions 4

1.4 Structure of the study 4

2 Central concepts **7**

2.1 Microservice architecture 7

2.1.1 Definitions 7

2.1.2 Background 9

2.1.3 Microservice best practices 12

2.2 Knowledge graphs 15

2.2.1 Graph databases 15

2.2.2 Data formats 17

2.2.3 SPARQL 19

2.3 RQ1: Graph databases in microservices 22

3 Case context **24**

3.1 APIs at Yle 24

3.2 Content metadata services 25

3.2.1 Content Index 26

3.2.2 Meta API 28
3.2.3 Relations API 30
3.3 Use cases 31

4 Migration to graph database 34
4.1 Background 34
4.2 Technological choices 36
4.3 Amazon Neptune 37
4.4 Proof of concept 38
4.5 Migration process 43
4.5.1 Phase 1 45
4.5.2 Phase 2 49
4.5.3 Phase 3 52
4.5.4 Phase 4 54
4.6 RQ2: A process for migration 55

5 Evaluation 58
5.1 Performance benchmarks 58
5.1.1 API response times 59
5.1.2 Load tests 62
5.2 Project retrospective 69
5.2.1 What was the original problem? 71
5.2.2 What was the most important thing you learned? 72
5.2.3 When would you choose a graph database now? 72
5.2.4 Is the project worth continuing? 73

6 Discussion 74
6.1 Fulfillment of requirements 74
6.2 RQ3: Perspectives to using GDB in microservices 76

6.3 Contributions of the research 79

7 Conclusion **81**

References **83**

Appendices

A Code examples **A-1**

B Retrospective answers **B-1**

List of Figures

1.1 DSRP process model (adapted from Peffers et al. 2006) 3

2.1 A partial visualization of the concept “cosplay”. 18

3.1 Data model of Content Index 28
3.2 Content-to-content links as shown in Yle Areena 29
3.3 An example of relation inheritance 32

4.1 Architecture of the Proof of Concept 39
4.2 Framework for Disciplined Evolution of Legacy Systems, applied from Weiderman et al. (1997) [90] 45
4.3 Architecture of Phase 1 46
4.4 Architecture of Phase 2 50
4.5 Architecture of Phase 3 53

5.1 Neptune CPU usage under load tests 67
5.2 Responses to the fourth question 73

B.1 What was the original problem? B-2
B.2 What was the most important thing you learned? B-2
B.3 When would you choose a graph database now? B-3

List of Tables

1.1 Thesis structure overview 4

2.1 Microservice best practices 13

4.1 Migration process phases 44

5.1 Performance comparison 60

5.2 Comparison of load test results 63

5.3 Comparison of load test results, with API v2 (opt) included 66

5.4 Comparison of load test results with API v2 (opt) and Neptune included 68

5.5 Resource details for load test runs 68

List of acronyms

ACID	Atomicity, Consistency, Isolation, Durability
Allärs	Allmän tesaurus på svenska
API	Application Programming Interface
BIDW	Business Intelligence Data Warehouse
CD	Continuous Deployment
CI	Continuous Integration
DevOps	Development and Operations
DSL	Domain-Specific Language
DSRP	Design Science Research Process
ECR	Elastic Container Registry
ECS	Amazon Elastic Container Service
GDPR	General Data Protection Regulation
IAM	Identity and Access Management
MSA	Microservice Architecture
PoC	Proof of Concept

RDF	Resource Description Framework
RDS	Amazon Relational Database Service
REST	Representational State Transfer
ROI	Return on investment
RPC	Remote Procedure Call
S3	Simple Storage Service
SNA	Social Network Analysis
SOAP	Simple Object Access Protocol
SOA	Service-Oriented Architecture
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SSH	Secure Shell
URI	Unique Resource Identifier
VPC	Virtual Private Cloud
WSDL	Web Service Definition Language
Yle	Yleisradio
YSO	Yleinen suomalainen ontologia

1 Introduction

In computing, ideas radically different from the established approaches sometimes take a while before bubbling to the surface from being an undercurrent. In recent years, we have seen the rise of concepts such as functional programming [1], artificial intelligence [2] and Lisp [3], all of which have long histories but up until now little relevance for the mainstream of computing. Graph databases are one such concept, with deep theoretical roots [4], that is experiencing a rise in popularity and practical applications, after seasons in the margins [5]. Graph databases excel in handling and querying interconnected, complex and feature-rich data and are optimized for use cases that require traversing relations between data components [6]. In an environment where both the ever more increasing volumes of data and the need to counter the shortcomings of traditional database systems built on the relational model [6], graph databases are ticking the right boxes for many use cases.

Another trend in application development today is structuring software systems into atomic, loosely coupled microservices [7]. In an architecture like this, the services expose a clearly defined interface, most commonly a REST API, for clients, while being black boxes implementation-wise. The experienced benefits of microservice approach include scalability, distributed development, modularity and maintainability of the system, as the components can be developed, deployed, scaled and monitored individually [8].

While microservices have become mainstream [9] and interest in graph databases is surging [10], these approaches have seldom been combined. In fact, they are sometimes perceived as a poor fit, as microservice architecture is organized on the principles of isolation and modularization, while graph databases emphasize integration between components, especially when they are used for organizing data into large, traversable knowledge graphs. Due to these considerations, it is worthwhile to investigate whether there is a fundamental disconnect between the two approaches and, if not, how they should be consolidated.

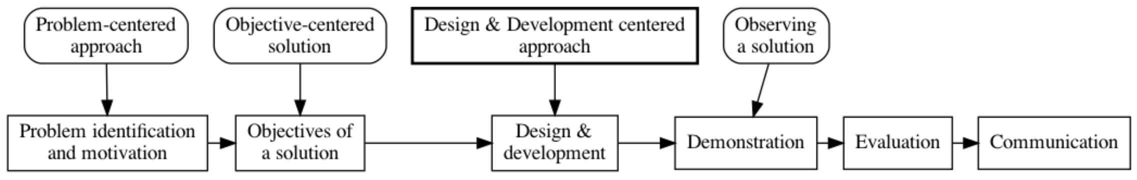
1.1 Context of the thesis

This study examines a software project conducted at Yleisradio, for which the abbreviated form *Yle* is henceforth used, the Finnish national broadcasting company [11]. In the project, three closely related microservices serving content metadata were migrated to use a graph database as a data store. In the process, the architecture of the components, for which the umbrella term *Content Metadata Services* is used, was re-structured. It is the aim of this thesis to describe, document and evaluate this transition and outline possibilities for further developing the resulting architecture.

1.2 Methodology

In this study, a *design science* approach is adopted. Philosophically rooted in pragmatism [12], design science is an approach in information systems research whereby a purposeful IT artifact is created to address an important organizational problem [13]. More specifically, the study is organized along the Design Science Research Process (DSRP) model proposed by Peffers et al. (2006) [14]. Based on a review of earlier research attempting to identify the common process elements, they identified six common phases of an information systems design science research process. The

Figure 1.1: DSRP process model (adapted from Peffers et al. 2006)


Problem-centered approach
Objective-centered solution
Design & Development centered approach
Observing a solution
Problem identification and motivation
Objectives of a solution
Design & development
Demonstration
Evaluation
Communication

phases are defined as follows:

1. *Problem identification and motivation.* Problem definition, showing the importance of it.

2. *Objectives of a solution.* Inferred from the problem statement. What would a better artefact accomplish?

3. *Design and development.* Develop the artefact based on the previous stage.

4. *Demonstration.* Use artefact to solve the problem. Generate metrics and analysis knowledge.

5. *Evaluation.* Observe efficiency, iterate back to design if needed.

6. *Communication.* Communicate the results.

Even though the model encompasses six phases, not every research process has to proceed linearly from the first to the last but the research can enter the process in different stages depending on the approach. In this study, a *design and development centered approach* is adopted. This approach takes the design and development of an artefact as the focal point. The preceding phases are touched upon, but more emphasis is laid to design and development, demonstration and evaluation. The thesis is the medium by which the results are communicated. The process is summarized in Figure 1.1.

Table 1.1: Thesis structure overview

Chapter	Main section	DSRP phase	Research question
1	Background		
2	Background		RQ1
3	Case study		
4	Case study	1, 2, 3	RQ2
5	Retrospective	4, 5	
6	Retrospective	6	RQ3

1.3 Research questions

This thesis addresses the following research questions:

- **RQ1** *How does using graph database as a persistence layer fit into microservice architecture best practices?*
- **RQ2** *How can microservices be migrated from a relational database into a graph database?*
- **RQ3** *What benefits, opportunities and risks does adopting a graph database provide in a microservice architecture?*

The questions will be addressed based on literature (RQ1), the experiences and artefacts from the migration process of the case (RQ2) and quantitative performance metrics, user comments and stakeholder perspectives gathered in a project postmortem session (RQ3).

1.4 Structure of the study

On a high level, the structure of the thesis can be broken down into three main sections corresponding to the research questions: background (Chapters 1-2, RQ1), case study (Chapters 3-4, RQ2) and retrospective (Chapters 5-6, RQ3). An overview of the structure is presented in Table 1.1.

On a more granular level, the thesis consists of the following chapters.

The introductory chapter gives an overview of the topics and problems addressed is this thesis, as well as providing motivation for the subsequent chapters.

Chapter 2 provides theoretical background for the core concepts of the study, the most central of which are microservice architecture and knowledge graphs. Concerning the former, definitions of the terms, background for the microservice approach and its objectives and related risks are touched upon. As to the latter, the theoretical foundations of graph databases are summarized and related standards, formats and languages are discussed. The ideas behind related concepts such as the Semantic Web and Linked Open Data are also covered in brief. To conclude the first main part of the thesis, an answer to RQ1 is formulated.

In Chapter 3 focus is shifted to the case study of the thesis. The content metadata services at Yle are presented along with the technical context they are running in. The three services central to the topic of this study, Meta API, Relations API and Content Index are presented. The motivation and objectives of the database migration as well as the related architectural changes are presented. In terms of the DSRP model, this chapter covers Phases 1 and 2.

In Chapter 4, the design of and migration to a renewed architecture is discussed. The process is analyzed in terms of the data model, software architecture and the process phases. In the DSRP model, this chapter corresponds to Phase 3. An answer to RQ2 is presented at the end of this chapter.

Chapter 5 is focused on evaluating the renewed system. Architectural viewpoints are discussed. The benefits, risks, maintainability and extensibility of the revamped architecture are discussed. The performance of the original and the renewed implementations are compared. In DSRP terms, this chapter corresponds to Phases 4 and 5.

The findings are discussed and RQ3 is answered Chapter 6. The limitations of the research are discussed and recommendations for future work are presented.

In Chapter 7, conclusions about the project are presented.

2 Central concepts

In this chapter, concepts central to the topic of study are discussed. We first delve into microservices — what they are, what the rationale behind them is and which problems they are used to address. Next, graph databases, knowledge graphs, semantic web and related ideas are discussed.

2.1 Microservice architecture

2.1.1 Definitions

Microservice Architecture (MSA) has become a popular approach to building back-end systems for web services in recent years. Perhaps the most widely accepted definition defines microservice architecture as "an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often as an HTTP API" [15].

As the preceding citation states, the approach is characterized by building a system using independent atomic constituent parts, which communicate using REST APIs or message queues. These components are small yet independent, each with clearly defined roles and responsibilities, forming a modular system [16]. In such a system, each component can be developed, administered, deployed, versioned and monitored individually [17]. Since services like this are loosely coupled and the communication between them is done via clearly defined APIs, they can be developed using different technologies. In order to be genuinely independent of each other, microservices should "own" the data they administer. In other words, according to microservice best practices, each microservice should have a separate database that is accessible to the other services solely through the application layer, such as a REST API [18].

Different breakdowns of the distinctive characteristics of microservice architecture have been presented. Nadareishvili, Mitra, McLarty and Amundsen [19] have defined three characteristics of a microservice architecture. First, microservice architecture is typically implemented for *big systems.* Although it is difficult to define exactly what constitutes a big system, breaking a large system down into small components is an approach for managing complexity. Conversely, it can be stated that such an approach probably would not serve a purpose for simple and small systems with limited functionality. The second characteristic of microservice architecture in their typology is that it is *goal-oriented.* This refers to the fact that this kind of approach is used for a reason, that it is a means to an end — a solution for a problem. The third aspect in their breakdown is that the constituent parts are *replaceable.* In a microservice architecture, it is thus possible to switch independent parts without the others being affected.

Bogner, Zimmermann and Wagner [20] have defined five principles for microservice architecture, these being *Bounded Context*, *Decentralization*, *Lightweight Communication*, *Single System* and *Technological Heterogenuity.* Bounded Context is a term derived from the Domain-Driven Design, referring to a specific domain where a interpretation of a concept is valid — or that an application serves. Decentralization, on the other hand, refers to the decentralized structure of the microservice-based system, where distinct services operate independent of each other, with no orchestration or technological standardization. Parts of the whole are loosely coupled, communicating via REST or RPC APIs, message queues or other lightweight messaging protocols, which is here termed Lightweight Communication. This principle is also often known as *smart endpoints and dump pipes* [7]. Single System, on the other hand, refers to the identifiable whole that the components form. Lastly, Technological Heterogenuity entails that different parts of a microservice architecture can be implemented using different technologies. This is more a possibility that microservice architecture offers than an integral feature of its implementations.

2.1.2 Background

What is known as *Unix philosophy*, as formulated in the Unix Time-sharing System manual released by the Bell Labs in 1978, is often quoted as the earliest antecedent to microservice thinking:

> *1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.*
> *2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.*
> *3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.*
> *4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.* [21]

Indeed, the analogy between Unix command-line applications and microservices is not as far-fetched as the temporal distance between the dawn of the Unix era and the current age of cloud computing might suggest. As is often the case in computing, ideas have lasted longer than implementations. These kind of "maxims", to use the original terminology of the Bell Labs manual, are applicable on a high level. They are about how to think about building software rather than anything as palpable as an architectural pattern. As Unix utilities, microservices should have limited scope, be robust and work predictably, be composable and cheap to spin up and rewrite.

As an architectural style, microservices are seen to have grown out of *Service-Oriented Architecture (SOA)*. Like microservices, SOA is characterized by the pattern of assembling a complex system out of autonomous components. The SOA design paradigm is defined on a high level, without consideration to actual implementations or standards [22]. Despite the general nature of the official definition, SOA implementations have traditionally shared certain common technological choices, such as SOAP/WSDL APIs and building the system around a central Enterprise Service Bus (ESB). It is a question of definition whether microservice architecture is viewed as a form of SOA pattern or a separate paradigm that has grown out from it [20][23]. Some consider the term SOA too carelessly used to be useful for describing microservice architecture, which is understood in a more specific sense [7]. The relation between these two concepts has sparked a multitude of definitions and terms. Microservices have been called "fine-grained SOA" and "SOA done right" among other things [16]. Cockcroft (2016) summarized the idea of microservices as "loosely coupled Service-Oriented Architecture with bounded contexts"[24].

Some general technological trends can clearly be identified as drivers to the rising popularity of the microservice approach. The most crucial of these are *cloud computing*, *container technologies* and *DevOps*.

DevOps is a philosophy and a set of practices intended to bridge the gap between development and operations. Where these functions have traditionally served by distinct teams, one developing features for the software and the other managing it in production, DevOps tries to bring down the proverbial wall separating these silos [25]. While the concept primarily refers to an approach and not any singular tools or technologies, using modern tools and cloud-based infrastructure makes this feasible [26]. Central to the DevOps way is automating testing and provisioning software using CI/CD pipelines, as well as automated monitoring and managing infrastructure as code. Having automated pipelines for provisioning code into cloud makes it possible to deploy new features continuously without explicit releases.

Container technologies are often utilized in DevOps context [25]. Containers, the best known and most widely used being Docker, are a common way to package and run applications in a similar environment in different contexts, such as locally on the developer's machine and in the cloud. Containers are an approach to virtualization, where the applications run in separate environments but share the OS kernel [27]. Compared to virtual machines, containers are lightweight, consume less resources and can be built, started up and shut down as needed almost instantly [28]. They can also be organized into clusters and orchestrated using tools like Docker Compose and Kubernetes. Container environments, such as Docker images, can also be published and shared using centralized or private registries like Docker Registry or Amazon's Elastic Container Registry (ECR). A typical way to deploy microservices, or any other software for that matter, using DevOps principles is configuring a CI/CD tool like Travis, CircleCI or Amazon CodeDeploy to build a container whenever new commits are pushed to version control, run the tests and if they pass and deploy the artefact into the cloud.

Microservice architecture ties into DevOps also in the way roles and responsibilities are managed. In the microservice way, the responsibility of operating the service usually lies on the same team, usually a small one, as its development. As Werner Vogels, then CTO of Netflix, famously stated the principle: *You build it, you run it* [29]. With responsibility comes autonomy. In microservices, a distinction can be made between shared and local capabilities [19]. The former refer to a shared set of services in a large organization, such as hardware, code management tool, service orchestration, security-related practices and common architectural policies. Local capabilities then encompass team-level choices and policies that the autonomy mandates. These can be for example different tools and configurations.

While microservices can be technically run on on-premise hardware, they are considered as a cloud-native architectural pattern [30]. Running microservices on a cloud platform makes resource allocation and scaling a matter of configuration. In a cloud environment, services can be *scaled horizontally* by adding more application instances. *Vertical scaling* by allocating more resources like memory and CPU as needed, is likewise possible.

2.1.3 Microservice best practices

As the idea and practice of using microservices is not based on a single authoritative set of rules but rather a multitude of somewhat different interpretations of the concept, no single official set of best practices for microservices has been compiled. Instead, many such listing have been published, with more or less overlapping suggestions. For this discussion, a yet new list has been compiled, combining and synthesizing ideas from different sources, [7] [8] [19] [18] [23] some of which are based on a literature review or industry survey themselves [17] [31] [32]. The points listed in Table 2.1 can be categorized under a few common themes: *APIs*, *data*, *design*, *communication* and *operation*.

On the topic of APIs, there is a consensus on that an API gateway should mediate traffic to microservices [8] [19] [18]. It is responsible for routing requests to different services, API composition and services like authentication. As these responsibilities are delegated to the gateway, the services can operate in protocols that would not be practical for internet access. The API gateway also determines the contract to the clients. Developers are free to alter the API of their service, as long as the API gateway is updated accordingly. According to the literature, customer-facing APIs should also be versioned and the old versions should stay usable to maintain backwards compatibility [8].

Table 2.1: Microservice best practices

Name	Description	Source
API Gateway	Helps with monitoring and API updates	[8] [32] [17] [19] [18]
API Versioning	Backwards compatibility	[8] [31] [18]
Data Ownership	Data is owned by a single service	[17] [31] [18]
Database Cluster	Database cluster with distinct schemas	[17]
Decentralized Data Management	Each service has its own database	[17] [7] [18]
Design for Failure	Circuit breaker, safeguards	[7] [8] [17] [32] [19] [18]
Evolutionary Design	Changing the design as needed	[7] [18] [23]
Infrastructure Automation	DevOps approach, automatic deployment	[7] [18] [23] [19] [17] [32]
Limited Scope	No micro-monoliths, maintainability	[31] [19] [18]
Loose Coupling	No ESB, no sharing custom libraries between services	[31] [19] [23] [18] [17] [32]
Organized around Business Capabilities	Capabilities over organizational structure	[7] [19] [31] [18] [23]
Service Registry	Discoverability	[17] [18] [19] [18]
Team Operates	Same team is responsible for building and operating	[18]
Service Independence	Maintainability, individual deployment	[8] [18] [31] [19] [17]
Smart Endpoints, Dumb Pipes	Communication via APIs and MQ's	[7] [8] [18] [19] [17] [32]

Concerning data, a widely held principle is that each microservice should control and "own" their data. In practice, this is usually taken to mean that each service should have its own database instance, which should only be accessible to other services through the application layer. However, there are alternative approaches too. One such approach is the "Database Cluster" pattern, which entails that each service stores its data into a shared database cluster or even a shared database [17]. In such a scenario, a comparative degree of data hygiene is achieved by using different database schemas for different services and allowing each service to only touch data stored in its own schema.

As to design, best practices as defined here state that microservices should have as limited scope as is feasible and be structured around business capabilities [19]. The same team should be responsible for building and operating the services. The teams should be cross-functional, encompassing people with diverse skill profiles. The services should evolve alongside business needs and their design should be evolutionary, not locked.

Communication between the services should be done via APIs, message queues and similar protocols. Also, sharing custom libraries between services is discouraged to avoid coupling the services together this way. A service registry of some kind is necessary for keeping track of the available services and making them discoverable [17] [18].

Finally, as to operations, state-of-the-art tools should be utilized for detecting anomalies. What is termed here as the Circuit Breaker pattern means setting up comprehensive logging, monitoring and alerts for services so that problematic situations in production can be detected as they occur and necessary measures can taken without delay.

2.2 Knowledge graphs

2.2.1 Graph databases

Graph databases are systems where the data is modelled as graphs — that is, as *nodes* and *edges* connecting them [33]. In a graph model, nodes describe entities and edges the relations between them. In a graph database context, both nodes and edges have *labels* describing their content. Labels can convey information about the entity set as well as the relation [34]. The labels are also known as *properties* and this kind of graphs as *property graphs.* The theoretical basis for this is derived from graph theory, a field of mathematics specializing in the study of graphs.

A graph can be defined formally as follows. A graph is a pair (V, E), where V is a set of nodes and E a set of edges connecting the nodes. In the case of a directed graph or *digraph* the set E is ordered. In other words, the pair (u, v) describes an edge from u to v, but if the path can be traversed the other way as well, the set E contains also the edge (v, u) [4].

A *labeled graph* can be defined as a tuple

$$G = (V, E, \Sigma, \lambda) \tag{2.1}$$

where V is a set of nodes, $E \subseteq V \times V$ a set of edges connecting them, Σ a set of labels and $\lambda : V \cup E \to \Sigma$ a function mapping the labels to nodes or edges [34].

In mathematical terms, a graph database is a set of directed labeled graphs. In technical terms, it is a software system comprised of two main layers: the *underlying storage* and the *processing engine* [33]. The first of these is responsible for persisting the data, either using a native graph storage or serializing it in a format accepted by a relational or document database. The latter accepts inputs from the user and performs them on the storage layer, returning results defined in the operations.

Unlike relational databases, graph databases are schemaless. This means that the structure of the graph does not need to conform to a predefined structure. Constraints are also typically not enforced, although there are tools and specifications for this in existence [35]. Also, where knowing the table structure of a relational database is necessary for performing queries on it, this is not the case for graph databases — knowing the types of relations inside a graph database is sufficient. The data models in graph databases are also often open for extension. In relational databases, altering a database structure calls for a migration but in graph databases this can be simply a matter of adding new relations. This is not to say that designing a data model is any less necessary in graph database context than it is in the relational world. The data of a graph database can be structured as one big graph or a set of distinct graphs [36].

Graph databases and formatting data as a graph are a natural fit for cases where the data is rich with connections. Typical examples are different kinds of social structures, recommendations, geospatial data, Master Data Management and network and data center management [33]. Sometimes, the interesting thing is understanding how a dataset is structured. One example of such a use case is *social network analysis (SNA)*, a popular research paradigm in the social sciences, where graph theory and graph databases are valuable tools [37]. A graph-based approach is also useful for cases where the data is not only densely connected but also dynamic or unevenly structured [37]. Shortly put, graph databases are a good fit for use cases where the data is structured as a graph or a web.

Graph databases are optimized for graph-like queries, such as finding a shortest path between two nodes. In relational databases, operations like this can be computationally expensive, requiring multiple joins between tables [38], where in graph databases, relations are first-class citizens. In other words, in relational databases the relations between tables are computed as the queries are executed, while in graph databases relations are stored along with the other entities [39]. The efficiency by which graph databases can process relations is in most cases based on a concept of *index-free adjancency*, at least in the cases where a native graph storage is used for persisting the data [36]. This means that the nodes stored on disk have physical pointers to their adjacent nodes and relations, which is why queries on the data are efficient without using indexes. However, methods for indexing RDF data for efficient queries have been developed [40] and are used in commercial products as well [41].

Knowledge graph is a concept that is closely related to graph data models. The term refers to a rich collection of information, modelled semantically in to a graph and arbitrarily queryable, intended to be used by humans. A definition formulated by Ehrlinger & Wöß (2016) reads as follows:

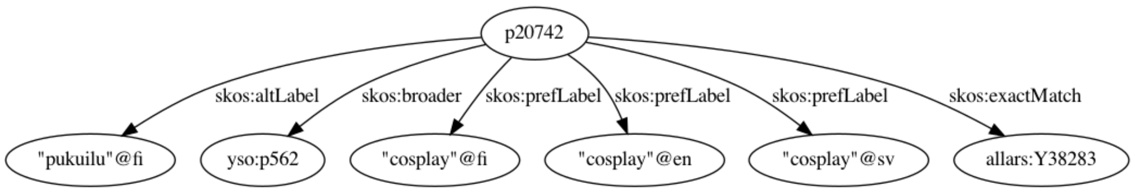
> *A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge.* [42]

This definition identifies two components to a knowledge graph; an *ontology* and a *reasoner*. An ontology, in the sense used in information sciences, is an "explicit specification of conceptualization" [43] [44] or, less formally, "a representational vocabulary for a shared domain of discourse - definitions of classes, relations, functions and objects" [43]. A reasoner or a reasoning engine, on the other hand, is a system that allows deriving new knowledge from the ontology — a database or a query engine [44]. In this definition, what differentiates a knowledge graph from a collection of data — an ontology — is the reasoner component. Following this definition, a graph database management system and the data stored into it qualifies as a knowledge graph.

2.2.2 Data formats

Data that is organized as a graph can be expressed in various different formats. For the present discussion, the most important of these is RDF or *Resource Description Framework*. It is a data model, or strictly speaking a family of specifications, by the

Figure 2.1: A partial visualization of the concept “cosplay”.



Word Wide Web Consortium (W3C). The first version of the RDF specification was published in 1999 [45], version 1.0 in 2004 and 1.1, which is the latest edition, in 2014 [46]. RDF, as well as many related technologies such as the SPARQL language and the idea of Linked Data [47], were originally developed with the vision of Semantic Web in mind [48]. Semantic Web was conceptualized as a global, distributed RDF-base data graph [49] [48]. Although the Semantic Web has failed to materialize in the degree it was originally envisaged, technologies pertaining to it have found more specialized use cases.

RDF is a data model that is based on making statements about resources. These statements are expressed as *triples* consisting of *subject*, *predicate* and *object*. In RDF terms, subject is a *resource* identified with a global unique persistent identifier, a URI. The predicate defines a *property* for the subject. In a graph comprised of RDF triples, the properties correspond to labels. Compared to tables in relational databases, predicates correspond semantically to table attributes [50]. The object can be defined in different ways, for example as a URI or a literal such as an integer or a string, much like the values in rows of SQL tables. Consequently, converting RDF data to relational format and back is feasible [51]. A simple example illustrates how RDF triples work.

The example comes from the General Finnish Ontology YSO [52]. The code, in Turtle format, is found as Appendix A. Its subject is the concept “cosplay”, denoted by the URI *yso:p20742*, a short form for *http://www.yso.fi/onto/yso/p20742*. In Figure 2.1, six properties have been defined for it. The *skos:prefLabel* predicates state that the concept is known as "cosplay" in Finnish and Swedish as it is in English. The *skos:altLabel* property denotes that the concept is also known by the alternative term "pukuilu" in the Finnish language. An hierarchical relation to another concept is defined using the *skos:broader* predicate. This states that "cosplay" is subcategory to the concept *performing (artistic creation).* Finally, the *skos:exactMatch* predicate forms what is known as a "bridge" between two ontologies, denoting that the concept refers to exactly the same thing as the term "cosplay" in Allärs, the deprecated Finnish Swedish-language thesaurus.

The relations defined via RDF predicates are not transitive. In other words, data structures expressed as RDF data are directed graphs. Although some types of relations are implicitly transitive, such as *skos:exactMatch* in the example, these kind of relations should be expressed explicitly in both directions.

RDF is a conceptual data model for representing graph data. To put it into practical use, the data needs to be serialized using a file format designed for it. Most popular options for this are RDF/XML, Turtle and JSON-LD. RDF/XML was a popular option in the early years of RDF but has been losing ground in recent years due to its verbosity, repetitiveness and being laborious to edit by hand. Turtle [53] stands for *Terse RDF Triple Language.* As is apparent from its etymology, the Turtle format was designed by W3C to curtail the problems of RDF/XML by being terse and readable for humans as well as for machines. JSON-LD [54] is an acronym from *JavaScript Object Notation for Linked Data* and it is yet another data format specification from W3C. JSON-LD is a way to encode RDF data using the popular and well-known JSON format.

2.2.3 SPARQL

Where the SQL language is the *de facto* standard for interacting with relational databases, it is not suited to be used with graph databases [33]. Because of this, graph databases are sometimes classified as belonging to the NoSQL category of databases. For graph databases, there is no single query language over others but modern graph database management systems typically support multiple query languages. Arguably the best known are *SPARQL*, *Gremlin* and *Cypher*, the first of which is examined here in more detail.

SPARQL, which is a recursive acronym for *SPARQL Protocol and RDF Query Language*, is a query language for RDF data developed by the World Wide Web Consortium in 2008 [55]. The latest version of the language (1.1) was published in 2013. As the name implies, SPARQL is based on SQL. SPARQL queries are based on pattern matching against the triples forming the graph, but the language also provides operations for ordering, filtering and other ways of manipulating the result sets.

The structure of a SPARQL query can be broken down into different parts. A *PREFIX phase* starts the query. Here, aliases are defined for the URI identifiers used in the query, so that they do not have to be repeated inside the body of the query. This is followed by the query body, which consists of three phases: *pattern matching*, *solution modifiers* and *output*. More generally, the query is of the form $H \leftarrow B$, where B is the body of the query, an RDF pattern expression potentially containing variables, conjunctions, disjunctions and optional parts, and H the head of the query, containing an expression that defines how the results should be formulated. As the query Q is evaluated against data D, the operation is done in two steps. First, the body of Q is matched against D to get a set of bindings, which are then formulated according to the instructions in the head of Q, using operators largely familiar from SQL [56].

Code Snippet 1 demonstrates a SPARQL query to obtain the total amount of programs from a data graph. In the first rows, two prefixes are declared. The first, *rdf*, is a reference to the RDF definition of the W3C and is included in almost

Code snippet 1 Retrieving the total number of programs in a graph

```
{sparql}

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX yleid: <http://id.yle.fi/>


SELECT (COUNT(?s) as ?count)

WHERE {

  GRAPH <http://content-index.yle.fi/>

  {

    {

      ?s rdf:type yle:ProgramEditorialObject .

    }

  }

}
```

all SPARQL queries. The other one identifies a namespace internal to Yle. The *SELECT*-part, or the head H of the query, determines the required format of the results. Here, the amount of instances of the variable *?s* are computed using the function *COUNT* and the result is bound to a new variable *?count*. The pattern matching phase, the body B, is defined inside a *WHERE* block. The operator *GRAPH* is used to point the query to the graph *<http://content-index.yle.fi/>* in the database. This is followed by the pattern matching, where each triple with the relation *rdf:type* defined for the object *yle:ProgramEditorialObject* is fetched, their subject is bound to the variable *?s* and returned. The dot at the end of the row is a statement terminator.

2.3 RQ1: Graph databases in microservices

Based on the preceding discussion, a synthesis between microservice architecture and knowledge graph -based approach can be attempted. In Section 1.3, RQ1 was formulated as *How does using knowledge graph as a persistence layer fit into microservice architecture best practices?*

Organizing applications based on microservice principles and organizing data into a knowledge graph can be seen as contrasting forces. Where microservice architecture stresses modularization and hygienic decoupling of services, knowledge graphs favour integration of data into a large interconnected structure. However, the two paradigms address different concerns — a knowledge graph is an approach and technical solution to organizing data, whereas microservice architecture is a paradigm for structuring, designing and operating software. It is in the intersection of these two worlds where they can be seen to be in conflict. A common principle in microservice architecture is to isolate the persistence layers of the individual services by assigning an own database for each microservice. A microservice application should own the data it manages, and the said data should only be accessible to other services or clients through the application layer [18]. Storing the data of multiple microservices into a single graph database seems to violate this pattern.

One approach to curtail this issue would be to follow this design principle strictly, and to assign each microservice a graph database of it own. However, the downside to this would be that performing queries combining data from different services would require multiple API calls and come with a possibly significant performance penalty imposed by the network latency. While this is feasible and it would be in line with the way microservices usually interact, some of the potential benefits of using a graph database is lost. Another approach more suitable to a graph database context is applying what is known as the Database Cluster pattern, where the services share a database but keep their data separate within it. In such a scenario, it is important to set up the database in a way that prevents the services accidentally messing each other's data. Graphs databases supporting the RDF or property graph specifications allow creating multiple *named graphs* inside a single database instance [57]. Here, a named graph is conceptually similar to a named schema in relational databases like PostgreSQL. If an architecture is designed in a way that only allows the microservice to modify data in the graph it owns, a reasonable degree of data hygiene is achieved. This can be considered as a variation of the *database cluster pattern* recognized in the microservice patterns overview.

The answer to RQ1 can be formulated as follows: *using a graph database as a persistence layer fits well to into microservice architecture best practices when (1) the data owned by the services is structured into different graphs and (2) measures are taken to ensure that the services can only modify data in the graph they own.*

3 Case context

In this chapter, focus is laid on the context of the case study. Before the case is delved into, the technical and business context are succintly presented. To conclude, a motivation for the architecture renewal and database migration are discussed.

3.1 APIs at Yle

The systems architecture at Yle is comprised of dozens of microservices. While some of the internally developed systems are structured as monoliths, such as news site backends based on Drupal and other content management systems, many customer-facing applications like the streaming service Yle Areena and the mobile applications, such as the Uutisvahti news app, fetch their content and the related metadata from a fleet of microservices. These services are hosted in Amazon Web Services and written mostly in the Clojure and Python programming languages, along with some Scala [58]. Yle was an early adopter of this architectural style, having organized its backends as microservices since the year 2014 [59].

For a broadcasting company, microservice architecture can have multiple upsides. For instance, Yle Areena, the most popular streaming service in the country [60], is available to users through the Web [61] and via applications for different platforms, such as Android, iOS, iPadOS and all major Smart TV brands. The backend to these systems is a constellation of microservices that each of the different clients can use to search and stream video and audio content and related metadata [62].

Some of the APIs are also open for public, which makes it possible to develop third-party clients [63] within the limits of API terms of service [64]. Another palpable benefit of using a microservice architecture hosted in the cloud is being able to cope with traffic spikes. Certain events, such as broadcast of the President's Independence Day Ball, election results broadcasts, the Eurovision Song Contest and major sports events with Finnish athletes participating, generate huge amounts of traffic to Yle's services [62]. During these peaks, the amount of data transmitted can exceed 800 gigabytes per second [65] [66]. To cope with such loads, the relevant services can be temporarily scaled up horizontally and vertically. For major events on TV, this is made easier by the fact that the upsurges of traffic are expected. For unexpected events like breaking news, which also generate traffic spikes to news sites, microservice architecture and related mechanisms are also valuable as monitoring helps notice the spikes quickly and auto-scaling can be enabled for crucial services [62].

Microservices architecture is also easier to extend than monoliths. For example, as more interactive elements are included in broadcast concepts, like chats, voting or gathering other kinds of real-time consumer feedback, the related applications can be developed and deployed separately, and used via APIs from all the different clients [65]. Another benefit achieved at Yle by using microservices is the loose coupling of legacy systems, for example those used for publication planning and managing content, from client applications [62]. The legacy systems need only to be integrated into a mediating microservice application, through which the clients can then use the underlying system indirectly.

3.2 Content metadata services

Three closely related microservices were involved in the graph database transition. Since each of these services processes metadata related to content, whether the content is video, audio or text articles, they are referred to using an umbrella term *Content Metadata Services* in this thesis. In the following subchapters, the roles of each of these services are briefly described. All of them are developed using similar technologies. They are written in Clojure, deployed as Docker containers into the Amazon Elastic Container Service (ECS) and their data is persisted into Amazon RDS PostgreSQL databases. All of the services also publish the changes in their data as change messages to a RabbitMQ message queue. Consumer applications inside the same network can listen to these messages and react accordingly.

3.2.1 Content Index

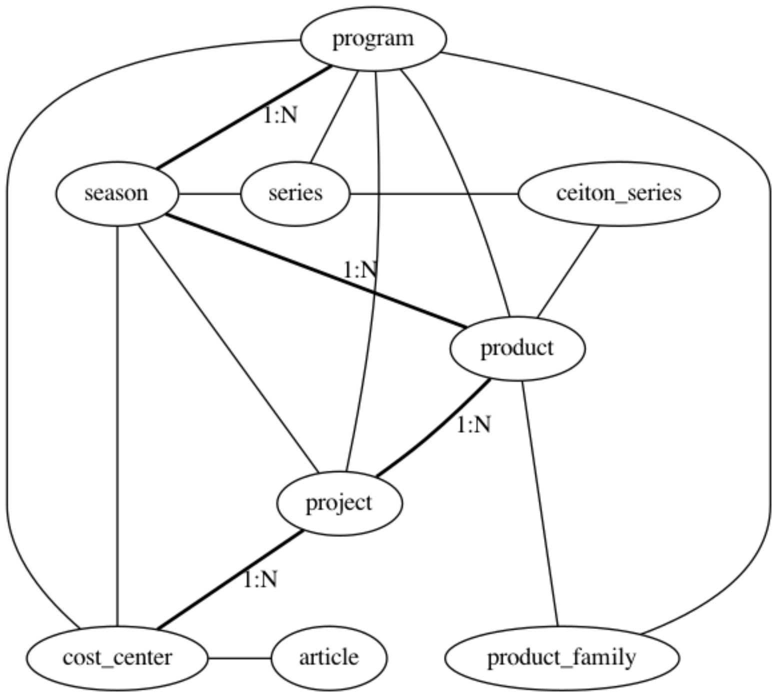
Content Index is a service that tracks relations between content IDs. It reads in operational data related to production planning and outsourcing from a Business Intelligence Data Warehouse (BIDW) service, as a daily data dump, and consumes RabbitMQ change messages from *Program Store* and *Articles API*. As the names of these services hint, Program Store stores and publishes information about TV and radio programs, their publication times and other related metadata. Articles API deals with similar metadata related to news articles originating from the different content management systems. The role of Content Index is to bridge the gap between operational data and content metadata by analyzing the relationships between these different IDs, using various heuristics, and to publish these relationships through its API.

Figure 3.1 shows the different ID types and their interrelations in Content Index. The lowest-level entity in the diagram is the *program*, a single program or clip that end users can consume through the various client applications. Oftentimes, a program belongs to a *season* or *series*, or both. A series can contain multiple seasons, or none. Seasons and series in turn belong to a *product*. In this context, a product is an operational concept that refers to an artefact produced in the scope of a *project*. An example of a product could be for example *Docventures 2020* — all the different episodes, articles, podcasts, clips and other content related to a production done in a year. Products can be grouped into *product families*. An example of a product family would the children's program Pikku Kakkonen [67], which has been running for decades and is comprised of a website, articles and other related content besides thousands of video episodes. A product family groups all these different artefacts from different years together. A project is the process wherein a product is produced. Finally, each project has a *cost center*, which is the most high-level concept in this hierarchy. Besides the audio-visual content, Content Index also keeps track of text articles, each of which has a cost center.

As is apparent from the diagram, the IDs do not form a clean hierarchy. It is made even messier by the fact that the cardinality of most of the relations is of type *N:M*. However, there exists a "default path" of *1:N* relations from a program all the way to a cost center, which can be traversed upwards and downwards, depending on the starting point. In Figure 3.1 this path is bolded. The most common use case of Content Index is requesting the *ancestors* or *descendants* of an identifier. Queries of the type ancestors typically return only one instance of each ID type above the queried ID in the hierarchy, whereas descendants-queries branch out on each level. The extreme case is ancestors-query for a cost center, which can have tens of thousands of related IDs — such as for the cost center of news broadcasts. Technically, traversing the data model and retrieving all the different related IDs requires multiple computationally expensive joins between tables. The appendix contains an example query and its result from Content Index as the Snippet 6.

Unlike the other two Content Metadata Services, Content Index is a service that is only available for internal use within the company. This is due to the fact that while it only serves ID hierarchies, some of the data it processes is sensitive. The data of Content Index that has other than internal use cases is also available through

Figure 3.1: Data model of Content Index

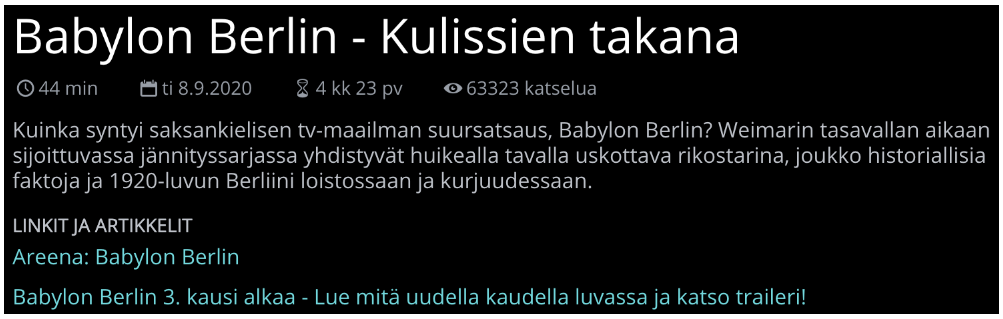


other channels, such as the public *Programs API* [68].

3.2.2 Meta API

Meta API, the name being short for *metadata*, is a service for managing and querying concepts and — for historical reasons — content-to-content-relations. The concepts in Meta API are used in descriptive metadata about content, such as subject headings describing the content, style, atmosphere or other qualities of an article, program, series or other entities. However, Meta API only stores information about what concepts are available for the annotations along with some metadata about them, while the information concerning what concepts are related to which content is maintained in another service, the Relations API. When these concepts are used in subject indexing, they are referred to using the ID of the concept, while the data

Figure 3.2: Content-to-content links as shown in Yle Areena



Babylon Berlin - Kulissien takana

44 min · ti 8.9.2020 · 4 kk 23 pv · 63323 katselua

Kuinka syntyi saksankielisen tv-maailman suursatsaus, Babylon Berlin? Weimarin tasavallan aikaan sijoittuvassa jännityssarjassa yhdistyvät huikealla tavalla uskottava rikostarina, joukko historiallisia faktoja ja 1920-luvun Berliini loistossaan ja kurjuudessaan.

LINKIT JA ARTIKKELIT

Areena: Babylon Berlin

Babylon Berlin 3. kausi alkaa - Lue mitä uudella kaudella luvassa ja katso traileri!

related to the concept, such as the textual title in different languages, are fetched separately from Meta API.

While the data of Content Index is updated programmatically as reaction to changes in the master data it reads in, the data in Meta API is curated both manually and automatically. The common way to add concepts in its database is via different content management systems as an article or some other content is published. In this case, the user searches for concepts in remote sources, such as Wikidata [69] or Finto [70], saving suitable concepts into Meta API. Concepts can also be added, searched and edited using a web-based GUI simply known as *Concept Editor*.

Meta API was originally responsible for storing both concepts and their relations to content items, but it was later broken down into two distinct applications. Relations between content and concepts were moved into Relations API, while the concepts were managed in Meta API. Counter-intuitive as it may seem, one type of relations was not moved out from Meta API, namely content-to-content-relations. In practice, this means relations from articles to programs. These links are created by journalists when publishing articles. If a video or audio program has relevance to an article, these are linked and the relation is stored in Meta API. Figure 3.2 illustrates the practice by an example from the Yle Areena streaming service, where articles related to the TV series Babylon Berlin are linked to a making-of -documentary about the show.

3.2.3 Relations API

Relations API stores relations between concept and content IDs. The data model of Relations API is the simplest of the three Content Metadata Services, as it stores only relations between IDs. For each relation, Relations API stores the ID of the relation, the IDs of the source and target, the ID of system that has created the relation and the type of the relation. Currently there are eight relation types: *isSubjectOf*, *isGenreOf*, *isPersonOfInterestOf*, *isAtmosphereOf*, *isRelatedEventOf*, *isTargetAudienceOf*, *isEditorialSectionOf* and *isCountryOfOriginOf*. From these labels it is apparent that the relations from concepts to content can be both descriptive and structural metadata [71].

The relations stored in Relations API are created both manually and automatically. For creating relations — or giving subject headings to different content items — manually, a tool named *Tagger* exists. It is a browser widget written in ClojureScript that can be embedded in browser UIs of different systems. It allows the user query concepts from Meta API and associate them to content IDs, saving the relations to Relations API. Using Tagger, programs and other content can be tagged with subject headings in different phases of its life cycle, inside different systems. This can also be done directly using the different content management systems used for publishing articles.

Besides manual subject indexing, the journalists have a tool for automated subject indexing at their disposal as well. This application, *Text Analysis API*, a service integrated to various content management systems for publishing articles. It scans the text to publish and suggests subject headings describing the content, from which the journalist can then pick the most suitable ones. The suggested keywords originate from a third-party service using a proprietary ontology, although open source alternatives are being tested as well [72].

An example query in Snippet 7 illustrates how Relations API can be used. In the example, the queried ID *1-2155797* refers to the TV series *Il commissario Montalbano*. The result set contains four relations that point to the series. The relation *52-47016285* is of type *isCountryOfOriginOf* and points to the ID *18-177017*. A query from Meta API reveals that this ID refers to Italy. The three *isGenreOf* relations tell that the series is — yes — a *series* (18-299297), a *thriller* (18-299306), the third genre being *crime* (18-299286).

It is apparent from the example that the data model of Relations API has been designed from the get go to emulate the statement triples of RDF. As was noted in Section 2.2.2, RDF statements follow the subject-predicate-object pattern. In Relations API, the subject is an identifier, the predicate a relation type and the object another ID.

3.3 Use cases

The three Content Metadata Services are functionally tightly interlocked. For annotating content with concepts, all three services are needed. In this domain, Meta API and Relations API are used for associating concepts to content, and Content Index for *keyword inheritance*. Using Content Index, it is possible to annotate the parts high in the ID hierarchy as depicted in Figure 3.1 with subject headings that the lower-level items inherit. This way, keywords shared by all low-level entities can be set to their parent entity instead of being repeated for, for example, every episode of a series. In the previous example with Snippet 7, the keywords for the TV series “Il commissario Montalbano” were set on series level, which means that they are inherited by each episode. Besides the shared keywords, the episodes can be annotated with keywords of their own, usually specifying the content of the episode in more detail.

Figure 3.3: An example of relation inheritance

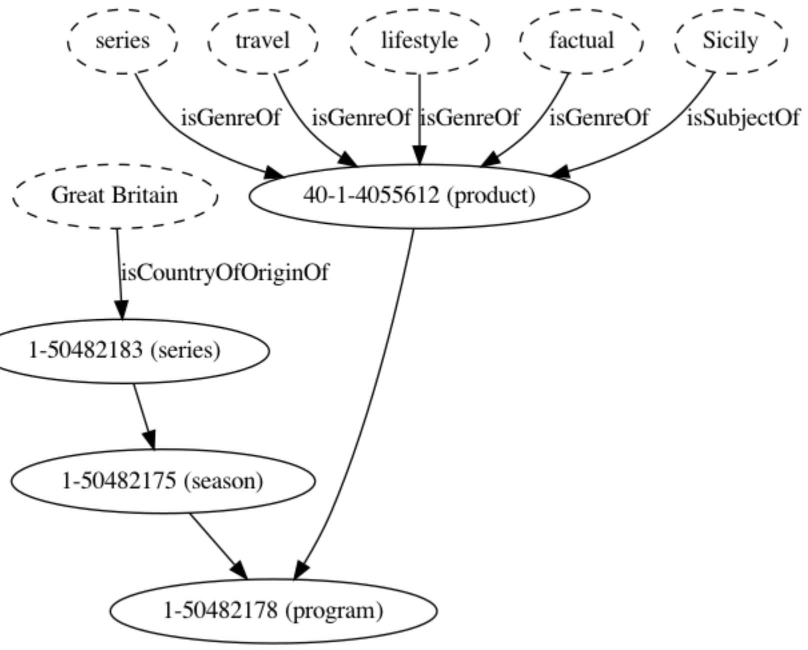


Figure 3.3 illustrates how a program inherits relations from above. The program in question is the first episode in the travel show *Ray Winstone's Sicily* [73]. The keywords associated to it are all set on the higher rungs of the ID hierarchy. The product level has concepts *series*, *travel*, *lifestyle*, *fact* and *Sicily*. The series level has the concept *Great Britain* with the relation *isCountryOfOriginOf*, along with some other subject headings that are redacted from the illustration to make it readable. All of these relations are shown for the individual episodes, which in this case do not have concepts related directly to them. This functionality is made possible by interaction of the three Content Metadata Systems: Relations API calls Content Index to get the hierarchy related to an identifier, which can then be used for showing the inherited relations either through its API or in its changes messages, which clients such as *Packages API* can consume.

Besides subject indexing, Content Index has a number of other use cases as well. Perhaps the most significant of these is its use by the customer analytics team at

Yle. The PostgresSQL database of Content Index contains a materialized view over the ID hierarchies, which analysts can access directly using the Tableau Business Intelligence software. This way, Content Index can be added to the list of data sources used for analytics and its data can be combined with other metrics. This makes it possible to aggregate usage metrics on a higher level than individual items or series. For instance, with this setup it is relatively straightforward to generate a view collecting the all the metrics related to a operational level entity, like product or a project. Combined with financial numbers related to a production, this makes it possible to calculate return on investment (ROI) which can in turn be used in planning future productions. The usage metrics can also be compared and visualized on a higher level than a mere series or season.

4 Migration to graph database

In this chapter, the case of the study is described. Systems architecture at the different phases of the migration project, data flows, data models along with the reasoning behind choices are discussed. To conclude, an answer to RQ2 is formulated.

4.1 Background

While the original implementation of the Content Metadata Services satisfied the functional requirements set to them, some of the non-functional attributes, such as the architecture and division of labor, had their drawbacks. Due to the strict separation of concerns between the services on the one hand, and to the tight interlock between them on the other, the system they form was considered to be complex [74]. Due to the original design, even a simple task such as associating a subject heading to an article requires at least five HTTP calls to the APIs of the three microservices [75]. The situation was not ideal from the users' point of view, due to the excessive amount of calls and the resulting added latency, and it was relatively difficult to maintain. As noted in Chapter 2, the idea of a microservice architecture is to break the services down into as small components as possible for enabling the necessary functionality, but it can be argued that the fragmentation was taken too far in the case of the three Content Metadata Services. This is more due to historical reasons and the incremental nature of adding functionality than to conscious design. As noted in the discussion about microservice best practices in Section 2.1.3, an evolutionary design and scoping of the services is by itself a typical *modus operandi* when developing microservices. However, in the case of long-living applications, a larger re-structuring is sometimes in order.

The latest of the three Content Metadata Services, Content Index, was implemented, first as a pilot project, during the latter half of the year 2019. The scope of the pilot phase of the project included testing out a graph database for the use case. This was due to the interest at the organization towards graph database technologies and the possibilities their adoption in suitable contexts could provide. Because of this, a Proof of Concept for an implementation on top of a graph database was created for the Content Metadata Services along with the implementation of Content Index, even though the original persistence solution was a traditional SQL database.

One key reason why the Content Metadata Services were selected for testing the graph database was, beside the aforementioned architectural factors, the structure of the data they manage. The content metadata in the three different services forms a logical whole, ergo it could be organized into a single data structure. The data also seemed to fit a graph representation due to its inherent structure [76]. Graph databases are all about explicitly encoded relationships between entities, which is exactly what Relations API and Content Index provide to their clients, whilst Meta API manages the entities themselves. Providing analysts with access to a SPARQL endpoint to this data could also enable interesting, unforeseen uses.

Before testing the graph database in practice, there were ideas about new functionalities the new implementation could enable. One idea present in the discussions was re-structuring the currently flat structure of the concepts used as subject headings into an hierarchical ontology [74]. This would enable making the subject heading searches and tagging more semantic, for example by returning articles tagged with a hyponym, or a more specific concept, of the search term used in a query. All in all, structuring different sets of data into a graph database and building services around it was seen as an exciting prospect with a lot of potential for intelligent services.

To formulate the motivation for the graph database migration in terms of the DSRP model as summarized in Figure 1.1, the two first definitions can be expressed as follows.

1. **Problem identification and motivation**. The structure of the current Content Metadata Services is too fragmented, HTTP call chains are unnecessarily long and data maintenance is unwieldy.

2. **Objectives of a solution**. The new implementation should be simpler and work with less requests. In the future, the potential of the graph database could be utilized more fully, for example by forming ontologizing the concepts currently stored in Meta API [74]. The solution should also be more performant for complex queries.

4.2 Technological choices

The most crucial choice to make in the scope of the experiment was choosing the graph database management system to test. Multiple alternatives are available in the market, the current market leader being Neo4j [77], an open-source solution [78] based on the Property Graph data model [37]. Many public cloud providers have also started to include a proprietary graph database solution in their services [77].

Since Neo4j had been tested before in an unsuccessful project, the team wanted to try out something different. Since practically all of Yle's microservices are hosted in Amazon Web Services [66], the recent graph database solution offered by Amazon, called Neptune, seemed worth looking into. Using a graph database hosted and managed as part of the AWS services seemed to offer perks in terms of integration to the existing cloud infrastructure, by means of for example Identity Access Management (IAM) and backups. Due to these considerations, a decision was made to test Neptune in the project.

As Neptune supports both the RDF data model with SPARQL query language and the property graph model with the Apache TinkerPop Gremlin language [79], but not both simultaneously, another choice needed to be made with regards to the data format. Since the team had a passing familiarity with RDF and SPARQL but none with Property Graph or Gremlin, RDF was chosen as the data model and consequently SPARQL as the language for interacting with the database system.

4.3 Amazon Neptune

Amazon Neptune is a recent, hosted graph database supporting open source data specifications[79]. Neptune appears to have been built upon a well-known open source project BlazeGraph [80], which is used by for example WikiData and is thus thoroughly battle-tested, although there has been no official confirmation about this [81].

According to Amazon, Neptune supports queries on “billions of relationships with millisecond latency” [82]. In production cases, Neptune has been used on massive scale with up to 300 reads and 1000 writes per second [83]. As a hosted service, data backups are managed automatically. The documentation states that Neptune maintains six copies of data spread across three availability zones to minimize possibilities for data loss [84]. Query processing in Neptune is ACID compliant with strong consistency [82]. In a recent report comparing the available graph data platforms, Neptune was ranked second, right behind Neo4J [77]. To cope with high loads, a Neptune cluster can be scaled horizontally by adding up to 15 read replicas [79].

In Neptune, data is modelled as *quads*, four-position elements comprised of the subject, predicate and object familiar from RDF, plus a fourth element, the graph

[41]. Each stored triple is thus associated with a graph, either a *named graph*, identified by a URI in RDF, or the *default graph*, the union of all named graphs [85].

4.4 Proof of concept

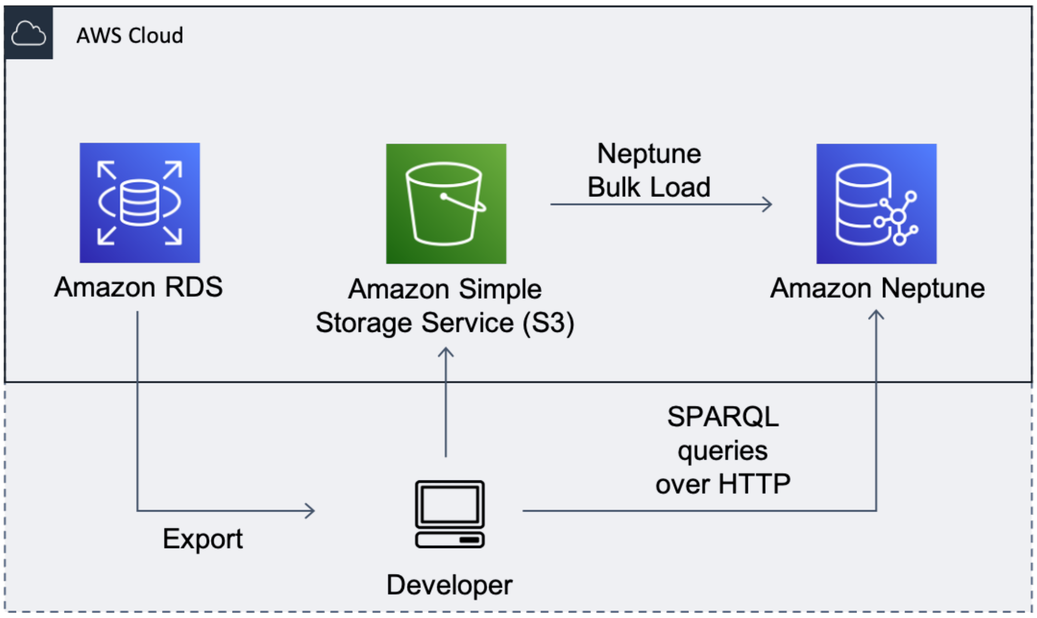
To make an informed decision about whether a migration to Neptune was feasible, a test phase was conducted as a proof of concept. The scope of the pilot was limited: to create a Neptune cluster for testing purposes, to import a subset of the data from the Content Metadata Services as RDF triples into the Neptune instance and replicate the common queries needed to implement the APIs of the Content Metadata Services in SPARQL format. It was not in the scope of the Proof of Concept to implement any API on top of the Neptune instance, or any automated data processing for that matter, but to manually test out Neptune with real data.

The architecture of the Proof of Concept is summarized in Figure 4.1. Implementing it was done in the following steps. First, a Neptune cluster and an Amazon S3 bucket were created. Second, database dumps were taken of the RDS databases of the three Content Metadata Services. These data sets were then converted into the N-Triple format, which is a serialization format for RDF data [86], and uploaded into the S3 bucket. Finally, the data was imported from S3 to the Neptune instance using the Neptune Bulk Loader functionality [87]. At this point, SPARQL queries could be performed on the data using the SPARQL endpoint of the Neptune cluster.

Case 1: Concepts and hierarchy

From the testing conducted during the Proof of Concept -phase, four use cases were identified where an implementation based on Neptune could bring benefits in comparison to the original architecture. The first of these was colloquially referred

Figure 4.1: Architecture of the Proof of Concept



to as *concepts and hierarchy* [75]. This means, in a nutshell, the way queries are simplified when all the interlinked data of the three Content Metadata Services live in a same graph database.

To illustrate, would the user want to retrieve all Finnish subject headings associated with a program, for example the film *Dances with the Wolves*, the first step would be to do a query to Relations API for all relations for the said ID (1-50203154). This will yield five relations to concept ID's, the Finnish titles of which have to be queried separately from Meta API. In total, six API calls would thus be needed. Using the graph database containing all the content metadata, a single query would suffice. The example in Snippet 2 illustrates how this looks like in SPARQL. In the example, the preliminary data model used in the Proof of Concept is used.

Case 2: Concept merge and relations

The second case tried in the Proof of Concept was called *concept merge and relations*. A relatively commonplace operation related to managing the concept data is

Code snippet 2 Retrieving the Finnish subject headings for a content ID

```
{sparql}
PREFIX purl: <http://purl.org/dc/elements/1.1>
PREFIX yleid: <http://id.yle.fi/>
PREFIX ylerelation: <http://relation.yle.fi/>

SELECT ?content ?concept ?relationType ?title WHERE {
  {
    yleid:1-50203154 ylerelation:isMemberOf* ?content .
    ?concept ?relationType ?content .
    ?concept purl:title ?title .
    FILTER (lang(?title) = 'fi') .
  }
}
```

combining concepts, when for example two semantically overlapping concepts have been retrieved as distinct concepts from external sources. In this scenario, merging is done using the Concept Editor application. The flow between different components in the case of merging two concepts is as follows. First, Concept Editor sends an API call to Meta API to combine the two concept IDs. Upon receiving the request, the Meta API application chooses one of the IDs as primary and the other as an alternative ID. After the merge, Meta API sends a change message via RabbitMQ describing the operation, which Relations API will in turn process, updating the changed relations to point to the new primary concept identifier. In SPARQL, the same result can be achieved by simply adding an *owl:sameAs* relation between the two concept IDs.

Case 3: Concept search and content hits

The third case examined during the pilot has to do with content-to-concept metrics. When querying Meta API, the resulting concept data includes a number describing how many content items have been tagged with the said concept. This data has currently been stored in a materialized view in the PostgresSQL database. The view is re-generated nightly, based on change messages received from Programs API and Articles API. In Neptune, this kind of calculation is based on relations going from concept to the content identifier and is thus efficient to compute on the fly.

What is not efficient, however, is doing full-text searches. If, for example, the user would like to query all program and article titles containing a certain word, this could be achieved by using a regular expression in the SPARQL query [88]. While the query language supports this kind of search, Neptune is not optimized for it. Consequently, such searches trigger a collection scan and are thus very slow. Fortunately, Amazon offers a solution around this where the data from Neptune is replicated into an Elasticsearch cluster, which can subsequently be used as an index for full-text searches [89]. However, testing this setup was not in the scope of the Proof of Concept.

Case 4: Combining the services

The fourth and final case examined in the Proof of Concept touched upon combining the data layers of the three Content Metadata Services. Being able to aggregate all the data needed for a use case in a single query, instead of cross-calling three distinct services, would simplify usage and maintenance. It could also allow for re-structuring of the three applications themselves into a logical whole.

Possible architectures

Finally, different possibilities for future architecture, with the Neptune cluster included, were considered. The possible architectures differed mainly in the division of labor between the microservice-specific RDS databases and the shared Neptune instance.

The first option was to migrate all data to Neptune from the RDS databases. The pros of this option are architectural simplicity and being able to access the data of all three microservices in a single query, which could potentially simplify queries and reduce latency. The downsides include the amount of work necessary to migrate all data to Neptune, transfer the functionality built on the RDS databases to work on Neptune and the potential performance surprises involved in working with new technology.

The second alternative presented for consideration in the Proof of Concept phase was splitting the data between the databases. In this scenario, part of the data of each microservice would live in the dedicated RDS database, while the rest of it would be stored in Neptune. This way, performance benefits could be gained by playing on the strengths of the different database systems — for example, by querying relations between identifiers from Neptune and doing full-text searches from the Postgres databases. The drawbacks to this approach would include complexity and difficulty of orchestration.

The third proposed possibility was using the Neptune database as a secondary data store, whilst the RDS instances would remain as master databases. In this architecture, the updates to the RDS databases would be replicated to Neptune. The state of Neptune graphs would stay in sync with the RDS databases on the principle of eventual consistency. In this model, the Content Metadata Systems could combine the different databases, optimizing performance. The downsides to this approach include cost, the complexity of managing the state of different data stores, data redundancy and potential data coherence problems.

4.5 Migration process

After the Proof of Concept, the team, the product owners and system architects discussed the alternatives. Eventually, the first scenario with the full migration from RDS to Neptune was selected, as the alternatives were considered to bring about unnecessary complexity. It was thus decided to migrate all three Content Metadata Services to use Neptune as the master data store, with the intention of phasing out the RDS instances altogether in time. However, since the services in question were already in production and had clients, the switch was planned to be gradual, and the original and renewed systems would live side by side during the transition. Essentially, the third scenario would be implemented first and converted then to the architecture of the first scenario.

Based on the experiences gained from the Proof of Concept and stakeholder discussions about the objectives of the migration, a number of steps were identified to be done in the scope of the transition [74]. First, a Neptune cluster needed to be created, populated with the data currently residing in the RDS databases and connected to the existing applications. Second, a solution for performing full-text searches needed to be implemented. After the data had been migrated and the functionality ported from the original implementation to the renewed solution, new functionality could be built on top of it. Based on these considerations, a process consisting of distinct phases with different goals, illustrated in Table 4.1, was sketched. In the phasing, Phase 1 is concentrated on setting up the infrastructure, initial data migrations and communication between the different components. In Phase 2, support for full-text searches are implemented and the data replication from the applications to the Neptune cluster is enhanced, so that both persistence solutions are in sync. In Phase 3, the RDS storage solutions are deprecated and

Table 4.1: Migration process phases

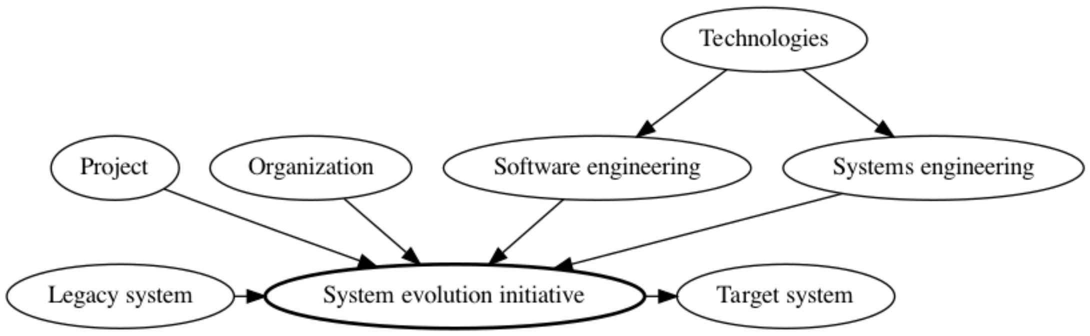
Phase	Description
PoC	Testing out Neptune with Yle's content metadata
Phase 1	Setting up infrastructure, data replication with eventual consistency
Phase 2	Implementing full-text search, synchronous data replication
Phase 3	Making Neptune the master data source, deleting Postgres instances, updating applications
Phase 4	Building new functionality, adding data graphs

the application architecture is modified to allow searches going across graphs. In Phase 4, the knowledge graph is enriched with more data sources allowing for new services. The horizontal line between Phase 2 and Phase 3 marks the demarcation between migration and development of new features. It also denotes the status of the project, as the last two phases are in progress as of this writing.

The process can be perceived in terms of the *Framework for the Disciplined Evolution of Lecacy Systems* model presented by Weiderman et al. (1997) [90]. A diagram depicting the elements involved in this model is found in Figure 4.2. In this framework, a transition from a legacy system to a renewed target system is achieved via a *system evolution initiative*, which is shaped by multiple influencing contextual forces. Here, new technology, like a graph database solution, enters the system via *software engineering* and *systems engineering* efforts. The element of *organization* marks the organizational context where the system is operated and which it serves, in this case Yle. Finally, *project* is a way to organize the development effort. In the case of the graph database implementation, the project was one task among many done by the team, also responsible for various other services and projects pertaining to content metadata.

While the model helps in making sense of the contextual factors involved in

Figure 4.2: Framework for Disciplined Evolution of Legacy Systems, applied from Weiderman et al. (1997) [90]



a systems renewal project, it reflects a sort of waterfall-model mindset. In agile development, which is the norm in Yle as in most organizations these days, it is not trivial to mark the cut point where a systems development initiative ends and a finished product starts. For the purposes of this study, it can be placed between Phase 3 and Phase 4, where the legacy implementation has been deprecated.

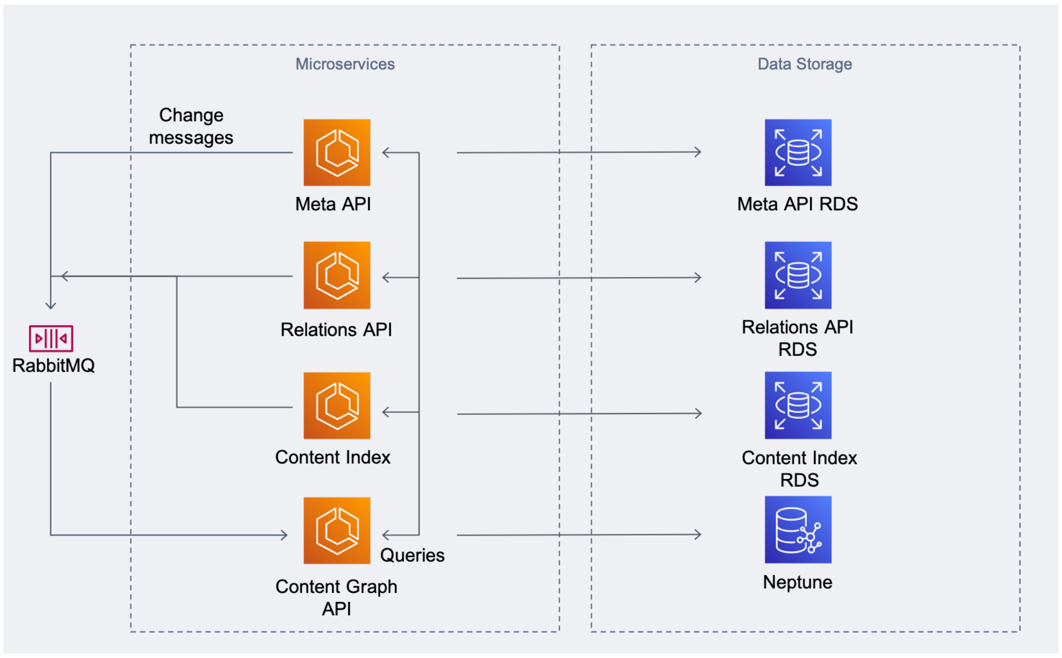
4.5.1 Phase 1

The goals of Phase 1 were to set up the Neptune clusters, for testing and production environments, to populate them with the data from the RDS instances and cascade all incoming changes to the data of the three microservices onward to Neptune as they occur. For now, the RDS databases would remain as master data stores. The architecture of Phase 1 is visualized in Figure 4.3.

Setting up the infrastructure

To shield the Neptune cluster, to control how it can be used and to decouple it from the clients using it, a fourth microservice was created to act as a gateway. This

Figure 4.3: Architecture of Phase 1



application, Content Graph API, is deployed into a Virtual Private Cloud (VPC) — not accessible from the outside — and is the only component communicating directly with the Neptune cluster. As the other microservices, Content Graph API is implemented in Clojure, exposes a REST API and is deployed as an ECS container. At its first iteration, Content Graph API exposed an API endpoint that the client services can use for performing non-mutating SPARQL queries.

As for performing updates on the Neptune cluster, the initial data migration from RDS was done, in the case of Meta API and Relations API, using a migration script. The script connects to the RDS database to process, retrieves the data dump, converts it to N-Triple format [86] using the *rdflib* Python package [91], stores it in an S3 bucket and triggers a Bulk Loader process from Neptune. As Neptune supports dividing data inside the same database cluster into multiple named graphs [41], the data originating from and owned by the different services were inserted into different graphs.

In the beginning of the migration project, different solutions were considered for validating the data to be written in the graph, as well as checking that the stored data conforms to the designed data model. While graph databases are traditionally schema-less, and their data models are open for extension, the team considered utilizing a solution such as SHACL [35] for data validation. However, this idea was later dropped and removed from the project backlog. Since Neptune does not support schema validation natively, a solution would have involved reading back data directly after a write to be able to validate it. It was decided that validating data against a data model, essentially a database schema, would be against the idea of using a graph database and implementing it would require contorting the tools available to an unoptimal degree.

Data replication

All new changes in the data of the Content Metadata Services are populated to Neptune by way of message passing. Every time one of the services mutates the data in its database, the application sends a change message to a RabbitMQ exchange [92]. Client applications interested in this data, such as Content Graph API, can then subscribe to these messages by creating a private message queue to the exchange. Each time the Content Graph API receives a change message from one of the services, it translates the change to a SPARQL statement and applies it to the relevant graph inside Neptune. This way, the changes to the master RDS databases are forwarded asynchronously to Neptune, the state of which is eventually consistent with that of the RDS instances.

Since the data model of Content Index is markedly more complex than those of the other services, its data was loaded to Neptune in a slightly different manner. After the handling of change messages of Content Index in the Content Graph API application had been implemented, it was noted that the number of different messages types was large and the logic for handling all the cases took a relatively large amount of code. Because of this, it was decided that replicating this logic to a migration script would be unnecessarily cumbersome. Instead, the bulk load phase was implemented by initiating a mass export from Content Index via RabbitMQ. This way, the contents of the whole RDS database of Content Index was exported via RabbitMQ change messages, or perhaps *pseudo-change messages*, which Content Graph API then interpreted, updating Neptune accordingly.

For the data in the Neptune cluster to be queryable for clients, new versions of the APIs of the three Content Metadata Services were implemented. For each of the three microservices, an API version 2 was published. Each new API replicated the functionality of the original API implementations on top of Neptune. An example of the flow between the components is given below for a basic use case.

Code snippet 3 SPARQL template for an ancestors-query into Neptune

```
{sparql}
SELECT ?id ?altid ?type ?editorialObjectType
WHERE {
  GRAPH <http://content-index.yle.fi/>
  {
    {
      <http://id.yle.fi/{{id}}> (skos:member)+ ?id .
      OPTIONAL { ?id owl:sameAs ?altid }
      OPTIONAL { ?id rdf:type ?type }
      OPTIONAL { ?id ebucore:editorialObjectType ?editorialObjectType }
    }
  }
}
```

1. A client application calls the API version 2 of Content Index, asking for all

the *ancestors* of an identifier.

2. The call passes through the AWS Api Gateway and load balancer to the Content Index service and triggers a handler function inside the application.

3. A SPARQL query is parsed based on the query parameters; the requested ID and the path parameter “ancestors”. The application reads a query template file, which looks as shown in Snippet 3, parses it into a query by placing the ID parameter between the double curly braces, URL encodes it and calls the */query/* path of the Content Graph API with the encoded query string as parameter.

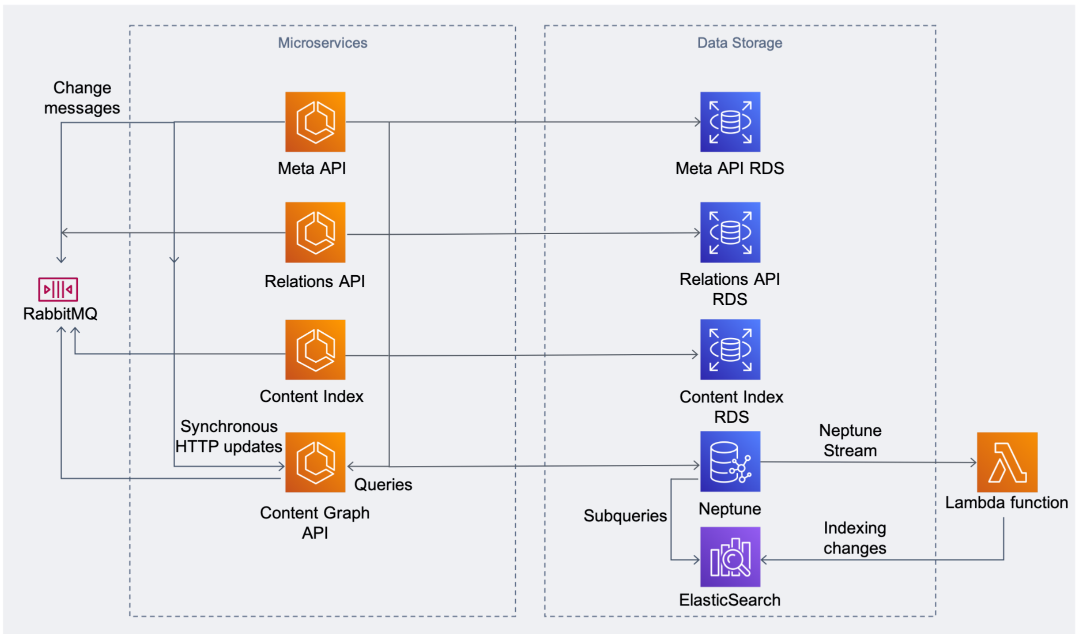
4. Content Graph API receives the call, extracts the query string and performs the query on the Neptune database. In the query itself, relations of type *skos:member*, which are defined from lower-level IDs to those higher in the hierarchy, are traversed along with some optional properties associated with them. The response from Neptune is returned as it is.

5. Content Index receives the response, in JSON format, parses it first into EDN, which is the native data format for Clojure [93], and then to the same format as the original API, before returning it to the client as JSON.

4.5.2 Phase 2

Two main goals were defined for Phase 2 of the migration: implementing support for full-text searches and making the replication of data from the microservices to Neptune synchronous and independent of the RDS databases. The architecture of Phase 2 is visualized in Figure 4.4.

Figure 4.4: Architecture of Phase 2



Full-text search

As for the full-text search, a solution extending the functionality of Neptune by using an Elasticsearch cluster as an additional search index was available from Amazon, as noted in Section 4.4. This approach utilizes a recent feature of the Neptune database called Neptune Streams [94], which was only consolidated as a stable feature of the Neptune engine from an experimental Neptune Lab Mode [95] status two months prior to its adoption in the project. Neptune Streams is a change-log stream of the database, which can be read by clients using an API over HTTP [96]. It is possible to index all changes to the database to an Elasticsearch cluster using a Lambda function that reads data from the Neptune Stream and writes it to Elasticsearch. To achieve this, a ready-made CloudFormation template is available [97].

After the Elasticsearch integration was set up, the data from the RDS databases needed to be re-imported into Neptune for the full-text index to be up-to-date. Although an experimental solution for exporting existing data from Neptune to Elasticsearch exists [98], setting it up seemed too time-consuming, and the team opted to use the existing import scripts and mass export functionality of Content Index instead.

With this setup, queries with text-search components utilizing both Neptune and Elasticsearch can be defined seamlessly in SPARQL. The full-text searches can be implemented as sub-queries directed to the Elasticsearch cluster [99]. The sub-queries return IDs that can be used by the rest of the query. In hybrid queries like this, Neptune can call Elasticsearch directly, provided that the necessary parameters, such as the address of the Elasticsearch cluster, are provided in the query and that the Neptune instance has been granted the necessary IAM permissions.

Synchronous updates

The second goal of Phase 2 was making the updates from the applications to Neptune happen synchronously and independently of the RDS databases. In Phase 1, the microservices processed changes by first updating the RDS database and publishing the change via RabbitMQ — in Phase 2, the updates are done without delay over HTTP to Neptune, via Content Graph API. This way, reads and writes to Neptune are close to being strongly consistent. For this to be possible, the API of the Content Graph application was updated to process updates and deletions in addition to the non-mutating queries of the first phase. To maintain hygiene between the different graphs, no generic update path was implemented. Instead, each microservice has its own update path in the API, which can only mutate the graph owned by the client application. As a consequence, logic for updating the Neptune database was moved upstream from Content Graph API to the individual applications — while in Phase 1 the Content Graph API needed to be able to interpret change messages from the applications and translate them to SPARQL statements, in Phase 2 the parsing is done in the applications and Content Graph API only calls Neptune on the instruction it receives.

Outcomes

As Phase 2 was concluded, the functionality of the three Content Metadata Services had been re-created on top of the Neptune database. The original RDS databases were still running, up-to-date and queryable via the original API paths, as all clients of the services still did at this point. However, the new API versions replicated this functionality on top of the new implementation, fetching the data from the graph. At this point, no real benefits had yet been gained, as the structure and the public APIs of the three Content Metadata Services were exactly the same as before. However, the next phases of the project, which as of this writing are underway, are all about enhancing the overall system and leveraging the benefits from the graph database.

4.5.3 Phase 3

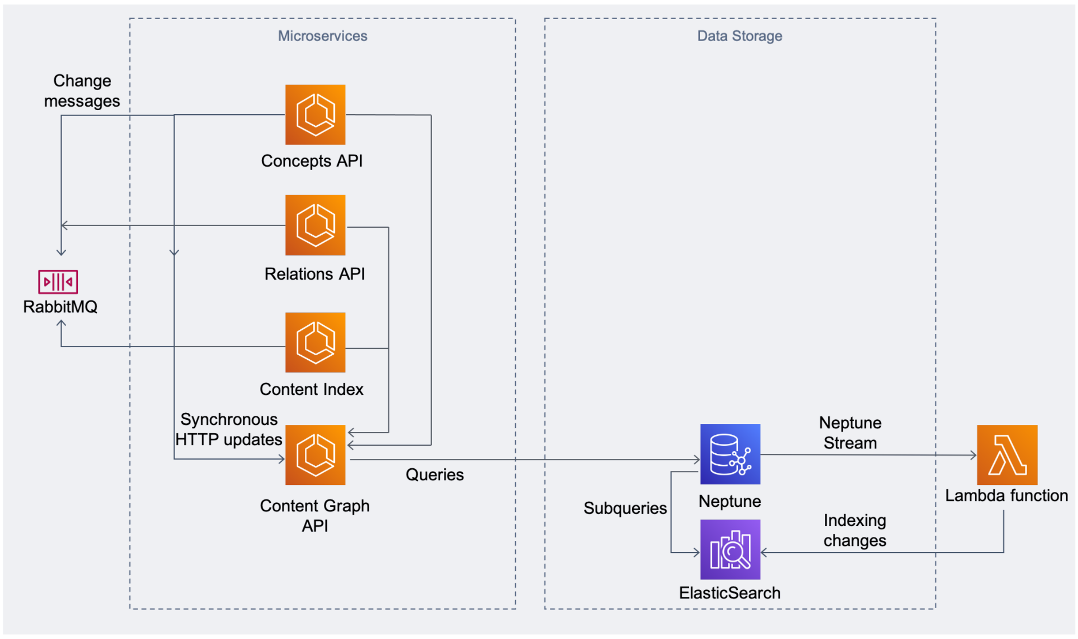
Phase 3 has three main goals defined for it: making Neptune the master data source, deleting the RDS databases and enhancing the application structure of the Content Metadata Services to be better aligned with the objectives of the project. The architecture of Phase 3 is depicted in Figure 4.5.

Making Neptune the master data source

In order to be able to delete the RDS instances of the services, the original APIs of the applications need to be deprecated. Before this, all clients using the APIs have to be instructed to move to the new API versions. For a transitional period, both APIs will be supported and used simultaneously. In the end, the RDS instances will be deleted, leaving Neptune the only database for the Content Metadata Services.

An alternative to disabling the original APIs altogether would be to redirect then to point to the new API implementations. Since the functionality and usage of the APIs are identical to the earlier versions, this change would in principle not be noticeable to the clients. In addition, it would follow the *API Versioning and*

Figure 4.5: Architecture of Phase 3



Backwards Compatibility principle outlined before in Section 2.1.3.

Enhancing the application architecture

To be able to reap benefits from having the data of the three Content Metadata Services in a same graph database, the structure and division of labor between the applications needs to be updated. The first step towards this is deprecating the Meta API and replacing it with a new service, Concepts API. This application is essentially a rewrite, or the next version of Meta API, written and deployed in tandem with the Neptune migration. It will allow queries for concepts and relations as the two services did before, but using Neptune. Concepts API will also support queries crossing over the two domains; for example, when querying relations for an ID, all the information related to the concepts can be returned along with concept ID. This way, the number of HTTP queries can be cut down as was one of the original goals for the migration.

4.5.4 Phase 4

Phase 4 of the project does not have clearly defined goals as of now. Rather, it marks a starting point for building new functionality and applications on top of the services and infrastructure created in the previous phases. Two new services are currently in the proof of concept -phase: Author API and storing location data into Concepts API. Both have been planned and discussed with the idea that the Neptune cluster serving the Content Metadata Services will become a master store for different types of data that are currently fragmented into different systems. This way, the data assembled into the Neptune database would be gradually refined into an *Enterprise Knowledge Graph* [100], a centralized, interconnected data structure connecting data from different domains of the organization.

For the time being, data about people creating content, such as journalists, photographers and illustrators is dispersed throughout the company. This data is found in content management systems, publishing systems and different administrative systems. The *status quo* is considered to be sub-optimal and discussions have been conducted about how this data could be stored and managed in a centralized manner. The working hypothesis has been that a new application, currently known informally as Author API, would be built on top of the Neptune instance, providing a centralized entry point to this data. Many questions are yet to be answered, though, from how to identify persons reliably from the data to how and by whom it should be managed. Processing personal data also brings strict data security and GDPR compliance requirements into the equation.

Another future service to be built on the Neptune database is a service for managing location data. Currently, different departments in the company are managing location data for their own use cases. In recent discussions on the subject, a need for an internal service managing and serving location data for the different use sites has been identified. As for the Author API, this has been planned to be developed using Neptune, since querying location data is one of the classical uses cases for RDF and graph databases [33].

A third possible avenue for investigation is the ontologization of concepts used as subject headings. Since the concepts originate in semantically structured sources, such as WikiData [69] and Finto [70], the relations found in the source ontologies can be used instead of encoding them manually, which would be laborious [47]. One possible route to achieving the three objectives outlined above would be to widen the understanding of the term *concept* in the context of content metadata, by handling the geographical concepts and author data as concepts used as subject headings. Regardless of what the implementation will be, semantic relations between the concepts allow for more intelligent search services and better utilization of the existing metadata.

4.6 RQ2: A process for migration

In the preceding sections, a migration of three microservices from service-specific relational databases to a shared graph database containing named graphs for each application has been described. To conclude the part of the thesis dealing with the case of the study, an answer to RQ2 — *How can microservices be migrated from a relational database to a graph database?* — remains to be addressed.

The migration process of the case project comprised of a number of consecutive major phases, enumerated below.

1. Phase 1

    - Graph database cluster initialization and setting up the infrastructure.
    - Replication of the data to the graph database along with the earlier data store.
    - Migration of historical data from the earlier data store to the graph.

- Implementing a functionally identical new version of the API on top of the graph database.

2. Phase 2
   - Implementing support for full-text search.
   - Implementing synchronous data replication.
   - Advising users to move to the new API.
3. Phase 3
   - Enhancing the application infrastructure, supporting cross-graph queries.
   - Redirecting the original API (v1) to the new API (v2).
   - Deleting the old persistence solutions.

Gradual transitioning to the new database and data model was a suitable approach for the case project. The transition to Neptune for the Content Metadata Services was an experimental effort for the company, where learning and experience were valuable results along with the actual revamped version of the system. Due to these circumstances, the team had a possibility to experiment, try different approaches, learn on the go and find a solution that works well for the use case. Since the migration project dealt with production systems, the transition to the target architecture needed to be gradual and the original implementations had to be supported until the renewed implementation was comprehensively tested. However, the process by which the transition was achieved is the result of iterative work in agile sprints and is only obvious and detailed when viewed with the benefit of hindsight.

While the migration of the services to work on top of a new persistence solution was successful, the main functional requirement for the project — a system working with less HTTP calls — has not yet been reached. At this point, Phase 1 and Phase 2 of the project, dealing with the migration to Neptune, have been completed and the latter ones are in still in progress. As the published APIs are identical to the original implementation, the usage of the Content Metadata Services works as before, including the amount of requests. One aspect where using a graph database benefits the current setup is data maintenance. As noted in the *concept merge and relations* use case identified in the Proof of Concept, described in Section 4.4, for example combining synonymous concepts necessitates far less ceremony when using a graph database as opposed to the original setup. The non-functional requirements like performance will be discussed in the next chapter.

While the described process is *a* way to migrate to graph database, it is obviously not *the* way to do it, as the procedure applied was designed for a specific use case. In any case, it is interesting to evaluate how widely the experiences gained from this project can be generalized. First, the process of gradual transitioning by first replicating data to the new persistence solution, implementing an API to use it, testing it and then making it the master data store is likely to be applicable for other production microservice systems. The architectural pattern emerging from the project, where a gateway application is used to mediate traffic to a shared graph database cluster, is an approach that could be considered when implementing similar projects in other contexts, as the gatekeeper application can provide security and data hygiene guarantees by restricting mutative operations to the graph owned by the client performing them. Based on the experiences gained from this case, it can be proposed to be used in contexts where maximising performance is not the main requirement.

5 Evaluation

This chapter starts the retrospective part of the thesis. In this chapter, the renewed architecture is evaluated and discussed. First, the performance of the two implementations are compared. After this, feedback gathered from the project retrospective session is presented and discussed. In terms of the DSRP model as outlined in Table 1.1, this chapter corresponds to Phases 4 (Demonstration) and 5 (Evaluation). In both sections of this chapter, these phases are intertwined. Both in performance evaluation and the retrospective session, metrics and analysis knowledge is first generated and then evaluated and discussed. Evaluation, as in the DSRP model, is further continued in the next chapter.

5.1 Performance benchmarks

Performance of the original and renewed implementation of the application architecture were compared by running two kinds of programmatic tests comparing the two API versions of the Content Metadata Services, using the Content Index application as a test case. First, a generic comparison of response times for different types of queries was performed. Second, a load test scenario was developed and run for the application, using both API implementations.

Going into the comparison, it was hypothesized that the two implementations would probably have minor differences in performance. This was partly due to the initial experiences gained at the Proof of Concept -phase [75]. The renewed implementation, or API v2, was expected to respond slightly slower for simple queries, partly due to the network latency of two extra HTTP calls, first to Content Graph API and onward to Neptune. For more complex queries, or queries where a large number of relations are traversed, API v2 was expected to give better performance. Before the overall architecture of the Content Metadata Systems is re-organized into utilizing the graph more fully for queries, instead of the current heavy cross-calling, a slight performance drop is to be expected. The main goal of the performance comparison is to validate that the performance of the new system is acceptable and will not impose a significant performance penalty compared to the original state.

5.1.1 API response times

API response times comparison was performed for the Content Index application. The data on API response times was collected as follows. First, a sample of IDs was gathered from the database of the testing instance of Content Index. The sample size n was 1000 for each identifier type except *cost center* — since the data contained only 371 distinct cost centers, all were included in the test. The selected IDs were saved into text files, each of which contained only one type of IDs.

Second, a Python program was implemented for performing and comparing queries on both APIs. It processed the sample IDs, performing the following steps for each:

1. Check the type of the ID. For *program* and *article*, perform an ancestors-query, for *cost center* perform a descendants-query, for other types perform both.
2. Parse the API query strings based on the ID. They are of the format *https://ENDPOINT/API-VERSION/QUERY-TYPE/ID&API-KEYS*.
3. Perform both queries, measuring the response times.

4. Save the responses, along with the response times, into a local PostgreSQL database.

5. Compare the responses field-by-field. When noting any discrepancies between the responses, such as IDs present only in one of the responses, save the differences to another database table.

After the data was collected, the response times comparison was formulated using an SQL query. For the results to be comparable, only the queries where both APIs returned an identical, non-empty response were included in the comparison. The results are collected in Table 5.1. Here, *v1* denotes API version 1, the original implementation using the PostgreSQL database, while *v2* is API version 2, the renewed implementation based on Amazon Neptune. For each query type, the mean, the standard deviation, the median, the minimum and the maximum are reported as to the response times. The recorded response times are in seconds. In these figures, smaller is better.

Table 5.1: Performance comparison

Id type	Query type	n	v1 mean	v2 mean	v1 stddev	v2 stddev	v1 median	v2 median	v1 min	v2 min	v1 max	v2 max
Cost center	descendants	86	0.233	0.215	0.123	0.058	0.206	0.202	0.128	0.121	1.238	0.42
Product	ancestors	573	0.36	0.343	0.429	0.332	0.25	0.268	0.099	0.11	5.819	6.13
Product	descendants	207	0.341	0.433	0.304	0.379	0.27	0.331	0.102	0.131	3.734	3.338
Program	ancestors	131	0.188	0.22	0.042	0.136	0.184	0.199	0.066	0.127	0.33	1.254
Project	ancestors	985	0.266	0.287	0.229	0.179	0.211	0.243	0.095	0.109	5.198	2.152
Project	descendants	710	0.269	0.299	0.157	0.183	0.229	0.258	0.1	0.108	2.306	2.51
Season	ancestors	307	0.259	0.27	0.146	0.093	0.221	0.243	0.103	0.124	1.463	0.628
Season	descendants	565	0.272	0.306	0.164	0.143	0.222	0.266	0.107	0.12	1.521	1.304
Series	descendants	450	0.196	0.231	0.054	0.068	0.189	0.216	0.107	0.125	0.633	0.783

The results seem to support the hypothesis that the API version 1 would be faster for simple cases, whereas API version 2 would outperform it in more complex queries. While the differences are minuscule, there is a slight performance difference in descendants-queries for cost centers, which is by far the most relation-rich of the different query types. For these queries, API version 2 is on average 8 % faster than the original implementation, despite the added network latency of two extra HTTP calls. Interestingly, also ancestor-queries for products are also slightly faster in API version 2. This is unexpected, since queries going upwards from product should only return a single project and cost center and are thus by no means highly connected. Looking more closely at the raw data, as well as the maximum times at the breakdown, reveals that the difference seems to be due to a handful of outliers in the data. Removing the queries where execution takes longer than 1.5 seconds from either API ($n = 34$) brings the means down to 0.307 for API version 1 and 0.315 for version 2. The outliers were probably due to some momentary slowdown in performance, for example during load spikes, instead of any systematic performance bottleneck in the implementations. This was confirmed by re-running the outlier queries on a later date, where the response times were closer to the mean.

While the data is enough to confirm that the performance of API version 2 is not considerably worse than that of the original implementation, and it thus fills the performance requirements set for it, there are some potential sources of error in the measurements. First, the number of comparable queries is small, especially so for queries with cost center IDs. This is due to the fact that at the time the performance data was collected, slight discrepancies between the responses of the different APIs occurred. For instance, only 86 of the 371 descendants-queries for cost centers returned identical results from both APIs and were thus included in the comparison data. There are two primary reasons for this: state for the data and implementation details. First, as the data was gathered from the testing instance, since no production environment existed, the data updated to the two database instances can have been out of sync at times due to momentary errors, such as failures in the message passing between the services. Second, while the both APIs were designed to format their responses similarly, some elements can have been formatted slightly differently and consequently excluded from the comparison. For instance, the identifier type *product family* is in an experimental state. More specifically, product families were actually lacking proper IDs. While this data was stored into Neptune, the API version 2 did not serve them at the time the tests were run.

Another question altogether is how the different components of the system affect performance. For instance, response to a query to the API version 2 is returned from Neptune to Content Graph API as JSON, which is then parsed into the EDN, the Clojure-specific exchange format, in Content Graph API and the parsed result is again converted into JSON that the client receives in his response. For very large result sets, this parsing overhead can potentially influence the overall performance — and undermine the potential performance gains provided by the Neptune database for complex queries. However, scrutinizing and optimizing this kind of performance details were not in the scope of the test.

5.1.2 Load tests

Another automated testing that was used to evaluate and compare the two different implementations were load tests. The purpose of running load tests on the system were getting data about how the applications perform under heavy traffic and whether this information could be used to optimize the performance. To do this, the open-source tool Gatling [101] was used. Gatling is a popular load testing framework written in Scala, first published in 2012 [102]. In Gatling terminology, a test suite, technically a Scala class, is called a *simulation* [103]. A simulation consists of one or more *scenarios*, which are essentially test cases. A scenario mimics a user interaction with the system, where multiple simulated users are simultaneously performing similar operations. Scenarios are implemented using a custom Domain-Specific Language [103]. Gatling uses the Akka [104] and Netty [105] toolkits to simulate simultaneous users [106]. Again, Content Index was used for testing the performance. As with the generic response time comparison, the load tests were run against both API implementations, version 1 and version 2, to get insight into how the transition into Neptune-based system affects performance under load.

The load test scenario performed on Content Index consisted of queries, in both directions, using identifiers of type *project*. In the scenario, the core functionality of which is shown in Snippet 4, project identifiers are read from a file, parsed into query strings and saved into the Set *pathSet*. In the scenario proper, a *feeder* [107] is created from the input data, which is then fed as an input to an object *GetHierarchy* that performs the API calls using the method *get*. Two parameters are modified between runs: *apiVersion*, which controls whether the API using the original or renewed implementation is used, and *concurrentUsers*, which controls the amount of simulated users. The tests were run from a local computer. Caching of responses in the API gateway was disabled by setting the *Cache-Control* header to *no-cache* [108].

Table 5.2: Comparison of load test results

Executions					Response times (ms)							
API version	Users	Total reqs	Failed	Reqs/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Stddev
1	10	11196	0	92.529	56	67	103	272	339	13914	107	201
2	10	6114	2	49.306	68	122	187	334	589	60003	197	1181
1	50	44201	0	337.412	56	89	145	306	613	18346	136	249
2	50	7765	3	51.424	91	639	918	1614	2076	60002	778	1315
1	100	45770	0	360.394	56	129	283	906	1904	18570	263	446
2	100	8289	3	55.26	432	1229	1726	2648	3237	60006	1455	1376

On the first iteration, the scenario was run six times with varying settings. The results are found in Table 5.2. Starting from the left side, the table contains data about the API version used, the number of concurrent users enabled in the scenario, the total number of requests performed and the request throughput — the average number of requests performed per second in the simulation. The section in the right side contains statistics, in milliseconds, about response times in the scenario.

Code snippet 4 Gatling load test scenario for Content Index

```
{Scala}

  object GetHierarchy {

    val get = exec {

      http(s"$${query}")

        .get(s"$${query}")

    }

  }

  val getHierarchiesScenario = scenario("Get hierarchies for projects")

    .during(duration seconds) {

      val queryFeeder: Feeder[String] =

        Stream.continually(pathSet.map(t => Map(("query", t))))

          .flatten.toIterator

      feed(queryFeeder)

        .exec(GetHierarchy.get)

    }
```

The results suggest a significant difference in performance under load between the implementations. Based on the simulation runs, the original implementation of Content Index can handle an average number of 360 requests per second, while the mean in API version 2 is around 50 queries. Even then, singular requests result in timeouts, as can be seen from the *Max* column, as 60 seconds is the timeout limit. A possible explanation for the timeouts could be the garbage collection process running in the Java Virtual Machine of running the Content Graph API, a nuisance reportedly experienced by teams developing API services at the company. It is also noteworthy how responses from API version 2 start to take longer when the amount of concurrent users rises. In the case of the test run with 100 simulated users, the average response time for API version 2 is almost 1.5 seconds, which compared to the 0.26 seconds from the original API is dramatically slower. Under such load, even the minimum response time rises to 0.43 seconds for API version 2, whereas for API version 1 this metric stays constant throughout the runs. Curiously, a number, albeit a small one, of queries fail and result to timeout on each test run against the API version 2.

Based on these observations, a number of optimizations were implemented. First, on the infrastructure level, the Neptune cluster, originally consisting of one writer and a single reader instance, was scaled up horizontally by adding a new read replica to it [109]. On the application level, the Content Graph API application was modified to direct all non-mutating queries to its *reader endpoint* whereas all operations were previously pointed at its *cluster endpoint* [110]. A related application-level optimization was changing the logic for parsing SPARQL queries to be proxied to Content Graph API inside the Content Index application. As described in Section 4.5.1, Content Index parses SPARQL queries by placing parameters into query templates. Originally, these templates were stored on disk inside the Docker container the application runs in and read from file on each request. Since reading from disk is relatively slow, the read latency can theoretically form a performance bottleneck under extreme load. Due to these considerations, the handling of the template files was changed to be done only once at compile time, after which they are stored in memory for faster access.

After the optimizations, the load test scenario was re-run. Table 5.3 contains the original load test data augmented with results for running the scenario on API version 2 after the optimizations. These figures are found under the API *2 (opt)*. By comparing the results to the previous runs, it is evident that the optimizations boosted the performance of API version 2 significantly. For instance, with 100 simulated simultaneous users, the performance of API version 2 was previously drastically hindered, while after the optimizations for example the minimum response times are

Table 5.3: Comparison of load test results, with API v2 (opt) included

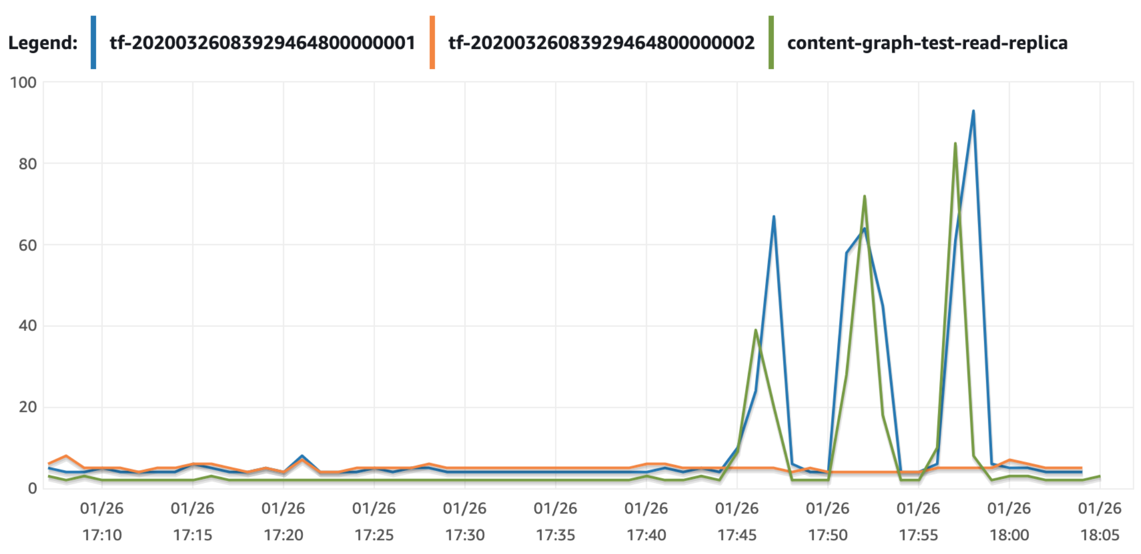
Executions					Response times (ms)							
API version	Users	Total reqs	Failed	Reqs/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Stddev
1	10	11196	0	92.529	56	67	103	272	339	13914	107	201
2	10	6114	2	49.306	68	122	187	334	589	60003	197	1181
2 (opt)	10	8059	3	52.331	69	89	112	201	462	60004	154	1245
1	50	44201	0	337.412	56	89	145	306	613	18346	136	249
2	50	7765	3	51.424	91	639	918	1614	2076	60002	778	1315
2 (opt)	50	14352	7	81.585	67	273	477	1009	2009	60004	428	1506
1	100	45770	0	360.394	56	129	283	906	1904	18570	263	446
2	100	8289	3	55.26	432	1229	1726	2648	3237	60006	1455	1376
2 (opt)	100	13425	6	100.985	68	626	966	2670	5528	60006	913	1737

consistent along the runs.

While the aforedescribed load tests give insight into the performance of the system as a whole, they fail to illuminate where the performance bottleneck lies. This also did not become immediately evident by monitoring the different components in the system during the test runs. For example, the CPU usage of the Neptune cluster did not reach full capacity, as can be seen in Figure 5.1. To get data about this, one more test run was performed. This time, the load test scenario was run directly on the Neptune reader endpoint, without layers of indirection in between. Data about the performance of Neptune under load gives some indication about how much of the performance drop from API v1 to v2 is due to the database itself and what is the role of the different applications.

In order to run the tests straight to the Neptune cluster, which lives inside a Virtual Private Cloud that is not open to the internet, an SSH tunnel was opened to the reader endpoint in the cluster via a jump server, through which the cloud resources can be accessed at Yle. The tunnel was bound to a localhost port, which was then used as the endpoint for the queries. The simulation was modified to parse

Figure 5.1: Neptune CPU usage under load tests



the test IDs into SPARQL queries, which were then sent to the cluster in the body of a HTTP POST request with the Content-Type header set to *application/sparql-query*. The responses are gathered on the final Table 5.4 under the API *Neptune*. For readability, data from the previous runs are included as well.

The results indicate that the differences between API version 2, after the optimizations, and plain Neptune are not as great as one might expect. For instance, their response times in the simulation with 100 concurrent users are quite near each other. Unlike with test runs on API version 2, queries to the plain Neptune instance do not result in timeouts, although a handful of responses come in very slowly, taking up to 47 seconds to complete. Even so, the amount of very slow responses is small, as the 99th percentile is nowhere near these numbers. One area where plain Neptune clearly outperforms API version 2 is the amount of queries it can handle simultaneously. According to the simulation, the average number of queries plain Neptune can process in a second is over 200, while this metric is 100 queries for API version 2 after the optimizations. Even the latter is a high number which far exceeds the traffic the Content Metadata Services currently receive. User experience reports

Table 5.4: Comparison of load test results with API v2 (opt) and Neptune included

Executions					Response times (ms)							
API version	Users	Total reqs	Failed	Reqs/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Stddev
1	10	11196	0	92.529	56	67	103	272	339	13914	107	201
2	10	6114	2	49.306	68	122	187	334	589	60003	197	1181
2 (opt)	10	8059	3	52.331	69	89	112	201	462	60004	154	1245
Neptune	10	24497	2	204.158	60	548	908	2240	3796	47044	856	1953
1	50	44201	0	337.412	56	89	145	306	613	18346	136	249
2	50	7765	3	51.424	91	639	918	1614	2076	60002	778	1315
2 (opt)	50	14352	7	81.585	67	273	477	1009	2009	60004	428	1506
Neptune	50	25285	2	210.708	60	549	883	2403	5442	43585	829	1365
1	100	45770	0	360.394	56	129	283	906	1904	18570	263	446
2	100	8289	3	55.26	432	1229	1726	2648	3237	60006	1455	1376
2 (opt)	100	13425	6	100.985	68	626	966	2670	5528	60006	913	1737
Neptune	100	25405	2	211.725	59	469	819	2254	6770	46735	825	1318

have indicated that Neptune is able to handle up to 300 reads and 1000 writes per second with 75 millisecond response times [83]. This is in line with the simulation in terms of throughput, but not in terms of latency. Exactly why higher latency is experienced in the simulation is difficult to evaluate, but setup, type of queries and cluster settings, like enabled capacity, may play a role. In any case, for the present discussion, the exact performance details of Neptune are not the main focus, but the notion that the API applications built upon it seem to limit the throughput Neptune can handle, but do not contribute significant latency to the requests.

Table 5.5: Resource details for load test runs

API version	Number of containers	Container memory	Database	Database class	Database RAM	DB vCPU	Database engine version
1	2	2048 MB	Amazon RDS PostgreSQL	db.m4.large	8 GB	2	11.1
2	2 + 2	2048 MB + 1024 MB	Amazon Neptune	db.r5.large, db.r5.large	16 GB + 16 GB	2 + 2	1.0.2.1
2 (opt)	2 + 2	2048 MB + 1024 MB	Amazon Neptune	db.r5.large, db.r5.large, db.r5.xlarge	16 GB + 16 GB + 32 GB	2 + 2 + 4	1.0.2.1
Neptune	-	-	Amazon Neptune	db.r5.large, db.r5.large, db.r5.xlarge	16 GB + 16 GB + 32 GB	2 + 2 + 4	1.0.2.1

Table 5.5 lists the resources available for each of the load test runs. Starting from left, the table lists the API version, the number of container instances, the amount of memory allocated to the containers, database type, database class in

AWS terminology, the amount of RAM allocated for the database, the number of virtual CPUs in the database instance and the database engine version. Since two applications are involved in the API version 2, namely Content Index and Content Graph API, the resource details of both are reported, in this order. In the database class property for the Neptune cluster, the first value denotes the writer instance while the subsequent values are for readers.

From the tests it becomes obvious that the renewed implementation performs notably worse under heavy load than the original system based on the PostgresSQL database, but what are the implications of this? For one thing, this gives extra impetus to cut down the number of HTTP calls the Content Metadata Services perform across each other, even though it is unlikely that the limits of the system are going to be reached in the foreseeable future. To achieve this, the possibilities of having all the data of the three Content Metadata Services in the same graph database cluster should be utilized more fully. This means for example offering clients more "intelligent" API endpoints, where all the data pertaining to a user need could be pulled using a single request. Even though premature optimization is among the most infamous anti-patterns in software development [111], in terms of future-proofing it is good to be conscious of the limitations of a system, especially as the Neptune cluster sitting behind Content Graph API is planned to act as a central data store for many stakeholders in the long run.

5.2 Project retrospective

A project retrospective session for the graph database migration was held on 14.12. 2020. Participants of the post-mortem included developers involved in the different phases of the project, current and previous product owners, systems architects and a facilitator. While the graph database migration was not done in isolation, but concurrently with the development of multiple other services the team the developers were working in, and was consequently discussed in the team-level retrospectives in the course of normal sprint cycles, the migration debriefing was devoted entirely to this project and a wider array of stakeholders were present.

In agile software development, retrospective sessions are considered as "collective learning activities" [112], where the team can reflect the experiences and problems related to a development project, adjusting its practices accordingly. In a retrospective session, three steps can be identified: *target definition*, *reflection* and *corrective action development* [113]. The objectives of the session followed this structure closely: the aim of the meeting was to evaluate the choices made in the project (reflection), how the new implementation fits the original requirements (target definition) and what should the next steps be going forward (corrective action development).

According to Myllyaho et al. (2004), the benefits of conducting project post-mortems include 1) *helping team members share and understand each other's perspectives*, 2) *integrating individual and team learning*, 3) *identifying hidden problems*, 4) *documenting good practices and problems*, 5) *increasing job satisfaction through feedback* and 6) *improving project cost estimation* [114]. Of the six points in their model, the first three were explicit goals for the session: the session aimed at forming a shared understanding of the project and the potential of generated system, to identify its shortcomings and potential drawbacks and to create a vision of its future.

The session was conducted online and the input from participants was collected using Google Jamboard whiteboard software. The discussion was structured around four questions, which are discussed in the following subsections. The full answers to the questions, in the original Finnish, are found in Appendix B.

5.2.1 What was the original problem?

The first question discussed in the retrospective concerned the understanding of the original goals for the project. In this context, it was considered more interesting to discuss how the different participants understood these goals instead of how they were defined in the project documentation. From the answers and discussion, four themes can be identified: *competence building*, *data modelling and management*, *performance* and *architecture*.

Regarding competence building, it was stressed that the project was experimental in nature and one of the goals was to gain experience and insight into how graph databases could be utilized at Yle and what opportunities they could provide. It was noted that an earlier experiment had been conducted as a proof of concept, using Neo4J, but this project was terminated before it made it to production.

The points made regarding data modelling, data management and performance were conceptually intertwined. On the one hand, it was understood that the data in the three Content Metadata Services would form a graph or a network and be rich with relations. This being the case, a graph database seemed fitting for the use case both in terms of modelling the data as an RDF graph instead of using a relational approach, and that a graph database could potentially give better performance for complex queries.

In terms of systems architecture, an idea was expressed regarding the potential of the graph-based approach for managing product data in the long term. Although the scope of the data managed in the Content Metadata Systems is limited, it could be expanded to form a centralized Enterprise Knowledge Graph [100] [115] or Data Catalog [116], which could act as an entry point for all content resources. This would entail migrating more resources into the Neptune instance, finding ways to make the data model more interconnected, as opposed to the strict separation of graphs as in the earlier implementations, and building new services on top of it. It would also mark a wider shift in the systems and data architecture levels at the company.

5.2.2 What was the most important thing you learned?

The second question encouraged the participants to reflect on the learning achieved in the project. Answers centered around two main themes: *learning curve* and *scope of the project*.

By far the most common point made in the answers concerned the learning curve involved in the transition from relational to the graph database. The migration was said to have taken much longer than was originally expected and it had involved learning a lot about the new data formats and technologies. One related issue in the answers was that the tooling for managing the graph database were not up to par with those designed for relational databases and that transitioning from one to the other had crystallized the differences.

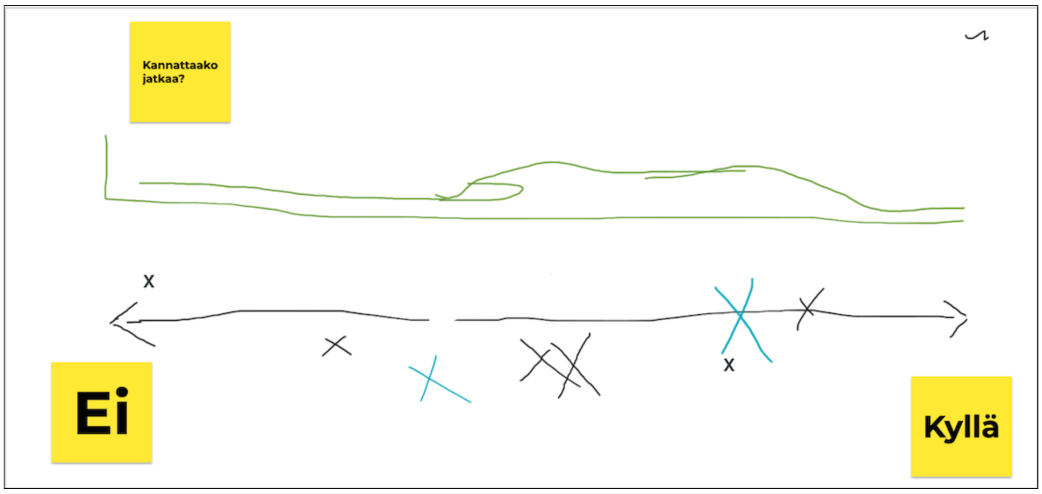
The other point made in the answers was that it is questionable how well the potential of the graph database can be evaluated based on the current use cases. The scope of the project was seen as limited, although one comment stressed that it had made clear that a graph database is a valid alternative for production use and “not a toy”.

5.2.3 When would you choose a graph database now?

The third question was about the kinds of use cases the participants would choose a graph database for, now that they have some practical experience and insight about them. The answers varied a lot in content and scope.

A single answer was sceptical and read: “With the current levels of competence and productization, probably not for anything”. The other comments were more optimistic. Multiple people mentioned data rich with relations as one criterion along with related queries traversing the relations. Two answers mentioned also

Figure 5.2: Responses to the fourth question



that they would only pick a graph database to use with a new application written from scratch.

5.2.4 Is the project worth continuing?

To conclude, the question *"Is the project worth continuing?"* was discussed. To elaborate, the idea of the question was to discuss whether the participants felt like it made sense to continue developing the solution built atop the graph database or if going back to the original implementation would still be advisable. The participants were asked to put a mark somewhere on an axis from *no* to *yes*.

As can be seen from Figure 5.2, the answers spread from very pessimistic to quite positive, falling mostly on the positive side. One participant expressed his feeling towards the matter by drawing a distribution of sorts, also leaning clearly in favor of continuing.

6 Discussion

This chapter concentrates on discussing the findings and perspectives emanating from the case study and the preceding theoretical discussion. Insights from the practical work on the migration project and experiences using a graph database in production are presented.

6.1 Fulfillment of requirements

When the graph database migration project was initiated, both functional and non-functional requirements were defined. The main functional requirement was defined to be reduction of complexity in the Content Metadata Services as a whole and decreasing the amount of HTTP calls. A non-functional requirement of better overall performance would be attained as a consequence — foremostly by achieving the use cases with less requests, secondly by making complex queries traversing a large number of relations faster. Four distinct use cases were defined to elucidate the potential benefits: *concepts and hierarchy* for replacing multiple API calls to different systems with a single query, *concept merge and relations* for simpler merging of duplicate concepts, *concept search and content hits* for lightweight calculation of the amount of relations going out from a concept and *combining the services* for merging the functionality of the three Content Metadata Services into a single application. In addition, ideas about enriching the data inside the graph with new data sources, such as geographical data and personal data about content authors for centralized access and for connecting it to the data already in the graph, were laid out.

For the time being, the Content Metadata Services have been migrated to use the Neptune database as their master data store, but the process of enhancing the overall application architecture and enabling the novel use cases is still a work in progress. Consequently, not all of the requirements outlined above have been attained — yet. For the use case, the adoption of a graph database is not a goal in itself, and will not solve any problems as such, but it is rather an enabler for and a step towards an architecture more fitting to the current and future needs for managing and serving content metadata at the organization.

To summarize, the *concept merge and relations* and *concept search and content hits* were successfully enabled by the migration, while the *concepts and hierarchy* and *combining the services* scenarios are yet to be achieved. However, implementing them for the Content Metadata Services is currently a matter of designing and developing the next API versions, foremostly of Concepts API, to better utilize the potential of SPARQL queries to aggregate all the necessary data for the use cases of clients. This requires discussions with the different stakeholders to better understand their needs and use cases and to design the next API versions accordingly.

On the non-functional requirements foremostly pertaining to performance of the renewed system, the results of the API performance comparison give some indication that the renewed system better in term of response times for queries traversing a large amount of relations, as was expected. However, the data on this is not conclusive, as the sample is rather small and the performance differences between API versions are not significant. However, based on performance testing, it can be stated with more confidence that the renewed system performs worse under load than original implementation. While this performance was enhanced by implementing a number of optimizations, the renewed system is still slower under heavy traffic and has about half of the throughput of the original version. While the relative numbers differ drastically between the implementations, it is unlikely that the performance limits are to be encountered in the foreseeable future. Performance under load will not, based on the current understanding, prevent further development of the system to support new use cases as planned.

6.2 RQ3: Perspectives to using GDB in microservices

After the preceding discussion, Research Question RQ3 remains to be addressed. It was formulated as follows: *What benefits, opportunities and risks does adopting a graph database provide in a microservice architecture?*

As noted in Section 2.3, a graph database is a valid choice in the context of microservice architecture, provided that basic precautions pertaining to data hygiene are taken. In microservice architecture, data should be owned by a single service. In the context of graph databases, this can mean a named graph inside a shared database. While the architecture needs to be set up in a way that only the owning service can modify data in the graph, non-mutating queries can transcend these limitations. In Neptune, querying data across named graphs is as simple as omitting the *GRAPH* parameter from the SPARQL statement. This way, there is a separation between the named graphs for modifications but all the data in a cluster can be traversed freely when doing queries. Achieving this naturally presupposes that the access to the cluster is protected and mutative operations are restricted. This approach is a variation of the *database cluster pattern*, wherein distinct microservices use a shared database cluster but only touch application-specific tables. Architecturally, the data hygiene can be enhanced by accessing the graph database through a gateway service that manages access to it. It can be argued that this approach — restricting modifications to a graph to “owning” applications but allowing queries from any client — gives the necessary safety guarantees as required by the microservice approach while maintaining the benefits of having the data in a shared graph database that can be queried across the named graphs inside it.

The migration project made it clear, as was discussed in the retrospective, that migrating production systems to a new persistence solution is not a trivial task. In general, switching core parts of a production system to new technologies is costly and such a project should only be initiated after thorough deliberation. On the other hand, probably the only feasible way to build organizational competence on novel technologies, such as on graph databases, is to experiment with them and to build services on top of them. One of the goals of the project was, as noted in the retrospective session, to build competence and gain experience on using graph databases. This expertise can then be leveraged when planning new kinds of services, playing on the strengths of the new technologies available.

From a managerial viewpoint, utilization of niche technologies such as graph databases is a moderate risk staffing-wise. As of now, it is likely to be difficult to find developers with prior experience in these technologies, as their use in the industry is still relatively rare. While there definitely is a learning curve involved with adopting a graph database in a project, this should not be overstated. The experiences from the case project suggest that experienced developers can pick up the core concepts quickly and be productive in a couple of weeks. However, having at least one person with experience on graph databases would have made all the difference in planning the project and doing effort estimation. As this was not the case, help was received instead from having a handful of benchmarking discussions with other organizations with relevant experience. Now that the original team has gained competence, it is likely to be easier for future new team members to get the hang of the system, as they have access to personal guidance and have someone present their questions to.

A difficulty in adopting graph database technologies, from a developer viewpoint, is to not approach them with the same mindset as relational databases. Schema validation, for instance, is a feature that graph database management systems do not generally support. Instead, the types of relations and entities present in a data graph determine its structure. Graph database systems also differ from relational databases in terms of tooling. For instance, an operation as commonplace as connecting to a database with a client application in order to browse the stored data might not be possible, or at least not as straightforward, with graph database systems. In the case of the Neptune database, the solution closest to this experience is using Neptune Workbench, essentially a modified Jupyter Notebook hosted in the Amazon Sagemaker service, for interactive queries and basic data visualization. The application libraries available for manipulating RDF data in Clojure were also lacking, but since Clojure has good interoperation capabilities with Java, the existing Java libraries could be leveraged.

Perhaps the most promising aspects of using a graph database in a microservice context are their flexibility and extensibility. For use cases based on tracking and traversing relations, sometimes complex and even messy, between identifiers, a graph database is a natural fit. In a graph database, knowing the cardinality of relations beforehand, for example, is not necessary, as new relations can be added cheaply. For the use case of the Content Index service, for example, this approach was a great fit. Arguably, storing the data of multiple microservices in a shared graph database can also help re-structuring the application architecture, if needed. In order to move functionality from one microservice to another using the same cluster, no database migrations would be needed. A new kind of microservice architecture could consist of a constellation of services responsible for maintaining an area of a large, interconnected knowledge graph. An approach like this could result in an architecture where focus is shifted from communication and message passing between components to data management, organization and complex queries across graphs.

6.3 Contributions of the research

In the Design Science approach, the result of a study should, by definition, be an artifact. This artifact is generated by following a process of distinct steps from statement of a problem to communication of the results, and it should solve a relevant problem in the organization. The main artifact produced via this thesis has been the new architecture of the services in the case study, built upon the Amazon Neptune graph database. In this architecture, multiple microservices share a graph database cluster, where each service owns and maintains a named graph but is allowed to perform arbitrary queries combining data from different graphs. In the system, the graph database cluster is decoupled from the client services by a mediating proxy microservice that defines the contract by which the storage layer can be accessed.

The proposed architecture, implemented for the case project, can be considered a valid approach for similar use cases. Like graph databases in general, the architecture is well suited for use cases where a large number of relations between entities need to be tracked and traversed. Extending the graph with an Elasticsearch index for full-text queries allows for querying text fields in addition to URI-based queries. Similar use cases are likely to be found in the broadcast industry as well as other industries where product data management or master data management are in focus. The architecture is also open for extension, as new microservices writing to their named graphs can easily be added to the mix. Managing a knowledge graph via specialized microservices helps decouple the different contexts in the data, here termed named graphs, making the complex data structure arguably more manageable than a single web of data would be.

In addition to the architecture, the research has presented a synthesis of literature on microservices and graph databases, two approaches that have thus far rarely combined. An approach was presented for implementing a software architecture utilizing the two paradigms in tandem, without compromising on using best practices related to microservice architecture. Furthermore, a process for migrating microservices into graph databases has been outlined and aspects for consideration in a similar process have been identified. While not all of the goals inferred from the problem statement have yet been achieved, a roadmap towards reaching them has been outlined.

7 Conclusion

This thesis discussed methods for fitting together microservices and graph databases. Following the Design Science Research Process approach, the thesis was structured into theoretical background, an empirical part and a retrospective section. The theoretical section synthesized literature on microservice architecture and graph databases, the empirical part described a case study of migrating three microservices into using the Amazon Neptune graph database, and the final part evaluated and discussed the outcome based on data from performance measurements and a project retrospective session.

Based on the theoretical discussion, a synthesis of literature on microservice architecture and graph databases was proposed. It was proposed that it is feasible to use a shared graph database from multiple microservices, as long as the data ownership principle is ensured by structuring the data into different named graphs "owned" by a single service.

In the case study, an architecture was implemented where three distinct microservices share a graph database cluster, each service modifying only a single named graph. An additional microservice was implemented to act as a gateway to the database, limiting where mutative operations can be performed. An Elasticsearch cluster was added to the system, to act as an additional search index for full-text queries.

Performance measurements gave slight indication that the new implementation would be more performant for complex queries where a large number of relations are traversed. However, the new system was found to less performant under extreme load.

References

[1] B. Moseley and P. Marks, “Out of the tar pit”, *Software Practice Advancement (SPA)*, 2006. [Online]. Available: `http://www.shaffner.us/cs/papers/tarpit.pdf`.

[2] L. Floridi, “AI and its new winter: From myths to realities”, *Philosophy & Technology*, vol. 33, Feb. 2020. DOI: `10.1007/s13347-020-00396-6`.

[3] R. Hickey, “A history of Clojure”, *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, Jun. 2020. DOI: `10.1145/3386321`. [Online]. Available: `https://doi.org/10.1145/3386321`.

[4] M. Ruohonen. (2013). “Graafiteoria”, [Online]. Available: `http://math.tut.fi/~ruohonen/GT.pdf` (visited on 10/06/2020).

[5] R. Angles and C. Gutierrez, “Survey of graph database models”, eng, *ACM computing surveys*, vol. 40, no. 1, pp. 1–39, 2008, ISSN: 0360-0300.

[6] R. Angles, “A comparison of current graph database models”, Apr. 2012, pp. 171–177, ISBN: 978-1-4673-1640-8. DOI: `10.1109/ICDEW.2012.31`.

[7] M. Fowler and J. Lewis. (2014). “Microservices: A definition of this new architectural term”, [Online]. Available: `https://martinfowler.com/articles/microservices.html`.

[8] V. F. Pacheco, *Microservice patterns and best practices: explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices*, 1st ed. Birmingham: PACKT Publishing, 2018.

[9] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does migrating a monolithic system to microservices decrease the technical debt?", *J. Syst. Softw.*, vol. 169, p. 110 710, 2020. DOI: `10.1016/j.jss.2020.110710`. [Online]. Available: `https://doi.org/10.1016/j.jss.2020.110710`.

[10] Y. A. Megid, N. El-Tazi, and A. Fahmy, "Using functional dependencies in conversion of relational databases to graph databases", in *Hartmann S., Ma H., Hameurlain A., Pernul G., Wagner R. (eds) Database and Expert Systems Applications. DEXA 2018. Lecture Notes in Computer Science, vol 11030. Springer, Cham*, 2018, pp. 350–357.

[11] (). "About Yle", [Online]. Available: `https://yle.fi/aihe/about-yle` (visited on 01/23/2021).

[12] J. Nurmi, *Enterprise architecture in public sector ecosystems: A systems perspective.* University of Jyväskylä, 2021, ISBN: 978-951-39-8518-9.

[13] A. R. Hevner, S. R. March, J. Park, and S. Ram, "Design science in information systems research", *Management Information Systems Quarterly*, vol. 28, pp. 75–, Mar. 2004.

[14] K. Peffers, T. Tuunanen, C. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge, "The design science research process: A model for producing and presenting information systems research", in *DESRIST International Conference on Design Science Research in Information Systems and Technology, Claremont, CA, USA, February 24-25, 2006*, 2006, pp. 83–106.

[15] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices architecture enables DevOps: An experience report on migration to a cloud-native architecture”, *IEEE Software*, vol. 33, pp. 1–1, May 2016. DOI: 10.1109/MS.2016.64.

[16] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead”, eng, *IEEE software*, vol. 35, no. 3, pp. 24–35, 2018, ISSN: 0740-7459.

[17] D. Taibi, V. Lenarduzzi, and C. Pahl, “Architectural patterns for microservices: A systematic mapping study”, in *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science, International Conference on Cloud Computing and Services Science - Funchal, Madeira, Portugal, 19 Mar 2018 – 21 Mar 2018*, Mar. 2018, pp. 221–232. DOI: 10.5220/0006798302210232.

[18] C. Richardson, *Microservices Patterns: With examples in Java.* Manning Publications, 2018, ISBN: 9781617294549.

[19] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*, 1st. O’Reilly Media, Inc., 2016, ISBN: 1491956259.

[20] J. Bogner, A. Zimmermann, and S. Wagner, “Analyzing the relevance of SOA patterns for microservice-based systems”, in *10th ZEUS Workshop, ZEUS 2018, Dresden, Germany, 8-9 February 2018*, Mar. 2018.

[21] M. D. McIlroy, E. N. Pinson, and B. A. Tague, “Unix time-sharing system: Foreword”, *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1899–1904, 1978.

[22] (2010). “SOA manifesto”, [Online]. Available: http://www.soa-manifesto.org/ (visited on 09/23/2020).

[23] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1: Reality check and service design", *IEEE software*, vol. 34, no. 1, pp. 91–98, 2017, ISSN: 0740-7459.

[24] A. Cockcroft. (2016). "The evolution of microservices", [Online]. Available: `https://learning.acm.org/techtalks/microservices`.

[25] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of DevOps concepts and challenges", *ACM computing surveys*, vol. 52, no. 6, pp. 1–35, 2020, ISSN: 0360-0300.

[26] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops", *IEEE software*, vol. 33, no. 3, pp. 94–100, 2016, ISSN: 1937-4194.

[27] S. Yegulalp, "What is Docker? Docker containers explained", *InfoWorld.com*, Apr. 2019. [Online]. Available: `https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html`.

[28] J. Muli, *Beginning DevOps with Docker : automate the deployment of your environment with the power of the Docker toolchain*, 1st edition. Birmingham: Packt, 2018, ISBN: 1-78953-957-9.

[29] J. Gray, "A conversation with Werner Vogels", *ACM Queue*, vol. 4, no. 4, 2006. [Online]. Available: `https://queue.acm.org/detail.cfm?id=1142065`.

[30] M. Garriga, "Towards a taxonomy of microservices architectures", in *Software Engineering and Formal Methods*, A. Cerone and M. Roveri, Eds., Springer International Publishing, 2018, pp. 203–218, ISBN: 978-3-319-74781-1.

[31] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells", *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[32] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study", *The Journal of systems and software*, vol. 150, no. 4, pp. 77–97, 2019, ISSN: 0164-1212.

[33] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New Opportunities for Connected Data*, 2nd ed. O'Reilly, 2015, ISBN: 978-1-4919-3089-2.

[34] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition", in *2007 IEEE 23rd International Conference on Data Engineering*, 2007, pp. 976–985.

[35] H. Knublauch and D. Kontokostas, "Shapes Constraint Language (SHACL)", W3C Recommendation, Jul. 2017. [Online]. Available: `https://www.w3.org/TR/shacl/`.

[36] J. Pokorný, "Functional querying in graph databases", *Vietnam journal of computer science*, vol. 5, no. 2, pp. 95–105, 2018, ISSN: 2196-8888.

[37] M. Needham and A. Hodler, *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O'Reilly Media, Incorporated, 2019, ISBN: 9781492047681.

[38] J. Pokorný, "Integration of relational and graph databases functionally", *Foundations of computing and decision sciences*, vol. 44, no. 4, pp. 427–441, 2019, ISSN: 2300-3405.

[39] Y. Unal and H. Oguztuzun, "Migration of data from relational database to graph database", in *ICIST '18: 8th International Conference on Information Systems and Technologies, March 16–18, 2018, Istanbul, Turkey*, 2018.

[40] A. Harth and S. Decker, "Optimized index structures for querying rdf from the web", in *Third Latin American Web Congress (LA-WEB'2005)*, IEEE, 2005, 10 pp.–80, ISBN: 0769524710.

[41] (). "Neptune graph data model", [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/feature-overview-data-model.html (visited on 01/12/2021).

[42] L. Ehrlinger and W. Wöß, "Towards a definition of knowledge graphs", in *Joint Proceedings of the Posters and Demos Track of 12th International Conference on Semantic Systems - SEMANTiCS2016 and 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS16), Leipzig, Germany*, 2016.

[43] T. R. Gruber, "A translation approach to portable ontology specifications", *Knowledge Acquisition*, vol. 5, pp. 199–220, 1993. [Online]. Available: http://tomgruber.org/writing/ontolingua-kaj-1993.pdf.

[44] C. Feilmayr and W. Wöß, "An analysis of ontologies and their success factors for application to business", *Data & Knowledge Engineering*, vol. 101, pp. 1–23, 2016, ISSN: 0169-023X. DOI: https://doi.org/10.1016/j.datak.2015.11.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0169023X1500110X.

[45] O. Lassila and R. R. Swick, "Resource Description Framework (RDF) Model and Syntax Specification", W3C, W3C Recommendation, Feb. 1999. [Online]. Available: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[46] G. Klyne, J. J. Carroll, and B. McBride, "RDF 1.1 Primer", W3C, W3C Working Group Note, Jun. 2014. [Online]. Available: https://www.w3.org/TR/rdf11-primer/.

[47] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - the story so far", *Int. J. Semantic Web Inf. Syst.*, vol. 5, pp. 1–22, 2009.

[48] T. Berners-Lee, J. Handler, and O. Lassila, "The semantic web", *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.

[49] N. Shadbolt, W. Hall, and T. Berners-Lee, “The semantic web revisited”, *IEEE intelligent systems*, vol. 21, no. 3, pp. 96–101, 2006, ISSN: 1541-1672.

[50] E. Prud’hommeaux, “Optimal RDF access to relational databases”, W3C, Tech. Rep., Nov. 2004. [Online]. Available: `https://www.w3.org/2004/04/30-RDF-RDB-access/`.

[51] P. Cuddihy, J. McHugh, J. Williams, and V. Mulwad, “SemTK: An ontology-first, open source semantic toolkit for managing and querying knowledge graphs”, Oct. 2017.

[52] (). “Cosplay”, [Online]. Available: `https://finto.fi/yso/fi/page/p20742`.

[53] D. Beckett, T. Berners-Lee, E. Prud’hommeaux, and G. Carothers, “RDF 1.1 Turtle”, W3C Recommendation, Feb. 2014. [Online]. Available: `https://www.w3.org/TR/turtle/`.

[54] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, P.-A. Champin, and N. Lindström, “JSON-LD 1.1”, W3C Recommendation, Jul. 2020. [Online]. Available: `https://www.w3.org/TR/json-ld/`.

[55] E. Prud’hommeaux and A. Seaborne. (2008). “SPARQL query language for RDF”, [Online]. Available: `https://www.w3.org/TR/rdf-sparql-query/` (visited on 10/07/2020).

[56] J. Pérez, M. Arenas, and C. Gutierrez, “Semantics and complexity of SPARQL”, *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 3, pp. 1–45, 2009, ISSN: 0362-5915.

[57] (). “RDF datasets”, [Online]. Available: `https://www.w3.org/TR/rdf11-concepts/#section-dataset` (visited on 01/18/2021).

[58] M. Hjort, “Polyglot microservices: Comparison between Javascript, Scala and Clojure”, ClojuTRE, 2015, [Online]. Available: `https://clojutre.org/2015/#polyglot-microservices`.

[59] J. Karemo, P. Valkonen, and T. Haapaniemi. (Jan. 2017). “A story of a microservice: Lessons from the trenches”. Reaktor blog, [Online]. Available: https://www.reaktor.com/blog/a-story-of-a-microservice/.

[60] (Jun. 2020). “Yle areena on kasvanut ylen kiinnostavimmaksi palveluksi”, [Online]. Available: https://yle.fi/aihe/artikkeli/2020/06/11/suomen-suosituin-suoratoistopalvelu-on-edelleen-yle-areena.

[61] (). “Yle Areena”, [Online]. Available: https://areena.yle.fi/ (visited on 11/29/2020).

[62] M. Hjort, “Modern Finnish enterprise level microservices at Yle”, Reaktor Breakpoint, 2015, [Online]. Available: https://vimeo.com/channels/973982/144467127.

[63] (2020). “Yle API tutorials”, [Online]. Available: https://developer.yle.fi/en/tutorials/index.html (visited on 11/29/2020).

[64] (May 2015). “Käyttöehdot Ylen API-rajapintaan liittymiselle”, [Online]. Available: https://developer.yle.fi/static/terms-of-service.pdf.

[65] H. Laukkanen, “Clojure powered services at Finnish Broadcasting Company”, Amsterdam Clojure Days, 2019, [Online]. Available: https://youtu.be/WRI71TJ_hX0.

[66] K. Ylä-Anttila. (Jun. 2019). “Yleisradion pilvimatka - oppeja ja kokemuksia”. Solita Public Sector Pulse, [Online]. Available: https://www.slideshare.net/Solita_Oy/yleisradion-pilvimatka-oppeja-ja-kokemuksia-kalle-ylanttila-yle.

[67] (2020). “Pikku Kakkonen”, [Online]. Available: https://yle.fi/pikkukakkonen/ (visited on 12/07/2020).

[68] (2020). “Programs API”, [Online]. Available: https://developer.yle.fi/en/api/index.html#/programs-api-search-programs-clips-and-episodes/ (visited on 12/01/2020).

[69] (). “Wikidata”, [Online]. Available: https://www.wikidata.org/ (visited on 01/05/2021).

[70] (). “Finto”, [Online]. Available: http://finto.fi/en/ (visited on 01/05/2021).

[71] J. Riley, *Understanding Metadata: What is Metadata, and What is it For?: A Primer*. NISO, 2017, ISBN: 978-1-937522-72-8.

[72] O. Suominen and P. Virtanen. (2020). “Yle meets Annif - an open source tool for automated subject indexing”. MDN Workshop 2020, [Online]. Available: https://tech.ebu.ch/contents/publications/events/presentations/mdn2020/yle-meets-annif--an-open-source-tool-for-automated-subject-indexing (visited on 12/03/2020).

[73] (2020). “Ray Winstonen Sisilia. Jakso 1: Palermo”, [Online]. Available: https://areena.yle.fi/1-50482178 (visited on 12/05/2020).

[74] K. Snabb and T. Kalvas, “GraphDB: Ehdotus ja roadmap”, Internal presentation, 2020.

[75] ——, “GraphDB demo: GraphDB as an option for improving meta-api, relations-api and content-index”, Internal presentation, 2019.

[76] J. Ruotsalainen, interview, Oct. 26, 2020.

[77] N. Yuhann, “The Forrester Wave™: Graph data platforms, Q4 2020 — the 12 providers that matter most and how they stack up”, Forrester, Tech. Rep., Nov. 2020. [Online]. Available: https://reprints2.forrester.com/#/assets/2/374/RES161455/report (visited on 01/31/2020).

[78] (). “Neo4j”, [Online]. Available: https://github.com/neo4j/neo4j (visited on 02/21/2021).

[79] (). “Amazon Neptune”, [Online]. Available: https://aws.amazon.com/neptune/ (visited on 01/09/2021).

[80] (). “BlazeGraph”, [Online]. Available: https://github.com/blazegraph/database (visited on 01/09/2021).

[81] B. DuCharme. (). “SPARQL and Amazon Web Service’s Neptune database”. blog article, [Online]. Available: http://www.snee.com/bobdc.blog/2017/12/sparql-and-amazon-web-services.html (visited on 01/09/2021).

[82] B. Bebee, D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan, S. Rondelli, A. Ryazanov, M. Schmidt, and K. S, “Amazon Neptune: Graph data management in the cloud”, 2018, [Online]. Available: http://ceur-ws.org/Vol-2180/paper-79.pdf (visited on 01/31/2020).

[83] T. Riggan, S. Marshall, and S. Singamaneni, “Using Amazon Neptune to power identity resolution at scale”, 2019, [Online]. Available: https://www.slideshare.net/AmazonWebServices/using-amazon-neptune-to-power-identity-resolution-at-scale-adb303-atlanta-aws-summit (visited on 01/31/2020).

[84] I. Robinson, “Amazon Neptune”, AWS Dev Day, Jul. 2018, [Online]. Available: http://aws-de-media.s3.amazonaws.com/images/DevDays%202018/DM4-AWS18_DevDay_neptune.pdf (visited on 01/31/2020).

[85] (). “SPARQL default graph and named graphs”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/feature-sparql-compliance.html#sparql-default-graph (visited on 02/06/2021).

[86] (). “RDF 1.1 N-Triples”, [Online]. Available: https://www.w3.org/TR/n-triples/ (visited on 01/13/2021).

[87] (). “Using the Amazon Neptune bulk loader to ingest data”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/bulk-load.html (visited on 01/03/2021).

[88] (). “Regex”, [Online]. Available: https://www.w3.org/TR/rdf-sparql-query/#funcex-regex (visited on 01/11/2021).

[89] (). “Amazon Neptune full-text search using Amazon Elasticsearch service”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/full-text-search.html (visited on 01/11/2021).

[90] N. Weiderman, D. Smith, and S. Tilley, “Approaches to legacy system evolution”, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-97-TR-014, 1997. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12919.

[91] (). “Rdflib 5.0.0 documentation”, [Online]. Available: https://rdflib.readthedocs.io/en/stable/ (visited on 01/13/2021).

[92] (). “AMQP 0-9-1 model explained”, [Online]. Available: https://www.rabbitmq.com/tutorials/amqp-concepts.html (visited on 01/16/2021).

[93] (). “Edn - extensible data notation”, [Online]. Available: https://github.com/edn-format/edn (visited on 01/13/2021).

[94] (). “Using Neptune Streams”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/streams-using.html (visited on 01/14/2021).

[95] (). “Neptune Lab Mode”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/features-lab-mode.html (visited on 01/14/2021).

[96] (). “Capturing graph changes in real time using Neptune Streams”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/streams.html (visited on 01/14/2021).

[97] (). “Using AWS CloudFormation to set up Neptune-to-Neptune replication with the Streams consumer application”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/streams-consumer-setup.html (visited on 01/14/2021).

[98] (). “Export Neptune to ElasticSearch”, [Online]. Available: https://github.com/awslabs/amazon-neptune-tools/tree/master/export-neptune-to-elasticsearch (visited on 01/14/2021).

[99] (). “Sample SPARQL queries using full-text search in Neptune”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/full-text-search-sparql-examples.html (visited on 01/14/2021).

[100] J. Aasman, “Transmuting information to knowledge with an enterprise knowledge graph”, *IT Professional*, vol. 19, no. 6, pp. 44–51, 2017.

[101] (). “Gatling”, [Online]. Available: https://gatling.io/ (visited on 12/17/2020).

[102] (). “Gatling (software)”, [Online]. Available: https://en.wikipedia.org/wiki/Gatling_(software) (visited on 01/23/2021).

[103] (). “Gatling: Concepts”, [Online]. Available: https://gatling.io/docs/current/general/concepts/ (visited on 01/23/2021).

[104] (). “Akka”, [Online]. Available: https://akka.io/ (visited on 01/23/2021).

[105] (). “Netty”, [Online]. Available: https://netty.io/ (visited on 01/23/2021).

[106] (). “Gatling”, [Online]. Available: https://github.com/gatling/gatling (visited on 01/23/2021).

[107] (). “Feeders”, [Online]. Available: https://gatling.io/docs/current/session/feeder/#feeder (visited on 01/24/2021).

[108] (). “Cache-Control”, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control (visited on 01/24/2021).

[109] (). “Adding Neptune replicas to a DB cluster”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/manage-console-add-replicas.html (visited on 01/27/2021).

[110] (). “Connecting to Amazon Neptune endpoints”, [Online]. Available: https://docs.aws.amazon.com/neptune/latest/userguide/feature-overview-endpoints.html (visited on 01/27/2021).

[111] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Third. Reading, Mass.: Addison-Wesley, 1997, ISBN: 9780201896831.

[112] T. Dingsøyr, “Postmortem reviews: Purpose and approaches in software engineering”, *Information and Software Technology*, vol. 47, pp. 293–303, Mar. 2005. DOI: 10.1016/j.infsof.2004.08.008.

[113] T. O. A. Lehtinen, J. Itkonen, and C. Lessenius, “Recurring opinions or productive improvements—what agile teams actually discuss in retrospectives”, *Empirical Software Engineering*, vol. 22, pp. 2409–2452, 2017.

[114] M. Myllyaho, O. Salo, J. Kääriäinen, J. Hyysalo, and J. Koskela, “A review of small and large post-mortem analysis methods”, VTT, Dec. 2004, [Online]. Available: http://virtual.vtt.fi/virtual/proj1/projects/merlin/pub/analysis%20_of_small_and_large_post-mortem_review_methods.pdf (visited on 01/31/2020).

[115] B. Bebee, R. Chander, A. Gupta, A. Khandelwal, S. Mallidi, M. Schmidt, R. Sharda, B. Thompson, and P. Upadhyay, “Enabling an enterprise data

management ecosystem using change data capture with Amazon Neptune", 2017, [Online]. Available: `http://ceur-ws.org/Vol-2456/paper49.pdf`.

[116] J. Stillerman, T. Fredian, M. Greenwald, and G. Manduchi, "Data catalog project — a browsable, searchable, metadata system", *Fusion Engineering and Design*, vol. 112, pp. 995–998, 2016.

Appendix A Code examples

Code snippet 5 The concept “cosplay” in the General Finnish Ontology in Turtle format

```
{Turtle}

@prefix yso: <http://www.yso.fi/onto/yso/> .

@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

@prefix allars: <http://www.yso.fi/onto/allars/> .

@prefix koko: <http://www.yso.fi/onto/koko/> .

@prefix ysa: <http://www.yso.fi/onto/ysa/> .

@prefix dc: <http://purl.org/dc/terms/> .

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .


yso:p20742

  skos:related yso:p19001, yso:p2901, yso:p1273, yso:p4733 ;

  skos:prefLabel "cosplay"@en, "cosplay"@sv, "cosplay"@fi ;

  a <http://www.yso.fi/onto/yso-meta/Concept>, skos:Concept ;

  skos:exactMatch allars:Y38283, koko:p50474, ysa:Y165778 ;

  skos:inScheme yso: ;

  dc:modified "2020-01-25"^^xsd:date ;

  dc:created "2009-05-15"^^xsd:date ;

  skos:broader yso:p562 ;

  skos:closeMatch <http://id.loc.gov/authorities/subjects/sh2015001998> ;

  skos:altLabel "pukuilu"@fi .
```

Code snippet 6 A query example.

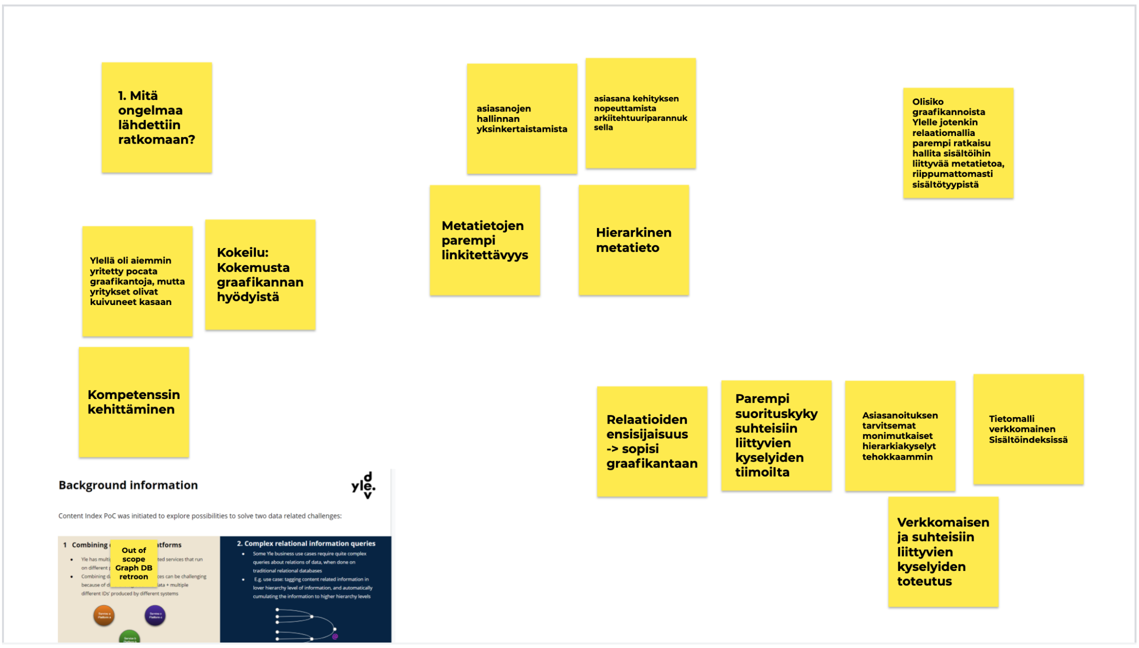
```
{json}

GET https://content-index.api.yle.fi/v2/ancestors/1-50547210

{

   "data":{

      "plasma-seasons":[

         "16-0-1113061592527"

      ],

      "projects":[

         "40-2-3050582"

      ],

      "products":[

         "40-1-4056529"

      ],

      "plasma-series":[

         "16-10-1189564427527"

      ],

      "cost-centers":[

         "60-12107"

      ],

      "areena-seasons":[

         "1-50547206"

      ],

      "areena-series":[

         "1-50654666"

      ],

   }

}
```

Code snippet 7 Example query from Relations API.

```
{json}

GET https://relations.api.yle.fi/v2/relation?targetId=1-2155797

{

   "data":[

      {

         "id":"52-47016285",

         "sourceId":"18-177017",

         "targetId":"1-2155797",

         "originId":"51-3",

         "relationType":"isCountryOfOriginOf"

      },

      {

         "id":"52-47016286",

         "sourceId":"18-299297",

         "targetId":"1-2155797",

         "originId":"51-3",

         "relationType":"isGenreOf"

      },

      {

         "id":"52-47016287",

         "sourceId":"18-299306",

         "targetId":"1-2155797",

         "originId":"51-3",

         "relationType":"isGenreOf"

      },

      {

         "id":"52-47016288",

         "sourceId":"18-299286",

         "targetId":"1-2155797",

         "originId":"51-3",
```

Appendix B Retrospective answers

Figure B.1: What was the original problem?



1. Mitä ongelmaa lähdettiin ratkomaan?

asiasanojen hallinnan yksinkertaistamista

asiasana kehityksen nopeuttamista arkiitehtuuriparannuksella

Olisiko graafikannoista Ylelle jotenkin relaatiomallia parempi ratkaisu hallita sisältöihin liittyvää metatietoa, riippumattomasti sisältötyypistä

Metatietojen parempi linkitettävyys

Hierarkinen metatieto

Yllä oli aiemmin yritetty pocata graafikantoja, mutta yritykset olivat kuivuneet kasaan

Kokeilu: Kokemusta graafikannan hyödyistä

Kompetenssin kehittäminen

Background information

Content Index PoC was initiated to explore possibilities to solve two data related challenges:

1 Combining … tforms

- Yle has multi… ted services that run on different …
- Combining d… ces can be challenging because of di… ata + multiple different IDs' produced by different systems

Out of scope Graph DB retroon

2. Complex relational information queries

- Some Yle business use cases require quite complex queries about relations of data, when done on traditional relational databases
- E.g. use case: tagging content related information in lover hierarchy level of information, and automatically cumulating the information to higher hierarchy levels

Relaatioiden ensisijaisuus -> sopisi graafikantaan

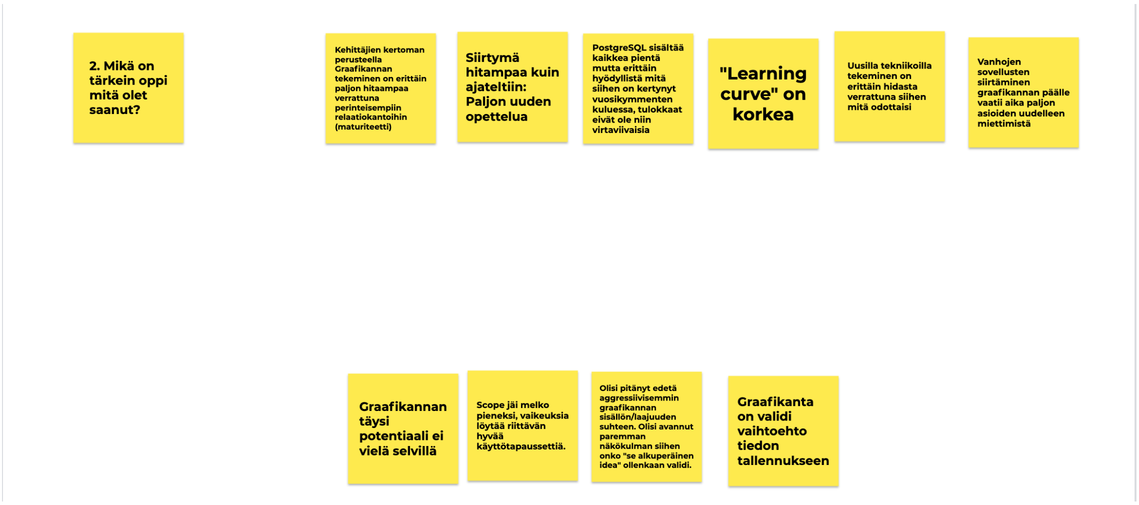
Parempi suorituskyky suhteisiin liittyvien kyselyiden tiimoilta

Asiasanoituksen tarvitsemat monimutkaiset hierarkiakyselyt tehokkaammin

Tietomalli verkkomainen Sisältöindeksissä

Verkkomaisen ja suhteisiin liittyvien kyselyiden toteutus

Figure B.2: What was the most important thing you learned?



2. Mikä on tärkein oppi mitä olet saanut?

Kehittäjien kertoman perusteella Graafikannan tekeminen on erittäin paljon hitaampaa verrattuna perinteisempiin relaatiokantoihin (maturiteetti)

Siirtymä hitampaa kuin ajateltiin: Paljon uuden opettelua

PostgreSQL sisältää kaikkea pientä mutta erittäin hyödyllistä mitä siihen on kertynyt vuosikymmenten kuluessa, tulokkaat eivät ole niin virtaviivaisia

"Learning curve" on korkea

Uusilla tekniikoilla tekeminen on erittäin hidasta verrattuna siihen mitä odottaisi

Vanhojen sovellusten siirtäminen graafikannan päälle vaatii aika paljon asioiden uudelleen miettimistä

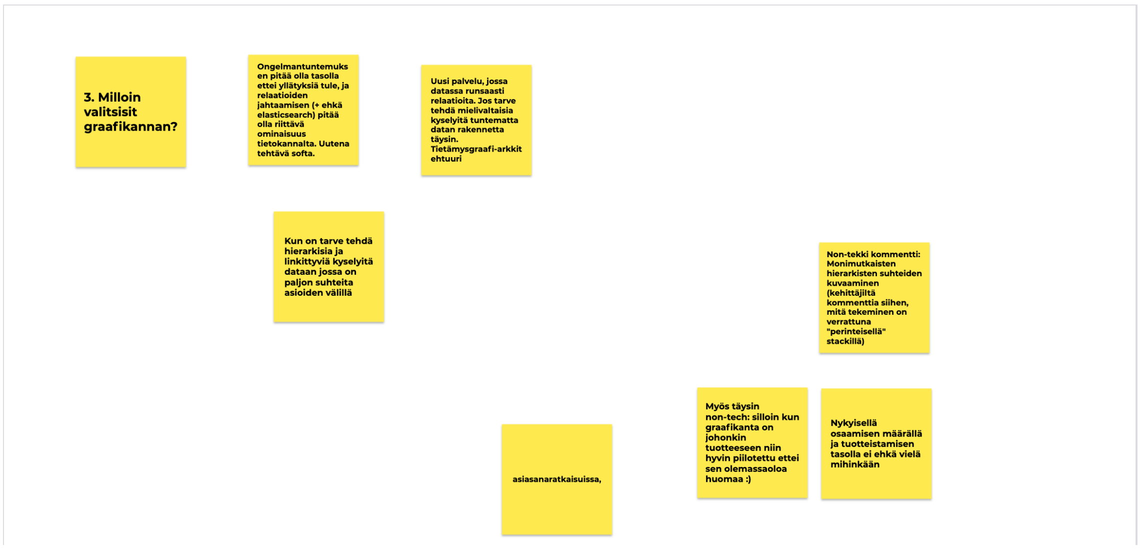
Graafikannan täysi potentiaali ei vielä selvillä

Scope jäi melko pieneksi, vaikeuksia löytää riittävän hyvää käyttötapaussettiä.

Olisi pitänyt edetä aggressiivisemmin graafikannan sisällön/laajuuden suhteen. Olisi avannut paremman näkökulman siihen onko "se alkuperäinen idea" ollenkaan validi.

Graafikanta on validi vaihtoehto tiedon tallennukseen

Figure B.3: When would you choose a graph database now?



3. Milloin valitsisit graafikannan?

Ongelmantuntemuksen pitää olla tasolla ettei yllätyksiä tule, ja relaatioiden jahtaamisen (+ ehkä elasticsearch) pitää olla riittävä ominaisuus tietokannalta. Uutena tehtävä softa.

Uusi palvelu, jossa datassa runsaasti relaatioita. Jos tarve tehdä mielivaltaisia kyselyitä tuntematta datan rakennetta täysin. Tietämysgraafi-arkkitehtuuri

Kun on tarve tehdä hierarkisia ja linkittyviä kyselyitä dataan jossa on paljon suhteita asioiden välillä

Non-tekki kommentti: Monimutkaisten hierarkisten suhteiden kuvaaminen (kehittäjiltä kommenttia siihen, mitä tekeminen on verrattuna "perinteisellä" stackillä)

asiasanaratkaisuissa,

Myös täysin non-tech: silloin kun graafikanta on johonkin tuotteeseen niin hyvin piilotettu ettei sen olemassaoloa huomaa :)

Nykyisellä osaamisen määrällä ja tuotteistamisen tasolla ei ehkä vielä mihinkään