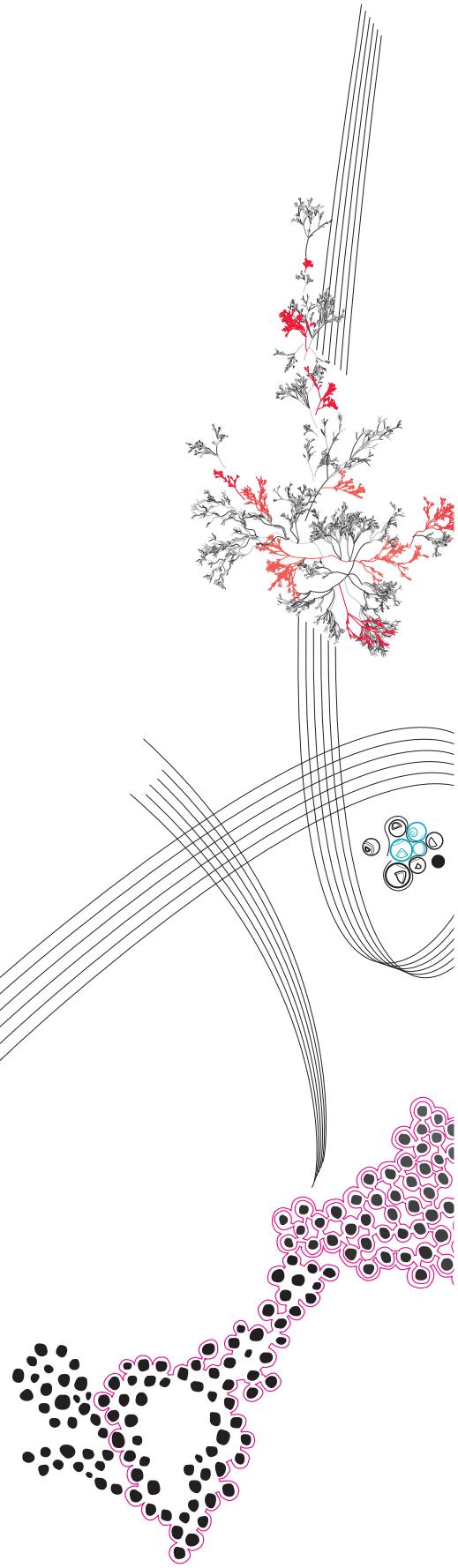


MSc Computer Science
Research Topics

RIRmap: Using a Graph Database to Enhance Exploratory Research for RIR WHOIS data

Jelle Nijland

Supervisor: Ralph Holz & Ebrima Jaw

July, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Regional Internet Registries (RIR) are organisations that manage the allocation and registration of IP addresses and Autonomous System Numbers. They publish WHOIS files which documents IP address assignments and their contact details. Additionally, they maintain delegation files which documents prefix and ASN allocation. Furthermore, they maintain rDNS zone files, these files document prefix to nameserver mappings. These data are valuable to researchers and operators; however, the relationships between prefixes, origins, organisations, and maintainers that are connected indirectly, span a complex graph. This paper attempts to fill this gap by exploring graph databases using data from `rir-data.org`, a project that aims to provide longitudinal RIR data in a consistent format. However, while parsing these data, we observed significant inconsistencies such as missing origins in their data. We utilized raw BGP Routing Information Base (RIB) files to mitigate this inconsistency and enrich the dataset. Then we build graphs out of the enriched data to allow traversal of relationships between organisations, maintainers, prefixes, and origins in an effort to complement exploratory research efforts. This study documents our design choices and query evaluations to solve currently time-consuming and resource-intensive queries in non-graph-based solutions. Additionally, we provide a suitable method of modelling the data to enable comparison as well as a library of queries to perform research. Finally, we highlight some open research challenges to extend our work, such as developing a unified query layer to leverage the strengths of both a graph and a relational database.

Keywords: RIR, WHOIS, Graph database

Chapter 1

Introduction

This chapter presents background information on our research's fundamental concepts, motivations, goals, questions, and contributions. We propose a graph database model for storing RIR WHOIS data and contribute a query library to enable exploratory research. Before we discuss this, we first look at the background.

1.1 Background

The internet has revolutionized communications, transforming how we connect and share information across the globe. Where sending mail across continents once took multiple days, it now takes mere seconds through digital means. A fundamental aspect of this connectivity is the use of unique numerical identifiers known as Internet Protocol (IP) addresses, which are essential for communication between devices on a network.

Every device connected to the internet has a unique number called an IP address. This unique identifier, such as `192.168.1.1` in the case of IPv4, allows a device to communicate with other network-connected devices both within and outside its local network. IP addresses function much like physical mail addresses; just as mail must be sent to the correct address to reach its intended recipient, data must be directed to the correct IP address to ensure it reaches the right device. Without unique IP addresses, the data could be misrouted, much like mail delivered to the wrong house.

The allocation of these unique IP addresses is managed by Regional Internet Registries (RIR). RIRs are responsible for distributing IP address blocks to various operators, ensuring that each address is unique and properly assigned. Operators include entities such as Internet Service Provider (ISP), companies, governmental agencies, and universities. These operators receive one or more Autonomous System (AS) numbers and corresponding IP address prefixes. An AS is a collection of IP networks and routers under the control of a single organization that presents a common routing policy to the internet.

The Border Gateway Protocol (BGP) plays a crucial role in facilitating communication between different ASes. BGP is used to exchange reachability information among ASes, allowing them to announce their IP address prefixes to their neighbours. This announcement process enables effective communication and data routing across the vast network of interconnected devices on the internet.

RIRs maintain comprehensive records of the IP addresses and AS numbers they allocate. These records, known as WHOIS and delegation files, contain essential information about the operators and their contact details. When a network is misconfigured or generates malicious traffic, these records enable network administrators to identify and contact the responsible operators to resolve the issue.

By providing operators with the necessary IP address blocks and AS numbers, RIRs facilitate the smooth functioning of the internet's infrastructure. Understanding the role of RIRs, the allocation of IP addresses, and the importance of protocols like BGP is essential for network researchers and professionals who work to maintain and improve the reliability and security of the internet.

Graph databases have gained significant attention in recent years for their ability to efficiently handle complex relationships. Our research is particularly inspired by the work presented by Schlamp et al. in the `HEAP` paper [1], which demonstrated the potential of graph databases in analysing RIR data for detecting malicious activities in BGP.

1.2 Motivation

RIRs publishing the daily prefix allocations date back to the inception of these registries, creating an ever-increasing amount of data. Each registry has their own inconsistencies when it comes to storing these data. Certain RIRs have different field names for the same properties, which can also contain circular references. For example, RIR_x says a prefix is managed by RIR_y while RIR_y says a prefix is managed by RIR_x. Along with these inconsistencies and lack of standardization, WHOIS services provided by RIRs have limited querying capabilities, making RIR data complex to parse.

Schlamp et al. has made substantial progress in using RIR data to infer and classify malicious activity in BGP networks. Despite its effectiveness, the `HEAP` implementation required manual intervention to maintain performance, highlighting a critical area for improvement.

This paper explores graph databases and GraphQL to store parsed WHOIS data. GraphQL returns the data to the user in a graph form. Storing RIR data in a graph representation allows for a flexible and intuitive way to show relationships between prefixes, AS-es, maintainers, and organizations. Graphs naturally represent networks and allow for advanced querying and analysis, such as community detection.

1.3 Goal, Research Questions & Approach

This subsequent section details our design and research goals, research questions, and the respective adopted approaches.

1.3.1 Research & Design Goal

The goal of this research is to design a system that makes RIR data available in graph form. It should ingest data from a source that processed WHOIS data, daily delegation files and Reverse Domain Name System zone fragment files. This data is enriched by adding origins, stored in a database and exposed via a GraphQL API to support exploratory research. This is achieved by making the data accessible on a day-to-day granularity, enabling comparison in the temporal domain. Additionally, we provide a query library for running queries on the graph database.

This study aims to design a system to support researchers working with longitudinal RIR WHOIS data. These questions will be answered by literature research and reaching out to researchers in the field. The design goal is to determine whether a graph database can support these needs.

1.3.2 Research Questions & Approach

RQ: What is the suitable design for modelling and comparing RIR data across temporal and spatial dimensions?

We will propose a suitable data model to store RIR WHOIS data in a graph database. As the data in the database will keep increasing, We need to design the database accordingly to enable comparison. We will develop a system to ingest data from the source, enrich it and store it in a format to easily convert it. We will evaluate the system by running the queries and benchmarking.

Before we can answer our main research question, we first need to review the following:

What is the most suitable graph database for storing RIR data based on literature?

We will investigate different database systems using existing literature to determine a suitable storage solution. We opted specifically for a graph-type database as it is indicated by literature, and instinctively, it lends itself well for analysing the underlying relationships in RIR WHOIS data. This paper has evaluated several systems by reviewing each system's stability, performance, and support.

What are the suitable use cases to evaluate the proposed system?

We will contact several users of the intended system to gather queries that interest them and determine which are time and resource-intensive, knowing that a graph-based approach does not loan itself well to aggregate statistics.

1.3.3 Contributions

We contribute to the field with a suitable data model for efficiently storing and retrieving RIR data. We implemented an ingester for processed RIR data and a graph creator for this processed data. We implemented a query library informed by queries that interest other researchers. We demonstrated which queries can best be solved by graph databases and which queries are best solved by other systems.

Chapter 2

Core Concepts

This chapter discusses the evolutions and fundamental topics relevant to this research. The subsequent sections present the overview of graph databases, GraphQL, and finally Regional Internet Registries (RIR) data.

2.1 Graph Databases

Building on the groundwork laid by the `HEAP` paper, we delve into the fundamentals of graph databases, which offer a robust framework for managing interconnected data without the overhead of extensive `JOIN` operations typical in RDBMS.

Graph Databases started being developed in the mid-1960s [2]. However, commercial applications of Graph databases did not appear until early 1990. In the mid-2000s, the first commercial ACID-compliant transactional databases became available. ACID stands for Atomicity, Consistency, Isolation, and Durability [3]. This means that the data in the database is not added unless specific safeguards have been met. Graph databases fall in the Not only Structured Query Language (NoSQL) family, which differs from traditional databases, as they do not use tables or rows to organize data. Instead, data is stored in a model to represent the data. This can be graphs, key-value pairs, a document store, or a wide column.[4].

Graph databases are designed to store and query interconnected data represented as graphs [4]. A collection of objects that can be represented as nodes and edges. Nodes are used to represent entities, such as a bank account, a person, or an IP address. For instance, the edges represent relationships like a person has a bank account or a person uses an IP address. Edges can have a direction or be undirected edges. If the edge has a direction, the relationship can be different depending on the direction. Undirected edges have a singular meaning. Information related to these nodes and edges is stored in properties. For example, the relationship between a person and an IP address can have a property describing the date and time or the ISP providing it. A person entity node can have properties such as name, age, or job title.

Additionally, Graph databases gained popularity in the industry with the rise of social media [4]. The growing connectivity of data required a more connected way of storing data to enable more efficient data retrieval and traversal. Conventional relational databases use tables, but they have inherently restrictive links between rows of data [4]. In relational databases, you can link data together using foreign keys. However, this does not allow further information to be stored on this link. For instance, we need to construct additional tables describing the relationship if we want to store more information about the link. As a result, we exploit the potential of graph databases, which allow data about the relationship

between two entities in the edge connecting the nodes to be stored. See Figure 2.1 for a visual representation of this difference.

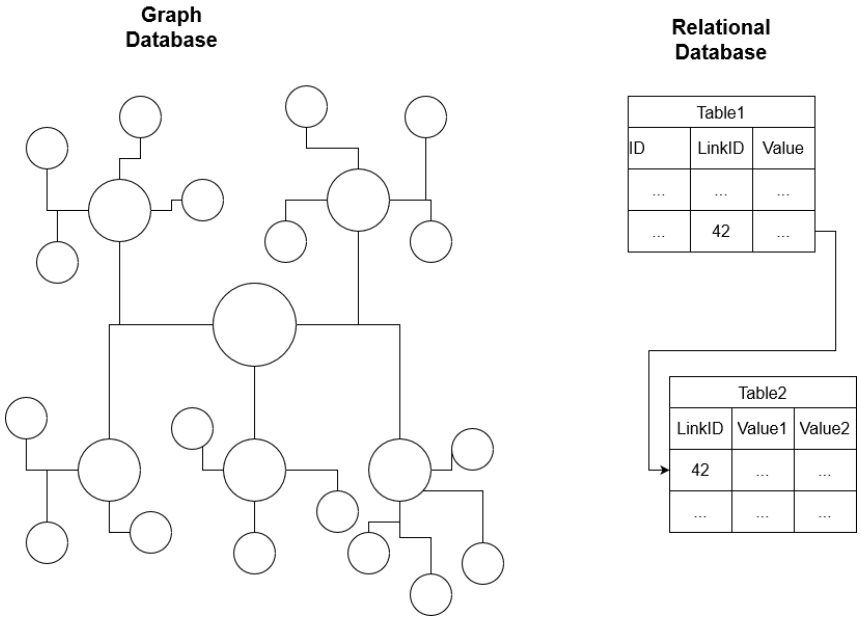


FIGURE 2.1: Model difference between a graph database and a relational database

Besta et al. [4] described many types of NoSQL databases, such as Resource Description Framework (RDF) systems, Native graph, wide-column, document, tuples, and key-value stores. The choice of database depends on the type of data you want to store. We will restrict ourselves to comparing native graph databases, as the data we wish to store is highly interconnected. These can be categorized into three groups based on their data model: Hypergraphs, RDF, and Labeled Property Graph (LPG) database models.

Hypergraphs are listed solely for completeness and are only used by one graph database. They contain edges that connect any number of vertices. This type was not considered, keeping the scope limited.

Resource Description Framework (RDF) model implementations are also called triple stores, as they are a collection of triples. Each triple consists of a subject, a predicate, and an object. The subject and object are different graph nodes, and predicates are the graph's edges. This data model can be implemented in both graph and relational databases.

In addition, the Labeled Property Graph (LPG) model enriches the classic graph model by adding labels to vertices and edges. These labels define the class a vertex or edge belongs to and contain properties. These are key-value pairs that store additional data on the graph.

We consider software implementing graphs, such as Spark. Spark is a general-purpose distributed dataflow framework designed to process and analyse large amounts of data in a distributed environment. Spark handles distributed task dispatching, scheduling, and input/output (I/O). It offers several high-level APIs for real-time processing, machine learning, and graph processing. GraphX and Graphframes are thin graph-processing frameworks on top of Spark. They provide an API for expressing graph computation and are built for large-scale graph data. Additionally, they are optimized for minimizing complex join operations, thus reducing the number of stages needed for a computation [5].

2.2 GraphQL

To improve our querying capabilities, we incorporate GraphQL, a powerful query language that aligns well with the expressive needs highlighted by the HEAP team.

In 2012, Facebook developed GraphQL and subsequently released it as an open-source project in 2015 [6]. Furthermore, it was incorporated into the GraphQL Foundation in 2018, establishing a neutral platform for collecting and distributing membership dues and community support [7]. Noteworthy participants, such as AWS, IBM, Microsoft, PayPal, and Shopify, contribute to its membership. The GraphQL Working Group facilitates development of GraphQL, convening monthly to deliberate on pertinent matters and collectively advance the core GraphQL project.

GraphQL is a declarative data retrieval API query language that enables the client to specify the data it needs [8]. This feature saves bandwidth on data a client does not need. It is presented as an alternative to REST, requiring multiple requests to different endpoints to retrieve data. This functionality enables retrieving all required data in a single request, reducing network load. It returns the requested data as `JSON` object with a graph representation. An example can be seen in Figure 2.2.

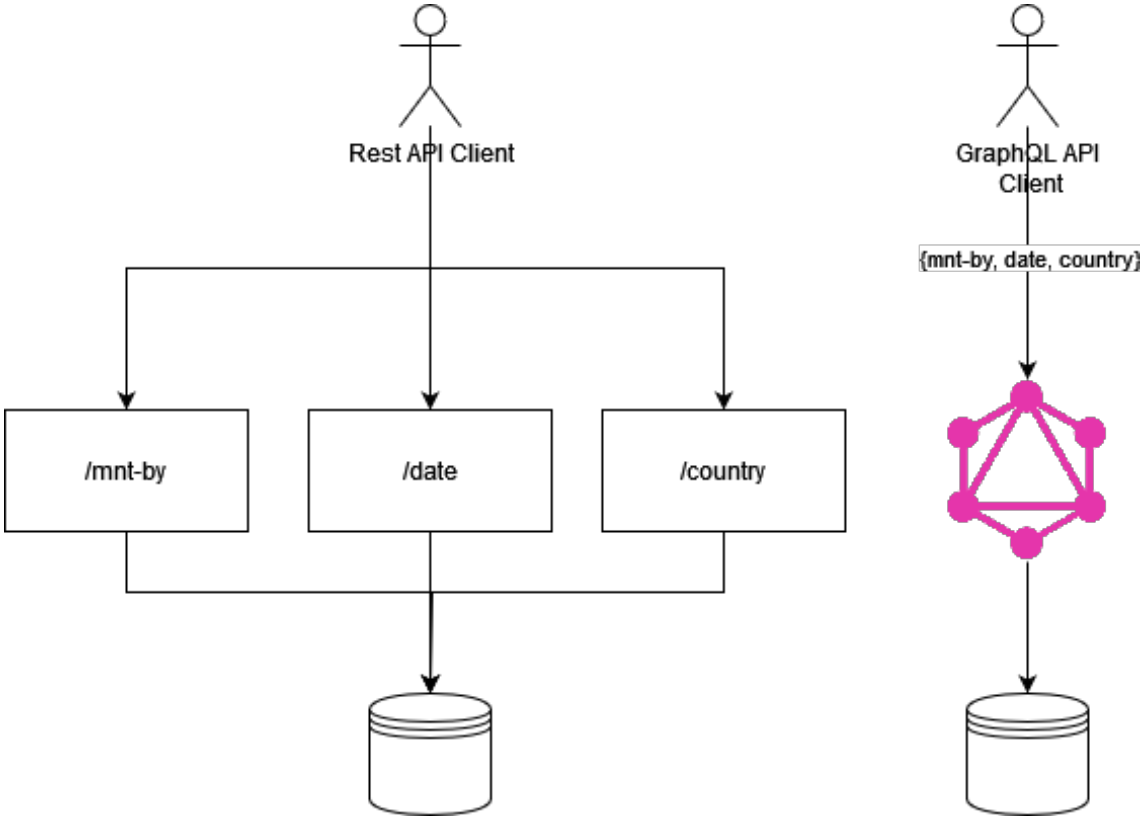


FIGURE 2.2: Model difference between a REST API and GraphQL API

2.3 Regional Internet Registries (RIR)

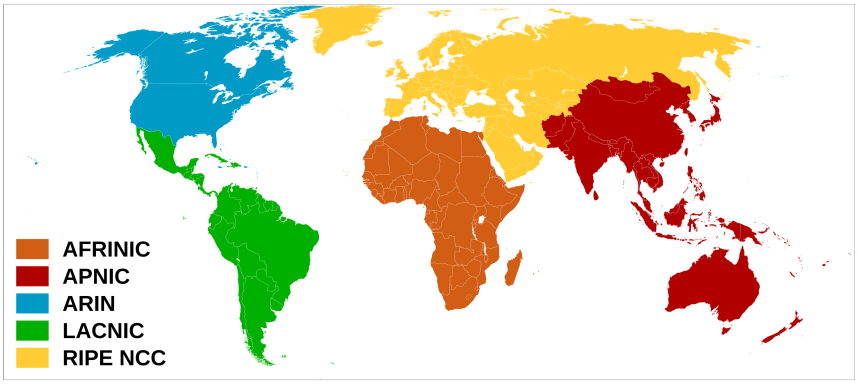
In line with the HEAP study, we focus on RIR data, examining its structure and the potential for deriving insights into network behaviors and anomalies.

RIRs provide a myriad of data sources, for this research we will confine ourselves to

WHOIS snapshots, delegation files and rDNS zone files. We restrict ourselves to the RIR databases, as other IRR databases contain some irregularities, as claimed by Du et al. [9]. We use `rir-data.org` RIR data because it is already parsed and stored in a standard format [10].

Regional Internet Registries (RIR) are organisations that allocate Internet number resources such as IP addresses (IPv4 and IPv6) and Autonomous System Number (ASN) [11]. These resources can be allocated to end users or intermediaries to allocate to their end users. These intermediaries can be Local Internet Registries (LIR)s, National Internet Registries (NIR)s, or Internet Service Provider (ISP). NIRs operate primarily in the APNIC region, but there are some in LACNIC too. RIRs are formally recognized by the Internet Corporation for Assigned Names and Numbers (ICANN). Historically, RIRs predate the ICANN, so the RIRs (at the time APNIC, ARIN and RIPE NCC) formally joined the structure of ICANN via the Address Supporting organisation (ASO) proposed in a Memorandum of Understanding (MoU) [12]. The ASO is responsible for recommendations regarding IP address policy and advises the ICANN board. After LACNIC was founded and recognized in 2003, the RIRs formed the NRO to coordinate on important matters of a global nature [13]. NRO, standardized the format to exchange transfers and statistics. In 2004, a new MoU was signed between the NRO and ICANN, replacing the previous MoU between the three RIRs and ICANN. This formalized that the NRO will fulfil the duties of the ASO. In 2005, AFRINIC joined the NRO following its formal recognition [14].

While the region in the regional internet registry might imply a limited scope, an RIRs typically serves an entire continent or a significant portion thereof. There are currently five RIRs. AFRINIC performs duties for Africa, ARIN serves North America, APNIC helps Asia, LACNIC aids South America and RIPE NCC benefits Europe. Figure 2.3 illustrates the geographical distributions of the five RIRs.



Source: `https://en.wikipedia.org`

FIGURE 2.3: Map of RIR regions

2.3.1 WHOIS

WHOIS is a protocol for querying databases for information about the registrant of a certain IP address or range of addresses or domains [15]. The current iteration of WHOIS is described in RFC 3912 [16]. The WHOIS data will serve as the basis for the research and will be enriched using all the data sources used by `rir-data.org`. As WHOIS' roots lie in the ARPANET (a precursor to the Internet) days, the protocol has several shortcomings in internationalization and security [16], [17]. Punycode encodes Unicode domain names for compatibility with ASCII systems [18]. This workaround has to be used for domain names

using non-ASCII characters. While ICANN has voted on phasing out WHOIS, RIRs still provide bulk collection of their current WHOIS database [19]. WHOIS data can be stored in a *thin* or a *thick* data model.

A thin data model stores minimal information, such as the domain name, registrar ID, and referral to the registrar's WHOIS server [20]. A thick data model stores all the registrars' full WHOIS information within a Top-Level Domain (TLD). Thin models are essentially redirects to the server responsible, requiring two WHOIS queries, whereas thick models only require one server to be contacted. Additionally, thin models lose their data if a registrar goes out of business, whereas thick models retain it. Thick models are usually faster and more complete than thin models.

Whether a domain uses a thin or a thick model depends on the policy of the Top-Level Domain (TLD). For example, `.com` and `.org` operate a thin data model, requiring the domain registrars to maintain the WHOIS information of its customers. While `.org` operates a thick data model. Country Code top-level domains like `.nl`) have their own policy. ICANN has specified policy that all TLDs are required to move to a thick data model [21].

Appendix A.1 presents an example of WHOIS output for the `utwente.nl` domain. It presents domain details, starting with the domain's status, followed by registrar and abuse contact information for addressing malicious activities. Sequentially, it documents the domain's creation and last updated dates. Furthermore, it includes indicators regarding DNSSEC support and enumerates the IPv4 and IPv6 addresses of its nameservers. Finally, the section concludes with details about the maintainer and a copyright notice.

2.3.2 Delegation files

There are two types of delegation files: Extended and Basic. RIRs already had a history of publishing delegation statistics. However, to standardize this format across RIRs, the basic statistics format was unified in 2004 [22]. The format of these files is standardized in the RIR Statistics Exchange format. Between 2008 and 2010, APNIC started testing an extended format version [23].

This extended format includes resources allocated to the RIR but not allocated to an organisation and reserved resources. These resources are in transition, either in the process of being delegated or returned to the pool of RIR resources. The extended format adds an *opaque-id* to the allocation records. This allows the trace of multiple allocations to the same organisation [24].

The general file structure starts with a header line, a three of summary lines, followed by the delegation records.

The header describes the version of the format, the registry that published it, and the file's serial. This is followed by the number of delegations present in the file, the start date and end date, and UTC offset, as shown in Listing 2.1.

Listing 2.1: Header example

```
1 version|rir|serial|count|startdate|enddate|UTC-offset
2 2|ripencc|1700693999|246568|19700101|20231122|+0100
```

The summary lines count the number of IPv4, IPv6, and AS number allocations and end on the string "summary" to distinguish between a summary and a record line. An

example can be found in Listing 2.2.

Listing 2.2: Summary example

```
rir|*|type|*|count|'summary'
ripencc|*|ipv4|*|92704|summary
ripencc|*|asn|*|45615|summary
ripencc|*|ipv6|*|108249|summary
```

The record lines describe the registry, the country code according to ISO 3166, the allocation type, the start number, the value, the date the allocation was made, the status, and the opaque ID [23]. Allocation types include IPv4, IPv6, or Autonomous System Number (ASN). IPv4 allocations specify the starting address and the number of addresses included, whereas ASN allocations denote the initial ASN and the quantity of ASNs allocated. In conclusion, timestamps indicate the epoch data and the allocation status.

The status can be available, allocated, assigned, or reserved. Available means it is not assigned or allocated, and the resource is still in the RIR pool. Assigned means an organisation can use this resource but not sub-assign it to other parties. Allocated means that a LIR has the resource and can allocate or assign it to its end users. Reserved means that a resource is not allocated or assigned. This can have multiple reasons, such as space reserved for the growth of an ISP, reserved addressed for experimental services, or returns that are not yet cleared and available for reassignment [25]. For an example featuring the IPv4 and IPv6 registrations of the University of Twente, see Listing 2.3.

Listing 2.3: Delegation record for the University of Twente

```
rir|Country code|type|start|count|date|status|opaque-id
ripencc|NL|ipv4|130.89.0.0|65536|19910412|assigned|
   cfad7b98-b8d3-4599-849a-67cc3d07dceb
ripencc|NL|ipv6|2001:678:d0::|48|20151223|assigned|
   cfad7b98-b8d3-4599-849a-67cc3d07dceb
ripencc|NL|ipv6|2001:67c:2564::|48|20111104|assigned|
   cfad7b98-b8d3-4599-849a-67cc3d07dceb
```

2.3.3 Reverse DNS

We generally speak about forward DNS, this system looks for the corresponding IP address when queried with a domain. For example, we use it When we want to navigate to `utwente.nl`, DNS translates this domain into an IP address to visit. Reverse Domain Name System (rDNS) works according to the same principle, only in the other direction. It has several use cases, such as authentication of mail servers or making `traceroute` output more human-readable. RIRs store this information is zone files per octet. Listing 2.4 illustrates a record from the `130.in-addr.arpa-RIPE` file.

Listing 2.4: Reverse DNS zone file snippet for University of Twente

```
domain | time-to-live | Record type | hostname
89.130.in-addr.arpa.   86400    NS   ns1.utwente.nl.
89.130.in-addr.arpa.   86400    NS   ns2.utwente.nl.
89.130.in-addr.arpa.   86400    NS   ns3.utwente.nl.
```

If the domain contains `in-addr.arpa` it is for lookups in IPv4, if it contains `.ip6.arpa` it is used for IPv6 lookups. Each RIR publishes these zone fragments on their FTP sites. Each line of these files contains a Pointer (PTR) record. A record starts with the IP address in reverse, the Time To Live (TTL), the record type, and the domain.

Chapter 3

Related Work

This chapter discusses related work for graph databases, GraphQL, and RIR data. We selected papers by searching for relevant terms in Google Scholar. After selecting a paper for further reading, papers were further filtered on relevance based on the introduction, methodology, discussion, and results. A paper's relevancy is judged on its insights and theories that support or contrast the research goals. Should a paper be discarded, the references were checked for any potential papers to include. When a paper passed all criteria, it was summarized and discussed below.

3.1 Graph Databases

In this section, we refer to the data model in the abstract rather than the specific data model we developed for the system architecture. We discuss various benchmarking tools and examine the landscape of Graph databases. Finally, we compare the performance of various database systems to aid in selecting the system architecture.

The HEAP project serves as a cornerstone for our research, demonstrating both the strengths and limitations of graph databases in a network analysis context. We build on their findings to identify suitable databases and optimize their use.

There are several things to consider when looking for the right graph database implementation. What conceptual model to use, Resource Description Framework (RDF) or Labeled Property Graph (LPG).

In 2009, Bader et al. defined the first version of HPC-SGAB [26] to develop a standardized method of testing graph databases to enable direct comparison to compare their performance. This paper describes a benchmark by measuring the Traversed Edges per Second (TEPS), which generates a sparse graph using `R-MAT` algorithm [27]. The benchmark focuses on shortest-path calculations. However, this paper aims to do queries focused on finding neighbourhoods more than doing path calculations. While these measures do not directly translate to useful benchmarks for the solution we will build, they allow for initial filtering.

Dominguez-Sal et al. [28] applied *HPC-SGAB* first to Neo4J, DEX, HypergraphDB, and JENA. In this paper, the authors measure the loading time of graphs generated by the HPC-SGAB and the shortest path calculations. From this test, DEX achieved the best performance, followed by Neo4J. Additionally, Ciglan et al. designed the first graph traversal benchmark [29]. The authors propose a benchmark based on breadth-first traversal and operations requiring full graph traversals.

One of these operations is community detection. Community detection is a relevant measure for the system this paper explores. This measure detects densely connected groups of vertices within a graph. In the context of this research, this allows for complex network analysis. Additionally, it can provide insight in resource allocation and policy analysis and assist with anomaly detection. Neo4J, DEX, OrientDB, NativeSail, and the experimental SGDB were benchmarked in this comparison. The benchmark considers loading times, three hops Breadth-first Search (BFS) operations, and computation of connected components for both in- and out-going edges. SGDB performed best in all tests, followed by Neo4J. Only in the loading benchmark was Neo4J's performance surpass by NativeSail. DEX interestingly did not complete the loading nor the three hop BFS for the larger graph sizes. However, the author does not give an explanation for this outlier.

Jouili et al. [30] proposed a benchmark by reviewing performance under concurrent use. The authors used their benchmark to compare any Blueprints-compliant graph database. Blueprints is a property graph model interface that provides a standardized way to interact with graph databases and processing systems. This enables developers to write system-agnostic applications working with different graph databases. Blueprints have since been integrated into Apache's TinkerPop [31].

In this benchmark, the authors measure loading time, traversal workload of shortest path and neighbourhood exploration (limited depth BFS), and an intensive workload. In the intensive workload, the authors instructed different clients to simultaneously fetch several vertices, update properties, and add an edge. They considered Neo4J, Titan, OrientDB, and DEX in their testing. Regarding insertions, DEX & Titan performed best at larger sizes. Depending on the buffer size, Neo4J would perform better until a certain threshold was reached, at which point execution times jumped up, and DEX achieved the best performance. For shortest path search and breadth-first search, Neo4J performed best, with DEX & Titan reaching second place dependent on the number of hops.

In 2023, Besta et al. analysed the graph database's landscape [4]. This paper considers all systems that can be used as graph databases, ranging from RDF systems implemented in traditional relational databases to the different types of NoSQL databases. The family of NoSQL databases consists of RDF databases implemented as NoSQL, Document, Tuple, Key-Value, Wide Column, and native graph stores.

The authors also include a section, 'Insights for practitioners', to discuss performance in general. While they initially start out preferring RDBMS above native graph database designs, the authors acknowledge the use case of native graph systems depending on the workload. Regarding RDBMSes, they listed several interesting concepts, ranging from SQL translation layer [32], a fork of existing RDBMS [33], graph implementation layer inside existing databases [34]–[37].

The evaluation of Neo4J and Spark/GraphX presents varying findings. Research outcomes regarding the comparative performance of GraphX against Neo4J are inconclusive. Ali et al. [38] measure the performance of the PageRank algorithm for Neo4J and GraphX. PageRank is an algorithm that determines the importance of a website by looking at the web pages that link to that website. The more links, the more important that website is, and this algorithm is the foundation of the early iterations of the Google Search engine.

In this paper, the experiment measures the execution time of PageRank on the Yelp dataset. They claimed that GraphX is around 12 times slower than Neo4J on both input sizes. The input size scales nearly linear with the increase of execution times. However,

the authors acknowledge that from a programmer's perspective, GraphX implementation is more efficient, which is attributed to the fact that Python code for Spark is deemed more readable and easily comprehensible compared to the Cypher query language used in Neo4J.

On the other side of the performance spectrum, Ballas et al. [39] applied PageRank to five other datasets provided by Stanford's Network Analysis Platform (SNAP). All datasets are smaller in the number of nodes, but the LiveJournal dataset is larger in the number of edges by roughly two times. In this experiment, Neo4J fails to complete the PageRank algorithm on the LiveJournal dataset. The authors note that Neo4J failed in the graph creation step after the data was loaded.

GraphX outperformed Neo4J by a small margin in three of the four datasets that Neo4J did manage to load. The larger the dataset, the better GraphX performed compared to Neo4J. In the smallest dataset, Neo4J had the fastest execution time.

Kalogeras et al. [40] explore Neo4J and GraphX performance in community detection. In this paper, the authors conduct experiments using the Label Propagation Algorithm (LPA) on three datasets: Digital Bibliography and Library Project [41], YouTube, and LiveJournal. SNAP provided all these datasets, and the authors measured the LPA execution time for each dataset. For DBLP and YouTube, Neo4J was faster by a small margin. GraphX was roughly twice as fast for the LiveJournal dataset.

Our project relies most on loading and community detection. Hence, we put the focus for selecting a system on loading and community detection benchmarks. Another emphasis of the system is path computation. These insights were incorporated in the database selection.

3.2 GraphQL

While there has been relatively little research regarding the application of GraphQL, none focused, understandably, on providing API access to RIR data using GraphQL. Our adoption of GraphQL is driven by the need for more efficient and expressive querying mechanisms, as observed in the HEAP paper's extensive use of labels on edges to store metadata.

Brito et al. [42] conducted an assessment of APIs that moved from REST to GraphQL, resulting in a substantial reduction in network load, reducing the number of fields by 94% and the number of bytes by 99%. Additionally, GraphQL enabled the API to reduce the number of calls needed to obtain the required data. GraphQL manages these significant improvements in two ways. First, it supports a hierarchical data model, reducing the number of endpoints with which clients must interact. This feature reduces the number of queries and, thus, the network load. Second, the data saving comes from the *client-specific queries*, meaning that clients only receive data they ask for, minimizing the number of fields returned and thus lowering the returned number of bytes.

Vogel et al. describe several challenges one should consider when moving from REST to GraphQL [43]. Denial of service due to overly complex queries must be accounted for. Other lessons mentioned, while useful, do not apply to the system proposed by this paper as it is a new implementation. Issues that arise from transforming existing implementations to GraphQL are thus less relevant. While there has been interest from the scientific community surrounding GraphQL, as noted by Quiña-Mera [44], there is still room for more validation in other realistic use cases aside from Facebook's use case.

3.3 RIR data

Various research has been done in parsing RIR data. We first review papers that parse RIR WHOIS data and conclude with papers that use RIR WHOIS data. Like the HEAP project, our work leverages RIR data to uncover patterns and detect anomalies between operational and administrative data sources, further validating the effectiveness of graph databases in this domain.

Beverly [45] parsed RIR data by ingesting the bulk WHOIS services provided by the RIRs to geo-audit address registrations. The paper proposes skipping prefixes a specific RIR does not manage. These prefixes are generally listed for completeness but carry a *not-managed-by* notice. This means that a prefix is listed in the WHOIS database of RIR_x while being managed by RIR_y. When combining data from all RIRs, these entries must be filtered out.

Additionally, the author identified particularities when parsing the data, such as the circular references and prefixes listed by an RIR but managed by another. These circular references are especially troublesome when storing RIR data in a graph database, as one needs to keep cycle detection in mind. The author concluded that time differences in the data dumps might cause these errors. However, upon scrutinizing the timestamps provided by the RIR, it was determined that they indeed contained errors in some instances.

Nemmi et al. [46] parsed the daily delegation extended form files and BGP data from CAIDA's *bgpstream*. This was done to compare the allocations stored by RIRs with the actual operational network of BGP in case there are duplicate records. The author also proposes to use regular delegation files when extended delegation files have missing records.

The authors also restored various artefacts in available data by filling registration gaps, if an AS appears in $date - 1$ and $date + 1$, it is assumed that the AS also exists at $date$. In some cases, the authors also restored the registration date to the date that the AS was first registered. In specific files, inconsistent dates were used, and they contained placeholders or earlier dates in later files compared to earlier files.

Streibelt et al. [47] devised a system to provide bulk historical WHOIS data to researchers. At the time, Team Cymru already provided a bulk WHOIS service; however, this was restricted to *current* data. However, the data on which Streibelt's system relies is inconsistent, as proved by Arouna et al. [10].

Previous studies show that parsing WHOIS data presents unique challenges and intricacies. Arouna et al. shows how different RIRs use different labels for the same data point. RIPE NCC, ARIN and LACNIC use the label '*created*' for the creation date, APNIC calls it '*last-modified*' and AFRINIC uses '*changed[0]*'. The authors consolidated data from the WHOIS snapshots, delegation files, and reverse DNS zones at the RIR level. The data is made available as an S3 object storage [48], and their website provides a sample code for getting started.

Cai et al.'s [49] paper analyses the relationships between Autonomous Systems and organisations. This paper provides valuable insight into linking different ASes to organisations. The study identified several ways to link different ASes back to the same organisation. The authors accomplish this by clustering various data points; organisation ID, phone numbers and email domains.

The Internet Yellow Pages of the Internet Health Report does similar things to this paper's objectives. It shows prefixes, ASNs, origins, organisations, and countries in a graph format. This project crawls various data sources. It parses data from Cisco Umbrella, Cloudflare Radar, IANA's DNS root zone files. They also include RIPE NCC's AS names, ROA, Atlas Probes and Atlas Measurements as input dataset alongside APNIC's population estimates dataset [50], [51]. This study focuses on RIR data from the administrative perspective instead of the operational perspective of IYP and provides more longitudinal access.

In 2015, Schlamp et al. [52] aimed to detect BGP hijacks by parsing RIPE NCC's Internet Routing Registry (IRR) database to classify benign sub-Multiple Origin Autonomous System (MOAS) event from malicious ones by looking at business relations. SubMOAS events can be announcements of rogue ASes where they advertise a route to a prefix fully contained within a prefix of a legitimate AS. The unsolved challenge is detecting whether a subMOAS event is benign or malicious. The paper argues that while a malicious party might announce a shorter prefix, they cannot alter RIPE NCC's objects without having the proper credentials.

Hence, they can classify the legitimacy by looking at the IRR data. The paper also explores designing a well-suited data model for a graph database. The paper describes storing the *MNTNER*, *ORGANISATION*, *INETNUM*, *AUT-NUM* and *ROUTE* objects and constructing a graph using these values as vertices. Schlamp et al. (2016) [1] extended their previous study by parsing the IRR databases from the RIRs to store relevant data for the BGP hijack alert classification system. Implementation details of their graph database are not shared.

Chapter 4

Methodology

This chapter presents our adopted research methodologies. First, we discuss our database choice and how we selected the queries. Lastly, we discuss high-level implementation choices.

4.1 Database selection

Guided by the performance issues faced by the HEAP team, we carefully select graph databases that promise improved efficiency and scalability for handling large datasets. We considered the following factors while choosing the adopted database:

1. *Scalability*
   Scalability comes in two types, such as vertical and horizontal scalability. Vertical scaling means adding more hardware to the machine. For instance, a CPU with higher clock speed, more cores, more RAM, or faster storage. Horizontal scaling means adding more instances or nodes to a system and distributing the load across multiple machines. We focus on horizontal scalability in the database choice.

2. *Performant under daily writes*
   The system needs to keep performing well under daily writes. The chosen solution needs to be able to deal with an increasing amount of data, as RIR data keeps increasing by the day.

3. *Supporting of comparisons in the spatial and temporal domain*
   This refers to the system's capability to compare data points with each other; additionally, it needs to be capable of comparing between different dates.

These factors are relevant to the project because of the very nature of RIR data. There is a significant amount of data produced on a daily basis. This invites the use case of comparing different dates with each other.

We used the above-listed factors to decide which database system to use. There are a multitude of options available, so we constructed a shortlist based on available literature. We then apply these factors to the shortlist and choose our database accordingly. We discuss the database systems mentioned in the Related Work section in the Implementation section. We chose to only review free editions or community editions. We did not want to encounter limitations that restrict our use.

4.2 Query selection

Our query selection process is influenced by the types of queries used in the HEAP project, with a focus on optimizing performance for both simple and complex queries. We adopted an interview approach to determine the researchers' need to design and implement a system to support the researchers in their needs. We designed a small survey outlining the high-level design of the system. We asked the following questions after explaining the benefits and limitations, such as the performance of aggregate queries:

1. What kind of relationships between entities would you query?
2. What kind of temporal relationships or longitudinal developments would you query?
3. What typical graph analysis would be useful? (e.g., strongly connected components?)

We emailed this survey to a group of network measurement researchers and collected their responses. We reached out to Johann Schlamp, one of the authors of the HEAP paper [1], and Oliver Hohlfeld, a professor at the University of Kassel with a focus on network measurements. We contacted Kimberly C. "KC" Claffy, the director of CAIDA, as well as Thomas Krenc, postdoctoral researcher at CAIDA. We reached out to Cristian Hesselman, the research director at SIDN Labs, and Moritz Müller, a research engineer at SIDN Labs. Cristel Pelsser, a professor at UCLouvain, and Shyam Krishna Khadka, a PhD candidate at the University of Twente, were also contacted. Additionally, we reached out to Taejoong (Tijay) Chung, an assistant professor at Virginia Tech, and Romain Fontugne, the deputy director at Internet Initiative Japan and maintainer for Internet Health Report and Internet Yellow Pages. We filtered the responses for queries we had already implemented and collected the remaining queries to discuss among ourselves. Then, we investigated whether the query was applicable to a graph system and constructed an implementation priority list based on the number of researchers requesting it and its perceived usefulness. Some requests, while very interesting, were too time- or resource-consuming to implement. Interestingly, RPKI was a much requested feature. However, we could not include this feature due to the lack of data in our data source and time constraints. Nevertheless, we aim to include this in our future work.

4.3 Implementation choices

We implemented the system as a `podman` container. While this functions as a constraint, it does allow this project to be run and maintain consistency across systems and architectures.

We implemented the system using `Python` as that language is popular among data scientists. It is a language the is efficient to prototype in and provides additional functionality by libraries. We developed the library in a `PySpark Jupyter notebook`. This supported our agile and iterative approach to development. We also think it is likely that researchers using our project will likely use `Jupyter notebook` too to perform their analysis, so it was important to us that it worked correctly in that environment.

We developed the code iteratively in a `PySpark Jupyter notebook`. Once the code worked, we moved it into its modules and generalized the code as much as possible to encourage re-usability. In order to make the code as usable as possible, we used *docstrings*, type hinting, clear variable naming, and modular functions. We ran our benchmarks inside `Jupyter notebook`s too and provide the code to run the benchmarks yourself in the `checker.py` module.

While we discussed GraphQL in earlier chapters, we did not implement it and considered it out of scope. We retain the ease of use by providing the project with well-documented Python modules that can be used in a `PySpark Jupyter Notebook` .

Chapter 5

Implementation & Design

This chapter discusses the research results and answers the questions in the Research Questions section in the order in which we posed them.

5.1 Suitable graph database

Reflecting on the HEAP project's use of Neo4j, we evaluate various graph databases to find a balance between expressivity and performance. As mentioned in the related work, we intended to evaluate alternative solutions to graph databases as proposed by Besta et al. [4]. We evaluated those examples based on the available literature and online material and reviewed them using the factors discussed in the methodology.

Emptyheaded [35] boasts excellent performance, but is not in active development (last commit seven years ago at the time of writing) and has no active community [53]. GRAIL [32] is a translation layer between graph-specific queries and translates them to SQL. This project has the same problems as EmptyHeaded [54]. Additionally, Vertica [36] did become an active product after its paper. However, it is limited to single host machines and can only load data up to 1 TB of storage [55]. This storage limitation would be encountered quickly, as a full day of data is around 130 GB when stored as `gzip` compressed `parquet` files. This limitation would be hit after downloading six days of data.

We also investigated the performance of several native graph databases. We reviewed Neo4J, Spark, DEX, HyperGraphDB, Jena, OrientDB, NativeSail, SGDB and Titan. We put emphasis on benchmarks that review loading and community detection, as these measures apply most to this project. After this evaluation, the shortlist came down to Neo4J and Spark. In performance comparisons, both systems showed promising numbers. We adopted Spark because of ease of development and performance considerations. We set up a `podman` container to run `PySpark` with `Jupyter notebook` [56], [57]. The system was tested on a machine with 50 CPU cores and 256GB of RAM allocated to the Spark instance. We selected `Spark` to function as our graph engine. `Spark` is not a database in the traditional sense. Its distributed computing system allows for large-scale data processing and analytics. We store processed data on disk to be retrieved to construct a graph on demand.

5.2 Suitable use cases

Following the HEAP team's success in using graph databases for network analysis, we identify use cases that benefit from the strengths of graph-based approaches. We expand these queries with queries obtained from a survey. These queries were extracted from

Table 5.1: Results of the survey

Query	Number of researchers requesting	Opinion of expert supervisors	State of implementation
Relationship between prefixes	4	yes	yes
RPKI	3	no (scope)	no
Frequency of change	2	no (resources)	no
Changes over time	1	if possible	partly
Sub-graph extraction	1	if possible	yes
Change of prefix ownership	1	no	no
Roaming prefixes	1	if possible	partly
Degree of vertices	1	yes	yes

discussions and email chains with researchers active in the network analysis field. We selected queries based on their applicability to graph analysis. Several queries that focus on aggregates or are otherwise better served by a RDBMS were not implemented.

After evaluating the queries, we realized that most demand focused on exploratory research. We still contribute various queries that allow spatial and temporal comparison, but the majority of queries focus on supporting exploratory research. When conducting our survey, several researchers responded with surprise. They did not consider applying graph database to RIR data. Hence, some answers are better suited to a relational database implementation. For an overview of the survey results, see Table 5.1. These queries from the basis to evaluate the system. Together with the expert supervisors, we added several queries based on our experience with exploring the data so far. The complete list of queries implemented can be found in Appendix B.

5.3 Suitable data model

Our goal is to create semantically meaningful edges to capture the relationships between the four vertex types. Taking inspiration from the HEAP data model, we refine our approach to reflect the data we have available while maximizing its usefulness for exploratory research. We construct the edges as follows:

1. **Maintainers and Prefixes:**
   An operator or maintainer is an entity responsible for administrating a network prefix. Since maintainers manage the prefixes, it is logical to establish edges between maintainers and the prefixes they control. These links allow for traversal and querying of which prefixes are managed by a particular maintainer and vice versa.

2. **Prefixes and Origins:**
   An ASN or origin is responsible for announcing network prefixes. We allow for traversal by creating edges between prefixes and the ASNs that announce them. This helps in understanding which origins are associated with specific prefixes.

3. **Origins and Organisations:**
   Origins (ASNs) are associated with specific organizations. Establishing edges between origins and their respective organizations allows us to explore this relationship. This link enables us to trace the ownership and gives insight into the control of network resources.

We constructed links bidirectionally to support `GraphFrames`' graph analysis algorithms. As these algorithms operate under the assumption that edges are directional. We propose the data model 5.1 to represent RIR data in graph form.

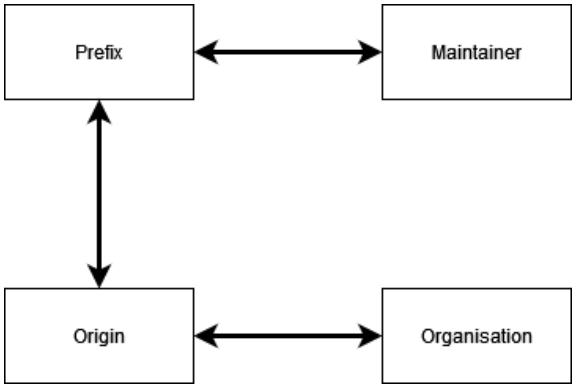


FIGURE 5.1: Schema of the data

5.4 Implementation details

Our implementation aims to address the performance bottlenecks experienced by the HEAP team, aiming for a more streamlined and efficient system. This section discusses implementing the Ingester, the Graph Creator, and the Query library. These were implemented using the `GraphFrames` library on top of `Spark`. `GraphFrames` work together effectively with `DataFrames` which are the basis of the `rir-data.org` dataset. `GraphFrames` is essentially the same as `GraphX`, adds some extra compatibility (support for `Python` and `Java` besides `Scala`). It misses the partitioning from `GraphX` but exchanges that for `Spark Catalyst`'s graph-based optimizations.

5.4.1 Ingester

The ingester has two parts, with a local caching step between the two steps. First, we retrieve the data for a specified date provided by `rir-data.org` to ingest. We start by dropping columns that have no purpose to increase our space-efficiency. We also rename the columns '*last-modified*' and '*mnt-by*' to their '_' counterparts (i.e. '*last_ modified*' and '*mnt_ by*') to make the column names more SQL-friendly. Once these columns are removed and renamed, the `DataFrames` are written to disk and reloaded to prevent re-fetching data.

In the second part, we start pre-processing by unwrapping the lists that contain the prefixes. For each value in the list of prefixes, we create a new row with the same maintainer, organisation, and origin. After unwrapping these lists, we drop prefixes without semantic meaning for the graph. We removed `0.0.0.0/0`, `10.0.0.0/8`, `172.16.0.0/12` and `192.168.0.0/16` because these prefixes either contain the full IPv4 address space or they represent private ranges. We also remove rows that are managed by other RIRs. When a redirect prefix is present, it is stated in the '*descr*' column that this *IPv4 address block not managed by %* where % is the RIR that does not manage it but does retain it in its WHOIS data.

While ingesting the data, we realized that most values in *origin* were `Null`. Only ARIN stores the 'origin' value in their WHOIS data; in all other cases, this is stored in ROUTE data and thus inaccessible via the `rir-data.org` dataset. As a result, we enrich the data by adding origins to the rows before storing the `DataFrame` to disk again for graph creation. We add this by downloading the relevant Route Views RIB file. Using this file and the 'start_address' value, we look up the origin with the `pyasn` module [58]. If an origin was added by this process, we set `True` in the '*external_origin*' column to indicate that the value was obtained from another source. See Fig. 5.2 for a visual representation. While this method does not fully resolve the issue, the following subsection elucidates how vertices are generated for origins that still possess a `Null` value.

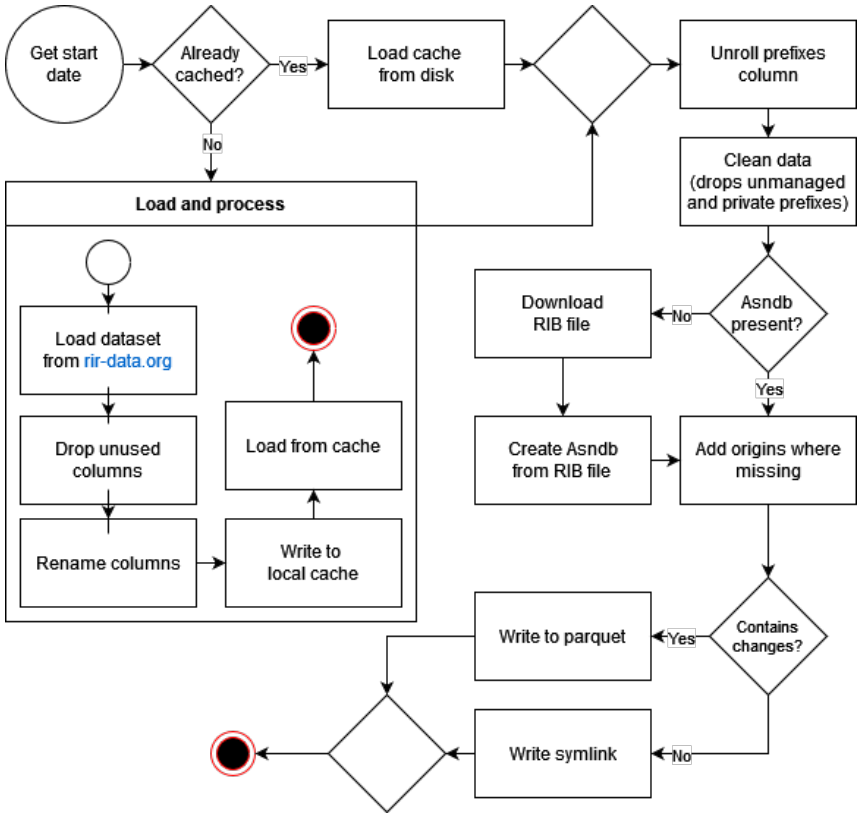


FIGURE 5.2: Flowchart of the ingester

5.4.2 Graph creator

The graph creator loads the `DataFrame` of a date specified by the user. First, it verifies whether the date is valid. The date must fall between 2022-11-01 and the current date. It also checks whether the data has been ingested yet because there is one day that has no available data (2023-07-26), and the two or three days leading up to the current date are not available as they are still processing on the `rir-data.org` side.

After loading the `DataFrame`, it generates IDs using the `sha256` hashing to guarantee unique IDs across multiple machines. The current implementation was tested on a single host server with sufficient resources. We designed the system with distributed `Spark` functionality in mind, should future researchers want to make use of it.

The code then creates a vertex `DataFrame` by taking all IDs with their relevant values and a boolean for '*external_origin.*' This column can only be valid for origin ver-

tices, and all other type vertices store a `False` for this column. If an origin value is *Null* even after adding origins in the `pyasn` step, it is replaced here with the string `{prefix-maintainer-organisation}`. This approach resolves the issue of null values while retaining semantic values.

Lastly, it creates an edge `DataFrame` by taking all prefixes' IDs and linking them to maintainers, maintainers to origins, origins to prefix, and origins to organisation. The vertex and edge `DataFrames` are passed on to the `GraphFrame` constructor to build the graph [59]. See Fig. 5.3 for a visual representation. Finally, this step returns the graph object to apply queries to or perform filtering using the query library.

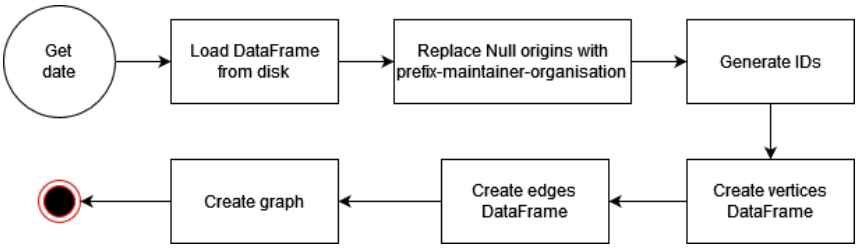


FIGURE 5.3: Flowchart of the Graph Creator

5.4.3 Query library

We implemented all the queries in the query library, which enabled us to implement several queries to facilitate exploratory research. Plain queries such as filtering for a certain value or list of values in the '*prefix*', '*origin*,' '*maintainer*' or '*organisation*' column. It can also find prefixes belonging to or maintained by a specific '*origin*,' '*organization*,' or '*maintainer*.' We also support finding links between origins by checking if they have a shared 'maintainer' or 'organisation.' The library also supports more advanced graph queries such as applying PageRank, (Strongly) Connected Components, Community Detection, and BFS path computation. We discuss the queries more in-depth below.

Filters

Description: A filter query selects and returns nodes or edges in a graph that meet the specified criteria.
Example use case: We need to review all prefixes announced by a certain origin, maintained by the same maintainer.
Reasoning: Filters allow for a more nuanced look at the data. We also implemented `LIKE` queries as a convenience for the researchers.

Traversal

Description: A traversal query identifies and visits nodes that are 1, 2, or 3 hops away from a given vertex in a graph.
Example use case: We aimed to find the organisation responsible for a specific prefix.
Reasoning: Traversal allows for insight into which vertices are connected to a vertex. This can range from which origin is connected to a prefix to which organisation does a maintainer belong to.

Link vertices

Description: The shortest path query identifies the shortest route between two vertices in a graph.
Example use case: We need to determine if two vertices are connected by a chain of influence via their organisations.
Notes: Path computation tries constructing a path from $vertex_x$ to $vertex_y$. If a non-empty path is returned, there is a link between two vertices, which can be used to find origins that share an organisation or maintainer. **Reasoning**: This query allows researchers to find the underlying ('*hidden*') connections by performing a BFS to find a path between two vertices. As the edges in the path `DataFrame` only describe vertex IDs, we provide a function to rewrite the `DataFrame` to human-readable format.

Degree

Description: A degree query calculates the number of edges connected to each node in a graph.
Example use case: We can determine the most interconnected vertices of the graph by computing their degree.
Reasoning: This was one of the requests from the survey. Nodes with a high degree are often hubs or points of failure, making this query essential for understanding the structure and identifying key nodes. We provide methods for splitting by vertex types as well as sorting by degree count.

PageRank

Description: PageRank calculates the importance of each node in a graph based on the structure and connectivity of the network.
Example use case: We can identify the most influential entities by computing the PageRank of organizations within the network.
Notes: PageRank can identify which vertices are most important within the graph [60]. As the number of incoming edges is of high importance to the algorithm, this will favour maintainers with lots of small (i.e., /28-/32) prefixes more than maintainers with a few large (i.e., /16) prefixes. PageRank is natively implemented by the `GraphFrames` library, but we extended it by creating filters per vertex type. This way, researchers can look at the most important organisation or maintainer, rather than only seeing origins and prefixes.
Reasoning: It is particularly useful in identifying influential nodes within a network. We provide additional methods for splitting pagerank score by vertex type, as well as sorting.

Connected Components & Strongly Connected Components

Description: (Strongly) Connected Components identifies all sub-graphs where every node is reachable from every other node within the same sub-graph.
Example use case: Identifying connected components within the network graph allows us to detect isolated subnetworks.
Notes: The Connected Components and Strongly Connected Components algorithm identifies sets of vertices where other vertices can reach each vertex in the same set, enabling the findings of connected groups of prefixes, origins, maintainers, and organisations. In the context of network analysis, identifying these strongly connected components can help understand the network's structural organisation and operational dynamics.
Reasoning: This query is useful for understanding the overall connectivity and robustness

of the network. We provide methods to extract sub-graph based on component as well as analysis functions to look at component sizes

Community Detection

Description: Community Detection identifies and groups nodes in a network that are more densely connected to each other than to the rest of the network, revealing the underlying community structure.
Example use case: Community detection assists in identifying clusters of interconnected nodes within the network.
Notes: Community detection aims to identify groups of vertices that have a community structure. It uses the Label Propagation Algorithm (LPA) to identify these communities [61], which helps researchers to identify vertices that exhibit strong internal connections within the network.
Reasoning: This reveals the underlying community structure, which is essential for various analyses, such as detecting groups of ASNs that frequently interact or clusters of prefixes managed together. We provide methods to extract sub-graph based on community as well as analysis functions to look at community sizes

Comparing different dates

Description: Comparing two graphs gives insight into which connections are removed and which connections are added between the two dates.
Example use case: To detect unauthorized changes and monitor the evolution of the network, we need to review prefix-origin mappings across different dates.
Reasoning: Comparing two graphs gives insight into which connections are removed and which connections are added between the two dates. This is essential for temporal analysis, allowing you to track changes over time, understand network evolution, and detect anomalies or trends. We provide additional methods analysing which vertex types changed the most.

Chapter 6

Evaluation

This chapter presents the results of the queries and reviews the output, the runtime, and the usefulness of graphs compared to RDBMS. The system utilized a dual `Intel ® Xeon ® Gold 6258R` CPU for all our runs with 50 cores and 256GB of RAM. We do not show the output of results, as the resulting `DataFrame`s are very wide and cannot legibly be represented.

6.1 Ingester

We can process RIR data using the `process_rir_data_dot_org` method, which takes around $1 - 2$ minute(s) per day of data, 40% of which is downloading the RIB file and converting it to a file that `pyasn` can use. The resulting `DataFrame` is stored in the dataset folder.

6.2 Graph Creator

We can pick a date to load a graph of RIR data using the `load_graph` method. This process of retrieving the `DataFrame` from disk, assigning the vertex IDs, and constructing a graph takes about 3 minutes. The result is a `GraphFrame` that can be searched through or have graph analysis queries applied to it.

6.3 Query Library

The query library incorporates lessons from the HEAP project, focusing on achieving a balance between expressivity and performance. We leverage the query library to analyse network data, including reviewing administrative changes and tracing invalid RPKI announcements. We evaluate the suitability of our query library with our general understanding of relational databases. Unfortunately, there is no RDBMS-based implementation of our project to compare against at time of writing. So we evaluate the suitability compared to RDBMS on a theoretical level. We executed these queries on a graph built from data of `2024-05-17`. This graph contains 5.346.192 prefixes, 118.581 origins, 30.297 maintainers, and 2.656.558 organisations for vertices. It encompasses 30.801.088 edges.

6.3.1 Filters

Benchmark: We ran filters looking for values related to the University of Twente prefix. We filtered the source `DataFrame` and the `GraphFrame`. We also ran the `LIKE`-filter for '*?niversity?*', this filter looks for all vertices that contain that string with '*?*' as SQL

wildcards. In this example, we replace the *U* of *University* for a wildcard to capture both capitalized and uncapitalized spellings.
Output: A filtered `DataFrame`
Runtime: 10 seconds
Comparison to a RDBMS: $--$
Relational databases are optimized for selection and pattern-matching queries. The `SELECT` and `LIKE` calls we use in the `apply_filter_*` methods rely on these calls entirely.

6.3.2 Traversal

Benchmark: We picked 5 random values to evaluate the *Traversal*. This is accomplished by seeding `Spark`'s `sample` function [62] using `Python`'s `random` [63]. We ran the queries 5 times and averaged those values to reach the runtime values, as shown below.
Output: `DataFrame` containing the destination(s)
Runtime: 30 seconds (see Table E.1)
Comparison to a RDBMS: $-/+$
Depending on the distance traversed, this is more suitable to RDBMS than graph databases. For `first_degree_traversal`, RDBMS would perform better as it will be a single row in a link table. But graphs databases perform better for `second-` and `third_degree_traversal` as they are specifically designed for efficient traversal operations.

6.3.3 Link vertices

Benchmark: We ran the Connected Components algorithm before picking 4 differently sized components to evaluate *Link Vertices* queries. We pick the largest, the smallest and 2 random components. For each component, we pick 3 random vertex pairs. Otherwise, it would be highly likely we pick random vertices that are not connected at all. This would not net us relevant results. We tested using the following *maxPathLengths*: $5, 10$ (default) and 15 to give insight into running the *Link Vertices* queries on smaller and larger components. See Algorithm 1 for a pseudocode overview of the benchmark. We ran the benchmark 3 times and averaged the runtime values, as shown below.
Output: `DataFrame` containing the path between the two vertices (if present)
Runtime: 35 minutes (see Table E.2)
Comparison to a RDBMS: $+$
Breadth-first Search is a core strength of graph databases. This operation is inherently more efficient in a graph database due to its optimized vertex traversal.

6.3.4 Degree

Benchmark: We ran the `sorted_xdegree_split` function 3 times and averaged the runtime. We restarted the podman container between runs to bypass `Spark`'s caching mechanisms.
Output: `DataFrame` of vertices, with an added column containing the degree count.
Runtime: 10 minutes
Comparison to a RDBMS: $-/+$
While graph databases can compute degrees fairly quickly, it does depend on an aggregate `COUNT`. This operation can be done faster in RDBMS.

6.3.5 PageRank

Benchmark: We ran the `pagerank_vertices_split` function 3 times and averaged the runtime. We restarted the podman container between runs to bypass `Spark`'s caching mechanisms.
Output: `GraphFrame` with a '*pagerank*' value column for the vertices and a '*weight*' column for the edges.
Runtime: 1 hour
Comparison to a RDBMS: ++
PageRank is a graph-focused algorithm. While there are RDBMS implementations possible, a graph-related approach will perform better.

6.3.6 Connected Components & Strongly Connected Components

Benchmark: We ran the `connected_components` and `strongly_connected_components` functions 3 times and averaged the runtime. We restarted the podman container between runs to bypass `Spark`'s caching mechanisms.
Output: `DataFrame` containing vertices with an added column containing the '*component*' value.
Runtime: 1 hour
Comparison to a RDBMS: ++
The algorithms to compute the (strongly) connected component require traversal. Thus, a graph database performs better than a RDBMS.

6.3.7 Community Detection

Benchmark: We ran the `community_detection` function 3 times and averaged the runtime. We restarted the podman container between runs to bypass `Spark`'s caching mechanisms.
Output: `DataFrame` of vertices with an added column containing the '*label*' value.
Runtime: 80 hours
Comparison to a RDBMS: ++
Label Propagation Algorithm is specifically designed for graph structures. Graph databases can execute this query more efficiently than a relational database.

6.3.8 Comparing different dates

Benchmark: We ran the `community_detection` function 3 times, comparing 2 random dates of our available data.
Output: `DataFrame` containing all edges that are present in $date_1$ but not in $date_2$ and vice versa, with an extra column '*missing_from*' to indicate its origin.
Runtime: 10 minutes
Comparison to a RDBMS: −
Relational databases are highly optimized for data operations and complex queries with multiple conditions. While graph databases can perform these operations, relational databases offer better performance in this regard.

Algorithm 1 Benchmark *Link Vertices*

```
components ← graph.connected_component()
lengths ← [5, 10, 15]
sample_components ← components.sample().limit(2)
sample_components.append(components.max())
sample_components.append(components.min())
for component in sample_components do
    vertices ← graph.vertices.filter(component)
    sample_vertices ← vertices.sample().limit(6)
    sources ← sample_vertices[0 : 2]
    targets ← sample_vertices[3 : 5]
    for source, target in sources, targets do
        for length in lengths do
            path ← graph.link_vertices(source, target, length)
        end for
    end for
end for
```

6.4 Analysis of Findings

6.4.1 Temporal Data Comparison

We applied our `analyse_changes()` method to two dates, `17-03-2024` and `17-05-2024`. This method looks at the edges of the graphs for those two dates. The method determines the changes between the two graphs using the edges. This data is then used to construct a new `DataFrame` with all changed edges along with a label of the missing date. When looking into the values that changed the most between those two dates, as well as the change relative to the object and the change relative to its type. We picked random dates and created two time deltas; four weeks and eight weeks. We observed the following:

TABLE 6.1: Four weeks

Date	Type	Value	Number of changes	Change relative to self	Change relative to type
15-02-2024	Prefix	157.10.64.0/26	64	25%	0.02%
	Maintainer	PHIX-NOC-AP	3524	0.72%	10.56%
	Origin	15557	103303	69.80%	22.07%
	Organization	A1 Telekom Austria AG - Business Customers	1688	1.71%	0.82%
14-03-2024	Prefix	157.10.65.0/26	64	25%	0.02%
	Maintainer	PHIX-NOC-AP	2075	0.42%	6.77%
	Origin	12670	71161	100%	15.90%
	Organization	Linkt Customer	1476	10.66%	0.73%

PHIX-NOC-AP operates as a tier 2 ISP in the Philippines. Altice, associated with AS15557, is a prominent ISP based in France. Meanwhile, 1 Telekom Austria AG - Business Customers serves as an ISP in Austria, specifically targeting business connectivity needs. Linkt Customer, linked to Linkt SAS, functions as a tier 2 ISP.

TABLE 6.2: Eight weeks

Date	Type	Value	Number of changes	Change relative to self	Change relative to type
17-03-2024	Prefix	103.187.39.16/32	4	50%	0.003%
	Maintainer	PHIX-NOC-AP	6201	1.27%	9.11%
	Origin	9299	12401	1.27%	9.26%
	Organization	BELTELECOM MinskFilial branch Republic of Belarus	5299	1.27%	8.01%
17-05-2024	Prefix	103.187.39.240/29	4	50%	0.003%
	Maintainer	PHIX-NOC-AP	4475	0.91%	7.23%
	Origin	9299	8898	0.91%	7.30%
	Organization	Failover IPs	2045	2.55%	3.39%

We did not observe any anomalies in the prefix changes. Every prefix that changes has four changes associated with it. This is because of the data model, as can be seen in Fig. 5.1. We construct four edges, to and from the maintainer and to and from the origin. For the other changes, the values are more interesting. Origin AS9299 is associated with the Philippine Long Distance Telephone company, a major telecom provider in the Philippines and a tier 2 ISP. The maintainer for this AS is `PHIX-NOC-AP`. In Belarus, `BELTELECOM MinskFilial branch Republic of Belarus` operates as a large telecom provider. Meanwhile, AS16276 is linked to the organisation `Failover Ips`. According to `OVH`'s website, failover IPs are beneficial during service interruptions, major incidents, or when servers exceed their capacity [64]. These additional IPs, offered as a paid service, ensure consistent accessibility even in unforeseen events.

We generated a cumulative distribution function (CDF) graph to illustrate the frequency of changes. Additionally, we constructed a bar graph to analyse the occurrence of changes, categorizing the number of changes subdivided by vertex type for the initial 10% of the dataset. This dual approach provides a comprehensive view of change distribution and frequency across different vertex types within the subset of the data.

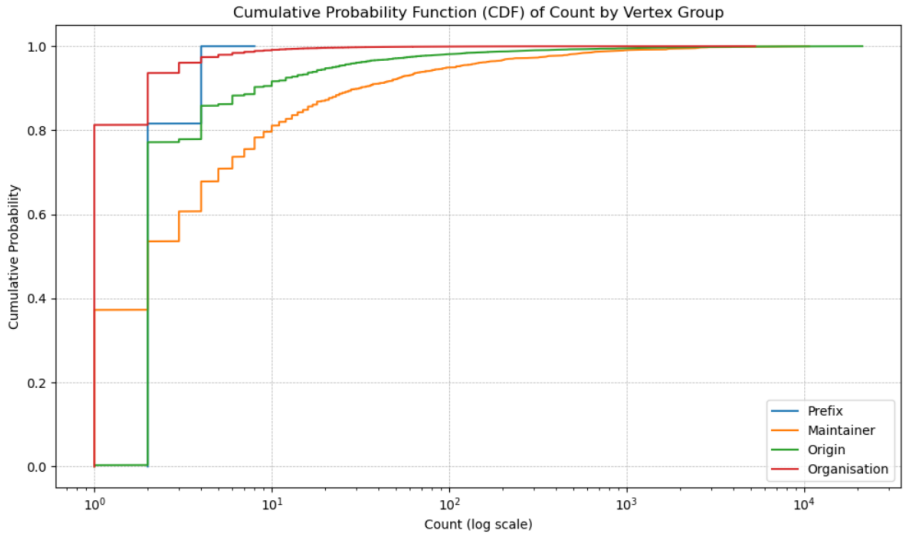


FIGURE 6.1: Cumulative Distribution Function of the number of changes

As we can see in the CDF graph, Fig. 6.1, the prefixes (depicted in blue) rises quickly and reach a cumulative probability of 100% at a relatively low count of changes. This

suggests that they are the most stable of the vertex types. The curve for maintainers (depicted in orange) shows a more gradual increase compared to the prefixes, indicating a wider distribution of changes. There is a noticeable step pattern, which might imply different levels or tiers of changes across the dataset. The origin curve (shown in green) starts with a slower rise but then accelerates more quickly, indicating there is a moderate number of origin vertices with few changes, while some have a significant number of changes. The curve for organisations (red line) is steep and reaches the cumulative probability of 100% quickly. This suggests that nearly all organisation execute very few changes, indicating a high stability similar to prefixes. In conclusion, prefix and organisation are relatively stable. The majority of vertices belonging to those groups experience few changes. Maintainers on the other hand show the greatest variability in terms of changes. The origins curve walks the middle between these two extremes, indicating moderate dynamics.

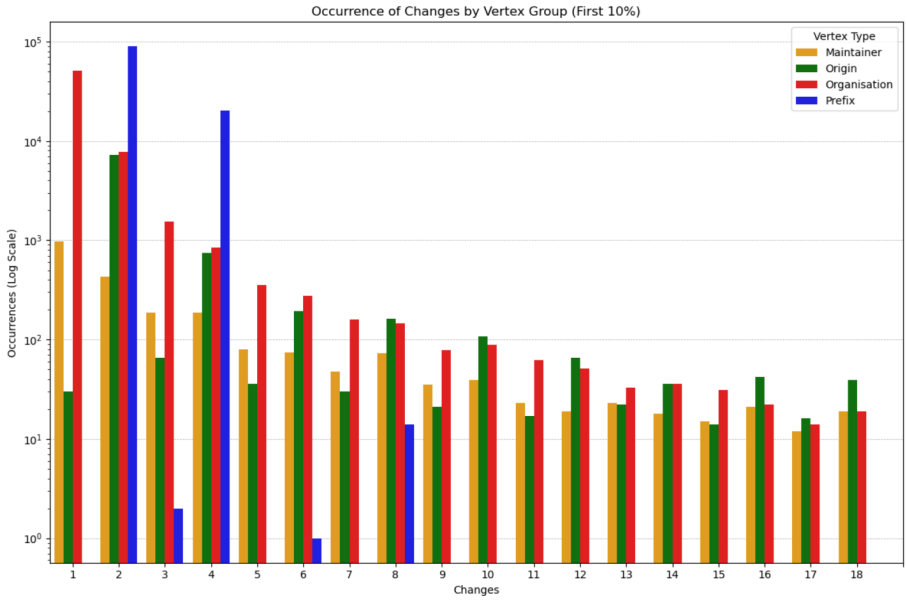


FIGURE 6.2: Occurrence of the number of changes

In Fig. 6.2, we see the occurrence of changes separated by vertex type (maintainer, origin, organisation, and prefix) for the first 10% of data. On the horizontal axis, we show the number of changes. While the vertical axis (log scale) represent the occurrences of these changes for each vertex type. Prefixes (depicted in blue) have a high frequency of changes at lower counts and completely stop after 8 changes. They also have the highest occurrence of changes compared to the other groups at 2 and 4 changes. This indicates that most prefix vertices undergo a few changes. For maintainers (depicted in orange) this distribution is more spread out, indicating variability in the number of changes. Origins (depicted in green) exhibit a broad distribution of changes, with a significant number of changes occurring for 2 and 4. Organisations (depicted in red) have a high frequency of change at $1, 3, 5, 6, 7$. Their occurrence of change is more consistent, spread across higher counts of change compared to other vertex types. This indicates that organisations often experience varied levels of change. In general, prefix and organisation vertices tend to experience fewer changes more frequently as opposed to maintainer and origin vertices which have a wide range of change frequencies. Along with the CDF graph, this visualization aids in understanding the dynamics of change and the stability of different vertex groups within the graph.

We demonstrate this query compares network data across different dates by analysing changes in prefixes, origins, maintainers, and organizations is valuable for both academic research and network operators. Academics can use it to study the evolution of internet governance, examining how administrative control and policies shift over time. This can reveal trends in ownership, regulatory compliance, and organizational changes. For operators, it helps in managing administrative responsibilities by highlighting updates in registry information, ensuring accurate records, and identifying potential compliance issues, thereby improving the integrity and reliability of network data management.

6.4.2 Linking Administrative Data with Operational Validity

In the work of Jaw et al. [65] the AS205220 announced several prefixes with invalid announcements of RPKI. These invalid announcements were mapped using `IP2ASN` to a different ASN than announced. We evaluate their undetermined finding by looking for registrations for the announced prefixes. For each `invalid_asn` announcement, we load the graph of that date and check the connected origin, maintainer, and organisation for the announced prefix. Out of the 410 records, 68 matched the criteria for further analysis. 47(69%) findings could not be classified. This indicates that they either did not have an active registration at the date in question or could mean that they did not have a registration on that specific day or the registration falls under a larger registration. For instance, an announcement of a smaller prefix than the registration. The other 21 (31%) registrations have been summarized in the following table:

Date	Prefix	Origin	Origin external	Maintainer	Notable organisations	Number of organisations
2022-11-28	45.45.152.0/22	205220	no	MNT-IL-845	IPXO, HKGO	11
	83.147.252.0/22	7029	no	IPXO-MNT CYBER-MNT	IPXO	20+
2023-01-16	185.81.217.0/24	*unlabeled*	yes	AZERONLINE-MNT	RH-NET	1
	46.23.100.0/22	15723	yes	AZERONLINE-MNT	Azeronline, RH-NET	8
2023-02-04	46.23.98.0/24	398343	no	AZERONLINE-MNT	Azeronline	17
	46.23.96.0/24	398343	no	AZERONLINE-MNT	Azeronline	17
2023-02-14	185.81.216.0/24	*unlabeled*	yes	AZERONLINE-MNT	Azeronline	1
	185.81.219.0/24	398343	no	AZERONLINE-MNT	Azeronline	17
	46.23.99.0/24	398343	no	AZERONLINE-MNT	Azeronline	17
2023-03-20	109.205.212.0/24	202335	yes	AZERONLINE-MNT	Azeronline	2
	188.64.12.0/22	17941	yes	AZERONLINE-MNT	Azeronline, Equinix	20+
2023-04-02	108.165.135.0/24	205220	no	MNT-IL-845	IPXO, HKGO	13
2023-04-24	50.114.212.0/22	205220	no	MNT-IL-845	IPXO, HKGO	9
	50.114.192.0/22	205220	no	MNT-IL-845	IPXO, HKGO	9
	50.114.244.0/22	205220	no	MNT-IL-845	IPXO, HKGO	9
2023-06-02	91.186.194.0/23	*unlabeled*	yes	IPXO-MNT CYBER-MNT	HKGO	1
2023-07-16	154.16.24.0/24	*unlabeled*	yes	NETSTACK-MNT	Digital Energy Technologies Limited	1
	188.64.143.0/24	834	no	SB-MNT	IPXO	20+
	196.251.251.0/24	40676	no	DAL1-MNT	Group8	20+
	104.234.155.0/24	205220	no	MNT-IL-845	IPXO, HKGO	10
2023-10-18	206.53.62.0/24	[30407,61317]	no	MNT-ONTAR-40	Velcom INC	2

Several values are particularly noteworthy. The entry `Azeronline-MNT` appears frequently; it corresponds to Azeronline Information Services, as ISP in Azerbaijan. `IPXO`,

based in the United States, is an IP address platform that offers services related to IP acquisition, management, security, and data intelligence. Their maintainer is `MNT-IL-845`. `HKGO` is an ISP for Taiwan, called August Internet. Additionally, RH-NET is recognized as a research and university network in Iceland.

As the findings were incomplete beforehand, this tool can support research in validating findings from the BGP or operational side of the internet by comparing it against RIR registrations. Academically, it supports research into internet security and governance by providing insights into the prevalence and causes of invalid route announcements, facilitating studies on the effectiveness of RPKI in securing internet routing. For operators, it enhances operational security by quickly identifying and addressing misconfigurations or malicious activities, helping maintain trust in network operations and ensuring compliance with routing policies. This capability improves the resilience and stability of network infrastructures by preventing route hijacks and other security threats.

Chapter 7

Discussion

In this chapter, we discuss the results of the research. First, we review our contributions and discuss two challenges we encountered while developing the system and their solution. Then, we discuss the trade-off decisions we made and the limitations of the project. Finally, we look at actionable recommendations and some improvements for future work.

7.1 Contributions

This research builds on the HEAP project's findings, contributing to the field by addressing its performance challenges and enhancing the usability of graph databases for network analysis. Our primary contribution to the field is demonstrating a graph-based data model for RIR WHOIS data works and provides several benefits. We provide `Python` modules that facilitate the conversion of data from `rir-data.org` into local storage formats. These modules introduce methods to convert the RIR data into graph representations. The graphs enable more powerful data analysis by leveraging the relationships in the dataset.

We offer a library with queries that can be used to perform exploratory research on the graphs. These queries are inspired by real-life examples to demonstrate practical use as well as outline future additions to the system.

7.2 Lessons learnt

Working with new data and tools always poses challenges and provides learning opportunities, and the subsequent sections discuss two of these challenges in further detail.

In our first explorations shows that the origin was present, which is an integral part of our input data. Later in the programming process, we discovered that most rows had a *Null* value in the origin column. Should the value be Null, we would create a vertex for that. Then graph analysis like connected components or community detection would result in worthless output as most prefixes, maintainers, and organisations would belong to the same component or community. We contacted the authors of `rir-data.org` to find out if this was by design or a bug. Unfortunately, this was a limitation of the RIR WHOIS databases themselves. Consequently, we opted to add origins by retrieving them from RIB files. This comes with the caveat that origins are largely retrieved from an operational source rather than an administrative source.

In earlier iterations of the system, we generated IDs using `JOIN` operations. We used to `SELECT` all values and using SQL `DISTINCT` find all unique values and computed IDs using a range of functions and subsequently join the IDs back to the original `DataFrame`. We tried `uuid`, `monotonically_increasing_id`, and other manners of generating unique IDs.

This worked fine until graph sizes of more than 260 thousand records. Beyond this point, the graph would remove existing edges, describe incorrect edges, or worse, construct edges that do not exist in the dataset.

After realizing the problem originated from the `generating_ids` functions, we rewrote the function to take the value of a prefix, maintainer, origin, or organisation, transform it into a `sha256` hash, and prefix it with the vertex type. For example, '*130.89.0.0/16*' becomes '*p_ e127ce3a558d2633367db4e1f77a7fe25257e50df1b0ad2417445c57a8bf3352*'. Once the function was rewritten to perform the operation without needing a `JOIN` operation, this resulted in a significant speed-up. Graphs could now be created at 1/4th of the time required compared to the old method. Using this knowledge of `JOIN` operations being this expensive, we reviewed all queries and looked into ways to perform them without `JOIN`s. Nearly all queries using `JOIN`s before could be rewritten in some way to perform `JOIN`-less with a substantial performance increase.

7.3 Considerations

We currently retrieve origins from an external source, as `rir-data.org` stores a limited number of '*origins*' in their data. The majority of origin data now comes from RIB files from Route Views [66], which is an operational source rather than an administrative source like the RIRs. This could impact research that looks at discrepancies between operational and administrative data. To enable researchers to recognize that data is coming from an operational source, we label all external origins, which can be found in the '*external origin*' label in the graph's vertices.

We realized that the dataset was not as reliable. When parsing the data for `2024-05-23`, we ran into weird errors in our `enrich_df` method or the `Add origins` step in 5.2. The '*origin*' column was missing entirely on this date. We made the method resistant to the missing column by checking if the origin is present. Depending on the outcome, it will set '*external origin*' to `True` for non-`Null` values in '*origin*' or set the value to `True` for all. While this is a minor inconvenience, it does point out reliance on a dataset that is not as reliable as we would hope.

Another limitation of `rir-data.org` is that it only goes back to `2022-11-01`. This project could not produce graphs before that date. There is also a single day missing in `2023-07-26`. It is unclear why this date is unavailable. When you attempt to load the dataset for that day, the system recognizes this date and will load the previous day. The dataset always remains behind for two to three days. When you try to load this date, the system will warn you that this date is not available. This is likely a processing delay.

The current version of the ingester drops the '*country*,' '*status*', and '*source*' values. However, this value could be retained and added to the graphs with relatively minor modifications to the ingester and the graph creator. The addition was considered out of scope for this thesis as it would significantly increase the data stored. This is a trade-off in storage functionality.

As a possible extension, it could allow for country or source (being the allocating RIR) based queries and other interesting research possibilities. Before this addition is implemented, the impact on graph analysis queries must be considered. As creating a significant amount of edges to a limited number of new nodes will create an unbalance in queries such as PageRank. Queries such as (Strongly) Connected Components and Community detection will likely lose their semantic meaning if '*country*' is implemented as a vertex. It probably makes more sense to store it as a label in vertices themselves, as we do with '*external_ origin*'.

Certain queries would perform better when run on a RDBMS, as presented in section 6.3. This design study focused on making the data available in graph form for intriguing graph analysis. However, certain parts of exploratory research can be performed just as well or better on a traditional relational database. We discussed an idea internally for a future project to build a layer on top of this graph system to redirect queries to the back end that could best serve that purpose, as discussed in the Future Work section.

7.4 Recommendations and Future work

We recommend researchers to use our system to research and see where the project can improve. We have already considered several improvements in the future work section. But should a use case not be covered, we encourage contacting us to see if this feature can be integrated.

We also suggest `rir-data.org` to investigate ways to incorporate origins in their data set. For example, the ROUTE objects can be used to find links between allocated prefixes and Autonomous System Numbers, because this will enable us to depend less on operational data when incorporated in their system.

The initial release of software frequently involves balancing functionality against the delivery timeline, and our work is no different. In the subsequent sections, we discuss possible extensions of the project and other future work.

We received several suggestions that we did not implement due to time or resource constraints. Some of these queries will require comparing multiple data dates, which is very resource-intensive. Additionally, Multiple researchers indicated an interest in RPKI data relating to the RIR WHOIS data when contacting them for potential queries. We opted not to incorporate this in the initial version, as this would expand the scope too much. But this could be a worthwhile addition for future research. These queries are in Appendix C.

Currently, comparing dates is restricted to reviewing which edges have changed. This does allow insight into which maintainer, origin, prefix, or organisation has changed the most compared to the previous date. However, it would also be very interesting to see changes in components, pagerank scores, or communities. We encourage researchers to implement these extended comparisons that we had to consider out of scope due to time constraints.

Our system compares the current day against the previous day when parsing RIR data. When a day of data is identical to the previous day, it creates a symlink rather than storing the data, thereby saving storage space. In contrast, when changes in the data are detected, our system stores the entire day of data. The storage mechanism could be improved by storing only changes and modifying the graph creator to compute the graph based on a weekly, monthly, or even yearly snapshot. This is followed by processing the difference between the added, removed, or modified records. As a result, the system would save on storage with a trade-off of requiring more time to compute the graph.

Community Detection is the algorithm in our library with the longest runtime. We encourage researchers to look into faster community detection algorithms and write an implementation for the `GraphFrames` or `GraphX` library. Algorithms like COPRA [67] could be interesting, as this algorithm can deal with overlapping communities.

Graph databases have distinct benefits over relational databases in certain aspects. One of the primary advantages of graph databases is their efficiency in traversing relationships, performing path-finding, and managing recursive relationships. These types of queries are inherently graph-centric, as they require navigating through vertices and edges, which graph databases are optimized for.

In contrast, Relational Database Management System (RDBMS) better handle large-scale aggregation and complex join. Queries that involve summarizing large datasets, such as computing averages, totals, or other aggregate metrics across numerous records, are better suited to the structured tabular format of an RDBMS. The indexing and optimization techniques available in RDBMSes are tailor-made for these kinds of operations, making them perform better for such tasks.

Given these strengths, a best-case solution would leverage both types of databases, depending on the nature of the query. We envision a query management layer on top of the two databases, capable of recognizing which incoming query can be best served by which database and redirecting it accordingly. This layer would review the query's required calls and decide whether the graph database or the RDBMS would provide the best performance, harnessing the strengths of both systems without experiencing their individual shortcomings. This system would also allow for a singular point of access, rather than require the query to be requested to both databases.

Our work lays a solid foundation for the graph database side of this implementation. We have focused on optimizing graph-related queries and identifying which queries are likely better served by an RDBMS. The next step would be building a RDBMS database to deal with the daily volume of data. This will involve designing a schema, deciding indexing strategies, and optimizing data storage to efficiently handle the expected data volumes.

Once the RDBMS implementation is complete, the final step would be to develop an integrated system that seamlessly interfaces with both the graph and relational databases. This could be implemented using `GraphQL` or another layer that can handle multiple backends. This system would include an intelligent query management layer capable of routing queries based on their nature and requirements. Additionally, it provides a unified interface for researchers, lowering the barriers of working with RIR data.

By completing this integration, future researchers can create a powerful, flexible RIR WHOIS data management solution that leverages the best aspects of both database types. This hybrid approach will improve performance for specific queries and provide a more expanded toolset for researchers to explore the RIR data.

Chapter 8

Conclusion

In conclusion, our research extends the pioneering work of the HEAP project, demonstrating the potential of graph databases in network analysis while addressing key performance and maintenance challenges.

In this study, we have explored the utilization of Regional Internet Registries (RIR) data, acknowledging its characteristics and challenges associated with its use. RIRs are organisations that manage the allocation and registration of internet number resources within specific regions of the world. There are five RIRs: RIPE NCC, ARIN, APNIC, LACNIC and AFRINIC.

RIR WHOIS data poses significant challenges for data analysis due to its complexity and differences in structure across different RIRs. Each RIRs data format can vary, creating obstacles for unified data analysis. Our project aims to mitigate these challenges by leveraging the already available data source `rir-data.org`, enriching it, and transforming it into graph structures that facilitate exploratory research. By doing so, we provide a more coherent and unified framework for analysing this data, enabling researchers to uncover insights that might be obscured by the raw, unstructured data.

Each RIR maintains a WHOIS database and delegation and rDNS zone fragment files. These files are important to network management and research. We proposed a graph database constructed from parsed RIR WHOIS data and supported by well-documented `Python` modules for data ingestion, graph creation, and analysis. This graph database structure aims to simplify the complexity of RIR data, making it more accessible to researchers.

Several lessons were learned throughout this project. First, data reliability, in our data source did not reliably provide us with origins. We mitigated this using raw BGP RIB files. We learnt to make our system resilient against missing values in the dataset.

Second, Performance impact of `JOIN`s, when developing the graph builder, we encountered bugs in edge creation. We were able to trace this back to the ID generation. During this debugging process, we realized the performance impact of `JOIN` operations. Using this lesson, we rewrote large parts of the query library to perform its queries with less `JOIN`s.

While our project provides a solid foundation, several areas remain for future research. For instance, the incorporation of Resource Public Key Infrastructure (RPKI) data. RPKI provides cryptographic verification of the route announcements in the Internet's routing infrastructure, enhancing the security and reliability of routing. Integrating RPKI data with our graph database would provide a richer dataset.

Additionally, a promising direction for future work involves running an Relational Database Management System (RDBMS) in parallel with our graph database, with a unified access layer connecting both systems. This approach would allow us to redirect

incoming queries to the database that best serve the researcher's need, combining the strengths of both database types. For instance, the graph database could handle complex relationship queries, while large-scale aggregations and complex joins could be managed by the RDBMS.

In conclusion, our proposed graph database model presents a viable and suitable solution for analysing RIR WHOIS data. This work not only fulfils the current research and design goals but also lays the groundwork for ongoing improvements and innovations in the field of RIR data analysis. Integrating RPKI data and developing a hybrid system combining graph databases with RDBMS are promising areas for future research. By continuing to build on this foundation, researchers can gain deeper insights into RIR data and contribute to the security and stability of the global Internet infrastructure.

Chapter 9

Acknowledgements

This work was done in the context of a Master's Thesis for the University of Twente, supervised by Ralph Holz, with daily supervision conducted by Ebrima Jaw. Thanks to SIDN for giving me permission to use the `utwente.nl` WHOIS example. Thomas Krenc, Romain Fontugne, Taejoong (Tijay) Chung, Shyam Krishna Khadka, Moritz Müller contributed several queries for the system to solve. Gustavo Luvizotto contributed with his code and technical know-how for *podmannerizing* the system.

Bibliography

[1] J. Schlamp, R. Holz, Q. Jacquemart, G. Carle, and E. W. Biersack, "HEAP: Reliable assessment of BGP hijacking attacks," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 6, pp. 1849–1861, 2016.

[2] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, pp. 1–39, 2008.

[3] J. Pokornỳ, "Graph databases: Their power and limitations," in *Computer Information Systems and Industrial Management: 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015, Proceedings 14*, Springer, 2015, pp. 58–69.

[4] M. Besta, R. Gerstenberger, E. Peter, *et al.*, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–40, 2023.

[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 599–613, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez.

[6] T. Andersson and H. Reinholdsson, *Rest api vs graphql: A literature and experimental study*, 2021.

[7] GraphQL, *GraphQL Foundation*, https://graphql.org/faq/foundation/, [Online; accessed 10-June-2024].

[8] S. Buna, *GraphQL in Action*. Simon and Schuster, 2021.

[9] B. Du, K. Izhikevich, S. Rao, *et al.*, "IRRegularities in the Internet Routing Registry," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, 2023, pp. 104–110.

[10] A. Arouna, I. Livadariu, and M. Jonker, "Lowering the Barriers to Working with Public RIR-Level Data," in *Proceedings of the Applied Networking Research Workshop*, 2023, pp. 24–26.

[11] NRO, *About RIRs*, https://www.nro.net/about/rirs/, [Online; accessed 11-June-2024].

[12] ICANN, *ASO & NRO*, https://aso.icann.org/about/aso-and-nro/, [Online; accessed 22-March-2024].

[13] LACNIC, *About LACNIC*, https://www.lacnic.net/1004/2/lacnic/about-lacnic, Online; accessed 11-June-2024.

[14] AFRINIC, *AFRINIC - SPEARHEADING AFRICA'S INTERNET SINCE 2005*, https://afrinic.net/ast/pdf/afrinic-10years-ab-sept-2015.pdf, Online; accessed 11-June-2024.

[15] C. L. Evans, *Broad band: The untold story of the women who made the internet*. Penguin, 2020.

[16] L. Daigle, *WHOIS Protocol Specification*, RFC 3912, Sep. 2004. DOI: 10.17487/RFC3912. [Online]. Available: https://www.rfc-editor.org/info/rfc3912.

[17] K. Elliott, "The who, what, where, when, and why of whois: Privacy and accuracy concerns of the whois database," *SMU Sci. & Tech. L. Rev.*, vol. 12, p. 141, 2008.

[18] A. M. Costello, *Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)*, RFC 3492, Mar. 2003. DOI: 10.17487/RFC3492. [Online]. Available: https://www.rfc-editor.org/info/rfc3492.

[19] ICANN, *2023 Global Amendments to the Base gTLD Registry Agreement (RA), Specification 13, and 2013 Registrar Accreditation Agreement (RAA)*, https://www.icann.org/resources/pages/global-amendment-2023-en, [Online; accessed 23-November-2023], 2023.

[20] ICANN GNSO, *PDP Thick WHOIS*, https://gnso.icann.org/en/group-activities/active/thick-whois, Online; accessed 11-June-2024.

[21] ICANN, *Thick WHOIS*, https://www.icann.org/resources/pages/thick-whois-2016-06-27-en, [Online; accessed 16-March-2024], May 2019.

[22] NRO, *New Format for Internet Resource Statistics*, https://www.nro.net/new-format-for-internet-resource-statistics/, Online; accessed 11-June-2024.

[23] ARIN, *Extended Allocation and Assignment Report for RIRs*, https://www.arin.net/reference/research/statistics/nro_extended_stats_format.pdf, [Online; accessed 23-November-2023], 2023.

[24] RIPE NCC, *RIR Statistics Exchange Format*, https://ftp.ripe.net/pub/stats/ripencc/RIR-Statistics-Exchange-Format.txt, [Online; accessed 11-November-2023], 2023.

[25] RIPE NCC, *IPv6 Address Allocation and Assignment Policy*, https://www.ripe.net/publications/docs/ripe-552#definitions, [Online; accessed 23-November-2023], 2023.

[26] D. A. Bader, J. Feo, J. Gilbert, *et al.*, "HPC Scalable Graph Analysis Benchmark," *HPC Graph Analysis*, vol. 2009, pp. 1–10, 2009.

[27] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, SIAM, 2004, pp. 442–446.

[28] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vanó, S. Gómez-Villamor, N. Martínez-Bazan, and J. L. Larriba-Pey, "Survey of graph database performance on the hpc scalable graph analysis benchmark," in *International Conference on Web-Age Information Management*, Springer, 2010, pp. 37–48.

[29] M. Ciglan, A. Averbuch, and L. Hluchy, "Benchmarking traversal operations over graph databases," in *2012 IEEE 28th International Conference on Data Engineering Workshops*, IEEE, 2012, pp. 186–189.

[30] S. Jouili and V. Vansteenberghe, "An empirical comparison of graph databases," in *2013 International Conference on Social Computing*, IEEE, 2013, pp. 708–715.

[31] Tinkerpop Apache, *Tinkerpop GitHub sunset notice*, https://github.com/tinkerpop/blueprints/blob/master/README.textile, [Online; accessed 18-December-2023], 2023.

[32] J. Fan, A. G. S. Raj, and J. M. Patel, "The case against specialized graph analytics engines.," in *Proceedings of the Conference on Innovative Data Systems Research*, 2015.

[33] D. ten Wolde, T. Singh, G. Szárnyas, and P. Boncz, "DuckPGQ: Efficient property graph queries in an analytical rdbms," in *Proceedings of the Conference on Innovative Data Systems Research*, 2023.

[34] Y. Tian, E. L. Xu, W. Zhao, *et al.*, "IBM db2 graph: Supporting synergistic and retrofittable graph queries inside IBM db2," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 345–359.

[35] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 4, pp. 1–44, 2017.

[36] A. Jindal, S. Madden, M. Castellanos, and M. Hsu, "Graph analytics using vertica relational database," in *2015 IEEE International Conference on Big Data (Big Data)*, IEEE, 2015, pp. 1191–1200.

[37] A. Pacaci, A. Zhou, J. Lin, and M. T. Özsu, "Do we need specialized graph databases? benchmarking real-time social networking applications," in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, 2017, pp. 1–7.

[38] A. A. Ali and D. Logofatu, "An analysis on graph-processing frameworks: Neo4J and Spark GraphX," in *IFIP International Conference on Artificial Intelligence Applications and Innovations*, Springer, 2022, pp. 461–470.

[39] I. Ballas, V. Tsakanikas, E. Pefanis, and V. Tampakas, "Assessing the computational limits of graphdbs' engines-a comparison study between Neo4J and Apache Spark," in *Proceedings of the 24th Pan-Hellenic Conference on Informatics*, 2020, pp. 428–433.

[40] G. Kalogeras, V. Tsakanikas, I. Ballas, V. Aggelopoulos, and V. Tampakas, "Community detection at scale: A comparison study among Apache Spark and Neo4J," in *Proceedings of the 26th Pan-Hellenic Conference on Informatics*, 2022, pp. 21–26.

[41] D. team, *Dblp faq*, https://dblp.uni-trier.de/faq/index.html, [Online; accessed 8-April-2024].

[42] G. Brito, T. Mombach, and M. T. Valente, "Migrating to GraphQL: A practical assessment," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2019, pp. 140–150.

[43] M. Vogel, S. Weber, and C. Zirpins, "Experiences on migrating RESTful web services to GraphQL," in *Service-Oriented Computing–ICSOC 2017 Workshops: ASOCA, ISyCC, WESOACS, and Satellite Events, Málaga, Spain, November 13–16, 2017, Revised Selected Papers*, Springer, 2018, pp. 283–295.

[44] A. Quiña-Mera, P. Fernandez, J. M. García, and A. Ruiz-Cortés, "Graphql: A systematic mapping study," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1–35, 2023.

[45] R. Beverly, "IP Neo-colonialism: Geo-auditing RIR Address Registrations," *arXiv preprint arXiv:2308.12436*, 2023.

[46] E. N. Nemmi, F. Sassi, M. La Morgia, C. Testart, A. Mei, and A. Dainotti, "The parallel lives of autonomous systems: Asn allocations vs. bgp," in *Proceedings of the 21st ACM Internet Measurement Conference*, 2021, pp. 593–611.

[47] F. Streibelt, M. Lindorfer, S. Gürses, C. H. Gañán, and T. Fiebig, "Back-to-the-Future Whois: An IP Address Attribution Service for Working with Historic Datasets," in *International Conference on Passive and Active Network Measurement*, Springer, 2023, pp. 209–226.

[48] A. Arouna, I. Livadariu, and M. Jonker, *Lowering the Barriers to Working with Public RIR-Level Data*, https://rir-data.org/, [Online; accessed 30-October-2023], 2023.

[49] X. Cai, J. Heidemann, B. Krishnamurthy, and W. Willinger, "Towards an AS-to-organization Map," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pp. 199–205.

[50] R. Fontugne, *Internet yellow pages*, https://github.com/InternetHealthReport/internet-yellow-pages, [Online; accessed 9-April-2024], 2023.

[51] R. Fontugne, *Internet yellow pages*, https://iyp.iijlab.net/, [Online; accessed 9-April-2024], 2023.

[52] J. Schlamp, R. Holz, O. Gasser, *et al.*, "Investigating the nature of routing anomalies: Closing in on subprefix hijacking attacks," in *Traffic Monitoring and Analysis: 7th International Workshop, TMA 2015, Barcelona, Spain, April 21-24, 2015. Proceedings 7*, Springer, 2015, pp. 173–187.

[53] Chris Ré, *EmptyHeaded github*, https://github.com/HazyResearch/EmptyHeaded, [Online; accessed 30-April-2024].

[54] J. Fan, A. G. S. Raj, S. Suresh, and J. M. Patel, *GRAIL github*, https://github.com/UWQuickstep/Grail, [Online; accessed 30-April-2024].

[55] Vertica, *Vertica Trial Page*, https://www.vertica.com/try/, [Online; accessed 30-April-2024].

[56] R. Hat, *Podman: A tool for managing OCI containers and pods*, https://podman.io/, [Online; accessed 30-April-2024].

[57] Docker Hub, *Jupyter/pyspark-notebook: Official Docker image for Jupyter Notebook with Apache Spark support*, https://hub.docker.com/r/jupyter/pyspark-notebook, [Online; accessed 30-April-2024].

[58] H. Asghari, *Pyasn*, https://github.com/hadiasghari/pyasn, [Online; accessed 29-April-2024].

[59] GraphFrames, *Graphframes*, https://github.com/graphframes/graphframes, [Online; accessed 29-April-2024].

[60] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[61] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E*, vol. 76, no. 3, p. 036 106, 2007.

[62] Apache SPARK, *pyspark.sql.DataFrame.sample*, https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.sql.DataFrame.sample.html, Online; accessed 11-June-2024.

[63] Python, *Python Random*, https://docs.python.org/3/library/random.html, Online; accessed 11-June-2024.

[64] OVH, *VPS IP*, https://www.ovhcloud.com/en/vps/vps-ip/, Online; accessed 30-June-2024.

[65] E. Jaw, M. Müller, C. Hesselman, and L. Nieuwenhuis, "Serial bgp hijackers: A reproducibility study and assessment of current dynamics," in *2024 8th Network Traffic Measurement and Analysis Conference (TMA)*, IEEE, 2024, pp. 1–10.

[66] U. of Oregon RouteViews Project, *RouteViews*, https://www.routeviews.org/, [Online; accessed 29-April-2024].

[67] S. Gregory, "An algorithm to find overlapping community structure in networks," in *European conference on principles of data mining and knowledge discovery*, Springer, 2007, pp. 91–102.

Acronyms

AFRINIC African Network Information Centre. 7, 14, 38

API Application Programming Interface. 2, 5, 6, 13

APNIC Asia Pacific Network Information Centre. 7, 8, 14, 15, 38

ARIN American Registry for Internet Numbers. 7, 14, 22, 38

ARPANET Advanced Research Projects Agency Network. 7

AS Autonomous System. 1, 2, 8, 14, 15, 29, 30, 32

ASCII American Standard Code for Information Interchange. 7, 8

ASN Autonomous System Number. 1, 7, 9, 15, 20, 32, 36

ASO Address Supporting organisation. 7

BFS Breadth-first Search. 12, 23, 24, 27

BGP Border Gateway Protocol. 1, 2, 14, 15, 33, 38

CAIDA Center for Applied Internet Data Analysis. 14, 17

ccTLD Country Code top-level domain. 8

CPU Central Processing Unit. 16, 19, 26

DBLP Digital Bibliography and Library Project. 13

DNS Domain Name System. 9, 14, 15

DNSSEC Domain Name System Security Extensions. 8

FTP File Transfer Protocol. 10

HPC-SGAB High Performance Computing Scalable Graph Analysis Benchmark. 11

IANA Internet Assigned Numbers Authority. 15

ICANN Internet Corporation for Assigned Names and Numbers. 7, 8

IHR Internet Health Report. 17

IIJ Internet Initiative Japan. 17

IP Internet Protocol. 1, 2, 4, 7, 9, 10, 30, 33

IPv4 Internet Protocol version 4. 1, 7, 8, 9, 10, 21

IPv6 Internet Protocol version 6. 7, 8, 9, 10

IRR Internet Routing Registry. 7, 15

ISO International organisation for Standardization. 9

ISP Internet Service Provider. 1, 4, 7, 9, 29, 30, 32, 33

IYP Internet Yellow Pages. 15, 17

LACNIC Latin America and Caribbean Network Information Centre. 7, 14, 38

LIR Local Internet Registries. 7, 9

LPA Label Propagation Algorithm. 13, 25, 28, 51

LPG Labeled Property Graph. 5, 11

MOAS Multiple Origin Autonomous System. 15

MoU Memorandum of Understanding. 7

NIR National Internet Registries. 7

NoSQL Not only Structured Query Language. 4, 5, 12

NRO Number Resource organisation. 7

PTR Pointer. 10

RAM Random Access Memory. 16, 19, 26

RDBMS Relational Database Management System. 4, 12, 20, 26, 27, 28, 36, 37, 38, 39

RDF Resource Description Framework. 5, 11, 12

rDNS Reverse Domain Name System. 1, 2, 7, 9, 38

REST Representational State Transfer. 6, 13

RFC Request For Comments. 7

RIB Routing Information Base. 1, 22, 26, 34, 35, 38

RIPE NCC Réseaux IP Européens Network Coordination Centre. 7, 14, 15, 38

RIR Regional Internet Registries. 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 20, 21, 26, 33, 34, 35, 36, 37, 38, 39

ROA Route Origin Authorizations. 15

RPKI Resource Public Key Infrastructure. 17, 20, 26, 32, 33, 36, 38, 39, 52

SIDN Stichting Internet Domeinregistratie Nederland (English: Foundation for Internet Domain Registration Netherlands). 17, 40

SNAP Stanford's Network Analysis Platform. 13

SQL Structured Query Language. 12, 19, 21, 26, 34

TEPS Traversed Edges per Second. 11

TLD Top-Level Domain. 8

TTL Time To Live. 10

URL Uniform Resource Locator. 53

UTC Coordinated Universal Time. 8

Appendix A

Raw WHOIS data example

Listing A.1: WHOIS output for utwente.nl

```
Domain name: utwente.nl
Status:      active

Registrar:
   Universiteit Twente
   Drienerlolaan 5
   7522NB Enschede
   Netherlands

Abuse Contact:
   +31.534891313
   email@utwente.nl

Creation Date: 1986-10-16

Updated Date: 2020-11-03

DNSSEC:      yes

Domain nameservers:
   ns3.utwente.nl          131.155.0.37
   ns1.utwente.nl          130.89.1.2
   ns1.utwente.nl          2001:67c:2564:a102::3:1
   ns2.utwente.nl          130.89.1.3
   ns2.utwente.nl          2001:67c:2564:a102::3:2

Record maintained by: SIDN BV

Copyright notice
No part of this publication may be reproduced, published, stored in a
retrieval system, or transmitted, in any form or by any means,
electronic, mechanical, recording, or otherwise, without prior
permission of SIDN.
These restrictions apply equally to registrars, except in that
```

Appendix B

Queries

Queries implemented in this project:

1. List all vertices of type x linked with a path to y.
   This query shows all prefixes managed by an organisation, for instance.

2. Is there a link between Vertex x and Vertex y
   These vertices can be the same time, but this is not required. The result of this query is the shortest path from x to y.

3. Find the highest or lowest degree vertex.
   This can be in-degree, out-degree, or both.

4. Apply the PageRank algorithm to a graph.
   This shows the most interconnected nodes of the graph.

5. Find Connected Components
   This query runs Connected Components algorithm over the entire graph, finding prefixes, maintainers, origins, and organisations connected to each other.

6. Find Strongly Connected Components
   This query runs the Strongly Connected Components algorithm over the entire graph, finding prefixes, maintainers, and origins connected to each other.

7. Apply Community Detection
   This query runs LPA on the graph to find communities of connected prefixes, maintainers, origins, and organisations.

8. Show all changes between date x and date y
   This shows all changes in the edges between prefixes, maintainers, origins, and organisations between two dates.

Appendix C

Future work queries

Queries not implemented in this project but relevant enough to warrant future work:

1. Show the difference between object x at different points in time
   This query displays the attributes that changed at different points in time.

2. Query change of ownership.
   This query returns the previous maintainer/organisation that owned the prefix, if available. Using the data in the *'last-modified'* we can load the graph of a date before that day to find the previous owner. This comes with the limitation that the `rir-data.org` dataset only goes back to `2022-11-01`.

3. Queries related to RPKI
   While these queries were much requested, these were considered out of scope as RPKI data is not included in the `rir-date.org` dataset.

Appendix D

Manual

D.1 Prerequisites

We assume you try to run it on a Linux based server. We tested it on `Debian 11 (bullseye)`. It will most likely work on other distributions, but we have not confirmed that it works. We used `podman 3.0.1` and `podman-compose 0.1.11`.

D.2 Container

We provide a `Dockerfile` to build the container. This starts the `PySpark Jupyter notebook`. Along some path and port configuration in `.env` and the `docker-compose.yml`. We provide a convenient script called `start-notebook.sh` to kick off the compose and pipe all output to a `session` file. This file will contain the token to access the `Jupyter notebook`. Once the container is up and running, you can access the server from the web browser using the URL or you can use an editor like `VS Code`. In the 'notebooks' folder, we provide a couple of notebooks. The most important ones are `spark-instance-rirmap.ipynb`, this contains all configuration necessary to run `spark` code. You can modify the core count and memory configuration to fit your machine. Be aware that lowering resource allocation might impact your performance negatively. The other two relevant notebooks are `ingest_and_store.ipynb` and `make_graph.ipynb`. These will be discussed in the following sections.

D.3 Ingesting

In the `ingest_and_store.ipynb` we show an example of how to ingest `rir-data.org` data. This uses the `ingester.py` module. If you make any changes to the directory structure where you store the dataset, this must be reflected here too. The main method that will be of interest to you is `process_rir_data_dot_org()`. This method requires a `SparkSession` which can be generated using the `spark-instance-rirmap.ipynb` notebook. You can choose to leave the `current_data` and `latest_date` field empty if you wish to ingest all data. By default, `current_data` looks for the first date that is not present in the dataset folder. `latest_data` looks at the `rir-data.org` dataset to find out the latest date that is present.

D.4 Graph Creation and Analysis

In the `make_graph.ipynb` we show an example of how to generate a graph as well as some analysis examples. This notebook uses the `rirmap.py` module. If you make any changes to the directory structure where you store the dataset, this must be reflected here too. To create a graph, you want to use `load_graph()`. This requires a `datetime` and a `SparkSession`. Alternatively, if you wish to perform analysis on the source `DataFrame` as well as the graph, you can load the source `DataFrame` using `load_df()` and `load_graph_df()` to get both objects separately.

Once you have the `GraphFrame` object, you can perform analysis using the methods provided in the `rirmap.py` module. Have a look at the *pydoc* or the Evaluation chapter 6.3 for inspiration.

Appendix E

Benchmark data

E.1 Traversal

These numbers were collected by running `benchmark_traversal()` method present in the `checker` module. It gathers five random vertices from each type (prefix, origin, maintainer, and organisation). For each of these values, we time a run of each traversal. We repeat this process five times. The runtimes for each of these runs are shown in the table E.2.

E.2 Link Vertices

These numbers were collected by running `benchmark_link_vertices()` method present in the `checker` module. It requires `connected_components()` to be run beforehand. It aggregates the vertices on component and sorts the count. It picks 4 components based on the count, it picks the largest, the smallest and 2 random components. The method then picks 3 vertex pairs in each component and computes the path between them at 3 different *maxPathLength*s: $5, 10$ and 15. The runtimes for each of these runs are shown in the table E.2.

TABLE E.1: Runtime per vertex type in seconds

	Type	Prefix					Origin					Maintainer					Organisation				
Run	step type	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	1	1.59	1.07	0.88	0.86	0.92	0.89	1.17	0.94	1.12	1.70	2.38	0.98	1.62	0.88	1.56	257.36	91.36	62.95	5.62	146.57
	2	138.64	0.90	1.03	52.6	7.78	3.26	4.56	3.22	3.66	2.80	0.51	0.47	0.47	0.46	0.48	0.88	0.88	1.20	0.85	1.02
	3	1.08	3.91	9.58	0.95	0.92	2.25	2.13	2.01	1.83	1.83	2.30	1.90	1.90	2.00	1.71	527.74	175.59	95.69	11.70	316.56
2	1	0.96	1.08	1.25	1.05	0.93	0.98	0.96	1.67	1.02	1.22	7.84	1.27	1.02	1.00	1.40	3.17	146.38	246.55	2.30	146.47
	2	51.60	4.52	8.02	3.38	130.56	2.93	3.41	4.19	5.31	2.75	0.51	1.22	0.79	0.85	0.59	2.64	0.91	1.01	0.93	0.95
	3	0.93	0.89	0.83	1.55	0.94	1.93	1.79	2.31	1.89	2.50	13.31	2.42	2.14	1.88	3.41	4.37	287.79	486.56	5.55	289.20
3	1	0.94	1.27	1.32	1.11	1.00	0.88	0.93	1.49	0.91	2.89	9.06	0.92	0.97	1.00	0.98	251.92	6.08	12.98	244.61	147.49
	2	3.16	7.92	3.76	127.70	6.07	3.06	3.34	3.70	2.80	2.85	0.56	0.53	0.50	0.67	0.52	0.96	0.93	1.29	1.01	1.16
	3	0.94	0.89	0.87	0.91	1.46	2.02	2.18	2.01	1.98	3.84	13.33	2.63	4.01	2.84	3.22	517.22	12.15	25.95	484.85	295.04
4	1	1.11	0.90	0.93	0.93	0.95	0.98	0.98	1.86	0.94	1.43	1.10	0.93	1.04	0.92	0.93	8.52	86.12	143.79	101.57	2.09
	2	49.26	35.85	97.80	2.81	126.01	3.38	4.66	3.21	3.20	3.60	1.01	0.85	0.75	0.52	0.51	2.53	0.99	1.12	0.99	0.93
	3	1.01	0.98	0.95	0.98	0.97	2.48	2.00	3.14	1.97	4.42	2.38	2.07	2.12	3.32	2.872	17.77	171.18	283.83	201.24	3.62
5	1	1.04	1.12	1.07	1.72	1.00	1.98	2.12	2.11	2.13	1.00	1.09	1.90	1.10	1.51	0.98	153.55	2.23	34.47	141.65	2.69
	2	148.15	11.34	3.19	4.19	236.33	5.19	4.82	4.23	3.56	4.45	1.88	0.62	0.81	0.64	0.66	1.02	1.68	1.07	1.16	1.09
	3	1.11	2.02	1.01	1.76	1.02	4.01	2.36	2.81	2.30	2.99	2.45	3.60	2.31	3.66	3.28	323.65	3.72	67.31	274.60	5.55

TABLE E.2: Runtime per path computation in seconds

	Component	1			2			3			4		
Run	MaxPathLength\Pair	1	2	3	1	2	3	1	2	3	1	2	3
1	5	621.47	271.62	1657.32	422.93	403.13	437.72	2514.87	3147.82	4028.26	361.29	325.15	504.11
	10	717.89	254.97	1679.20	408.00	423.37	459.59	2767.12	3310.62	4057.29	408.16	350.58	547.54
	15	685.62	270.07	1896.69	401.80	422.17	470.95	2908.27	3769.79	4170.53	405.54	367.68	544.85
2	5	1376.79	1431.00	1548.54	1617.67	1745.45	1795.50	7088.35	11315.47	2178.60	406.67	422.81	585.16
	10	1413.13	1501.70	1590.31	1645.10	1760.75	1598.15	7549.80	11324.50	2188.60	434.88	438.89	632.41
	15	1478.47	1532.27	1585.43	1685.35	1783.94	1607.63	7907.14	8860.86	2220.58	457.85	429.02	664.54
3	5	1075.20	1125.06	1155.51	5265.52	1343.77	1357.36	6315.67	6425.48	3362.27	481.62	473.22	690.97
	10	1093.52	1123.81	1601.55	5272.57	1351.31	1385.86	6133.41	6512.03	3358.19	502.53	469.99	691.70
	15	1084.31	1126.09	1917.66	5420.78	2721.97	1413.88	6201.93	6830.95	3392.94	527.10	597.84	682.09