FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Analysis of Alpine Linux packages using a graph database
Student:	Jakub Meinlschmidt
Supervisor:	Ing. Jakub Jirůtka
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of this thesis is to provide package maintainers and users of Alpine Linux with a tool to query information about packages (their metadata, relationships…) and to explore the feasibility and benefits of using a graph database for this purpose.

1. Analyse Alpine Linux package repositories and package formats. Focus on the contained metadata and relationships.
2. Research the state of the art in open source graph databases, graph modelling and the use of graph databases in package distribution. Select the most appropriate database for this task.
3. Design a graph data model to effectively store the metadata and relationships of packages in the Alpine Linux repositories in the chosen graph database. Consider the intended use of the database.
4. Design and develop a proof-of-concept implementation of the tool to collect and import said data into the database, and to periodically update the database with new data.
5. Demonstrate the capabilities of the implemented data model and database query language on a set of queries inspired by the practical needs of package maintainers and users. Compare with existing tools available in Alpine Linux.
6. Discuss the strengths, weaknesses and opportunities of graph databases for this domain, especially in comparison with relational databases and SQL. Discuss the benefits of this work for the Alpine Linux community.

Electronically approved by Ing. Michal Valenta, Ph.D. on 17 February 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Analysis of Alpine Linux packages using a graph database

Jakub Meinlschmidt

Department of Software Engineering
Supervisor: Ing. Jakub Jirůtka

May 11, 2023

Acknowledgements

I would like to thank my supervisor Ing. Jakub Jirůtka, for his guidance and valuable observations. Thanks for the expert advice belongs to Ing. Štěpán Plachý, Mgr. Vu Tung Anh and Hana Turková, DiS. for spelling corrections. In special thanks, I would like to mention Ing. Jan Bittner for the practical application of the methodology of enhancing the level of cognitive well-being (the so-called outdoor walks). I would also like to thank all the others who supported me on this journey.

Declaration

In Prague on May 11, 2023 ....

Citation of this thesis

Meinlschmidt, Jakub. *Analysis of Alpine Linux packages using a graph database.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023. Also available from: ⟨`https://github.com/jmeinlschmidt/linux-pkgs-in-graph-db`⟩.

Abstract

This thesis aims to provide package maintainers and users of Alpine Linux with a tool to query information about packages and explore the feasibility and benefits of using a graph database for this purpose. As part of this work, a *proof-of-concept* tool that represents package metadata and relationships between them in the Neo4j graph database, is successfully implemented, providing new possibilities for their analysis. It also addresses the differences in modelling this data in traditional relational databases versus the chosen NoSQL database. Finally, this thesis provides recommendations on how to make this tool accessible to the Alpine Linux community.

Keywords package management, Alpine Linux, graph databases, NoSQL, Neo4j

Abstrakt

Cílem této práce je poskytnout správcům balíčků a uživatelům systému Alpine Linux nástroj pro vyhledávání informací o balíččích a prozkoumat možnosti a výhody použití grafové databáze pro tento účel. V rámci této práce je úspěšně implementován *proof-of-concept* nástroje, který reprezentuje metadata balíčků a vztahy mezi nimi v grafové databázi Neo4j, a poskytuje tak nové možnosti pro jejich analýzu. Věnuje se také rozdílům v modelování těchto dat v tradičních relačních databázích oproti zvolené NoSQL databázi. Na závěr tato práce poskytuje doporučení, jak tento nástroj zpřístupnit komunitě Alpine Linux.

Klíčová slova správa balíčků, Alpine Linux, grafové databáze, NoSQL, Neo4j

Contents

List of Figures

List of Tables

List of Source Codes

Introduction

Alpine Linux, known for its lightweight nature and security-focused design, has gained significant popularity in the realm of Linux distributions. With its minimalist approach, Alpine Linux has become a preferred choice for various use cases, ranging from embedded systems to containerized environments. As Alpine Linux is constantly evolving to meet the diverse user needs, efficient *package management*[1] becomes crucial for maintaining a stable and reliable system.

Package management plays a vital role in the lifecycle of any Linux distribution. The availability of accurate and up-to-date information about packages is essential for both package maintainers and users. Package maintainers need to ensure the integrity and quality of packages, while users rely on this information to make informed decisions regarding the software components they include in their systems.

With the ever-growing complexity of software dependencies and the need for efficient information retrieval, it becomes imperative to have robust tools that facilitate package querying and analysis. This thesis focuses on addressing these challenges within the context of Alpine Linux, aiming to provide package maintainers and users with a tool that enables comprehensive information retrieval and analysis of packages

The existing package management system in Alpine Linux lacks adequate tools to efficiently explore package metadata and their relationships. This limitation prevents users and maintainers from gaining valuable insights into the package ecosystem, potentially leading to inefficiencies in software development processes.

[1]In Linux terminology, a package refers to an archive containing computer software or a library.

To overcome these limitations, this thesis explores the feasibility and benefits of employing graph databases as a solution. Graph databases, built upon graph theory principles, offer a powerful and flexible approach for representing complex relationships between entities. By leveraging the graph databases, the thesis seeks to provide a proof-of-concept tool that represents package metadata and their relationships in a structured manner. It is important to note that none of the popular Linux distributions currently provide their users with tools based on a graph database. However, in 2018, Preining partially addressed this area in the context of Debian Linux on his blog [1].

This thesis begins by describing the problem of package management in Alpine Linux, emphasizing the need for a more comprehensive and efficient information retrieval system. It then dives into graph theory and its application to convert the package domain into a property graph structure. Various graph database management systems are evaluated based on the requirements and the graph model created, leading to the selection of Neo4j as the most suitable solution for this purpose.

Building upon the chosen graph database, a proof-of-concept tool is developed, thereby allowing advanced querying and analysis capabilities. The performance and simplicity advantages of the graph database approach over traditional relational databases are demonstrated, providing a compelling case for its adoption in the Alpine Linux ecosystem.

Overall, this thesis aims to bridge the gap in package querying and analysis within Alpine Linux by harnessing the power of graph databases, offering package maintainers and users a valuable tool to enhance their understanding and management of the package ecosystem.

Chapter 1

State of the Art

The purpose of this chapter is to provide a comprehensive theoretical understanding of package management, Alpine Linux, and the operational mechanisms of package management within Alpine Linux, along with the available solutions. Besides, this chapter introduces graph theory, including relevant terminology and concepts. Finally, the chapter concludes by offering an overview of databases, with a particular focus on graph databases.

It also outlines the area of relational databases but it is not the primary purpose of this chapter. A closer connection between these areas is explored in Chapter 2.

1.1 Package Management

1.1.1 Motivation

One established way for users to install software or libraries in open-source Linux distributions is downloading the source code of the desired software and then compiling it into an executable form on their device.

Despite the fact that initially this method may seem universal and functional, it comes with various issues. One of the primary challenges arises when the installed software requires another program or library to function properly. Such required software is called a *dependency*, and it is the user's responsibility to install these dependencies before installing the desired software similarly. Notably, these dependencies can repeat transitively, and different programs may require different versions of the same dependency, leading to dependency conflicts.

Apart from dependency issues, the process of manual compilation can be time-consuming, with some compilations taking not only minutes but also several hours, and their successful completion is not always guaranteed. To simplify the above-described software installation process, a system called *package management* is leveraged.

Package management refers to a set of tools and methods for managing computer programs. [2] This includes their installation and automated dependency resolution required for installation, upgrading, configuring, and removal.

1.1.2 Package managers

Package management is implemented through a tool called a *package manager.* [2] It is possible for a package management system to support multiple package managers. While package managers are now a widespread feature of many operating systems, this thesis focuses on the open-source Linux distribution Alpine Linux. It does not describe proprietary package managers such as the Apple App Store or Steam[2].

Another widely used category of package managers, which is not addressed in this work, is *application-level package managers.* These package managers are typically specific to particular programming languages. Notable examples include npm, Maven, pip, RubyGems, and CTAN[3].

From the perspective of operating systems, two types of package managers can be distinguished [3]:

1. Source code-based,
2. Binary.

The first type, source code-based package managers, functions on a similar principle to that described in Section 1.1.1. The package manager automates dependency resolution and is responsible for compiling the source code on the target device. Compiling on the target device can provide better performance and flexibility by allowing the software to be built with custom compilation options. However, the compilation of packages from source is a complex task with high requirements in terms of time and system resources which is one of the negative aspects that have a negative impact on user friendliness. Notable source code-based package managers are Ports (FreeBSD) or Portage (Gentoo). Some, such as Homebrew (used in macOS) or Nix (NixOS), offer both Source-code and Binary approaches.

However, the most widely used package managers install software from distribution packages in binary form, compiled for the target CPU architecture and packaged on the infrastructure of a particular (Linux) distribution. This method is particularly popular because of its fast installation time, greater stability and predictability. The most notable examples include apk-tools (used in Alpine Linux and OpenWrt), dpkg (Ubuntu), and Pacman (Arch Linux). Software distributed within package managers is referred to as a *package.*

[2]Gaming package manager developed by Valve

[3]Package manager for TeX

1.1.3 Package

A package is an archive file containing computer programs or libraries to be copied to the system and other metadata[4]. It also contains install scripts required for its installation by a package manager. [2] The format of this file typically varies depending on the package manager used.

From the perspective of Linux distribution, packages are its fundamental building blocks. The role of a Linux distribution is integrative and involves orchestrating these packages into a functional whole. Usually, authors of these packages are not the authors of the software itself but instead contributors to the particular Linux distribution. They also play a role of a reviewer. As a result, the Linux distribution takes over some responsibility for the content it distributes via packages. However, this responsibility is typically not legally grounded.

In contrast to BSD, Linux distributions also perform system-wide updates through the package manager. It is because even the Linux kernel itself is distributed as a separate package.

1.1.4 Dependency

Package dependencies are requirements that must be met for a package to function properly. Dependency issues, including problems and conflicts resolution, have already been discussed in the previous section.

Another perspective on the significance of dependencies lies in the complexity of the code base. As the number of dependencies in software increases, so does its complexity. This complexity is accompanied by a higher likelihood of errors and an expanding attack surface. [4]

According to Kernighan [5]: *"Controlling complexity is the essence of computer programming."*.

1.1.5 Repository

A repository is a group of packages linked by a specific attribute. [6] To access these repositories, users need to specify them in the package manager's configuration. Some package managers allow users to access a broader range of packages and versions from various sources. [7] Managing multiple repositories in package management systems can pose a challenge due to conflicting dependencies or package versions.

1.1.6 Mirror

A mirror is a website or a server that hosts a copy of one or more repositories and provides users with faster and more reliable access to packages and updates. [6]

[4]Such as dependencies, name, version, author, etc.

Mirrors can also be used to distribute packages to more users, reducing the load on the primary repository and improving overall system performance. Some mirrors may offer additional features, such as customized packages or localized content, to better serve specific needs.

1.2 Alpine Linux

Alpine Linux is a lightweight, non-commercial, security-oriented distribution known for its small size. Its motto is “Small, Simple, Secure”. [8] Alpine Linux is based on *musl libc* and has a small base userland – in terms of disk size, memory footprint, and number of packages. Making it ideal for running in containers and embedded devices where disk space and memory resources are limited. The name *Alpine* originally stood for “**A** **L**inux **P**owered **I**ntegrated **N**etwork **E**ngine”. [9] It was created by the Alpine Linux Development Team in 2005. Initially, it was supposed to be an embedded-first distribution. [10]

However, during that time, it gained popularity as a distribution for container images in services such as Docker due to its small size. The container image size is approximately 8 MiB, while a minimal installation on a virtual machine or a bare metal requires roughly 130 MiB of disk space. [8] Having a minimal amount of packages creates a small attack surface, making Alpine Linux popular for container images and general use.

1.2.1 Package Management

Alpine Linux divides its repositories into three types [11]:

Main Packages in the *main* repository receive direct support and updates from the Alpine core team and the main team.

Community Packages in the *community* repository are created by users in collaboration with the official developers, who are closely involved in the Alpine package development process. The maintenance of these packages relies on the contributions made by the users. If the users cease to support the packages, their availability and functionality may be affected.

Testing Packages in the *testing* repository require testing first. Accepted packages from the testing repository are moved to the community repository. The testing repository is exclusively accessible in the *edge* branch.

Within the edge branch, there is also located the current *development tree*. Therefore, the edge branch is considered a *rolling release model*. Since the edge is considered a development branch, numerous modifications are not thoroughly tested or tested at all, and thus packages in the edge are susceptible to breaking without prior notice.

Alpine Linux releases a stable branch twice a year, receiving support for a designated period. [6] In Alpine Linux, packages are managed using the Alpine Package Keeper; this tool is distributed as *apk-tools*. More information about this tool can be found in Section 1.2.3.4.

The information mentioned above leads to the following constraints:

1. There are no dependencies between individual releases.[5] [6]
2. There are no dependencies between individual architectures.[6]
3. Packages in the individual repositories can depend on each other in the following way:
   a) A package in the testing repository can depend on packages from both community and main repositories.
   b) A package in the community repository can depend on a package from the main repository.
   c) Other dependencies between repositories are prohibited.

These constraints may suggest approaches that could be used when searching for disjoint subgraphs in data modelling. More information can be found in the upcoming Section 2.2.4. All packages can be listed on the web page pkgs.alpinelinux.org or installed using the package manager via apk-tools tool.

1.2.2 Package Structure

This section describes the principles of package creation, its definition, additional constraints, and metadata. Two sources serve as metadata sources for subsequent analysis: the APKBUILD file, which serves as a template for package generation and is described in Section 1.2.2.1, and the resulting binary package, which is described in Section 1.2.2.2.

[5]For example a package released in v3.17 cannot depend on a package released in v3.16.

[6]A package provided for architecture *x86_64* cannot depend on a package provided for architecture *aarch64*.

Both sources have a non-empty intersection, but for future analysis, merging the information from both is necessary. Although the second-mentioned source depends on the first one, there are time delays that can lead to database anomalies. This fact imposes an additional requirement on graph modeling in Section 2.2.

1.2.2.1 APKBUILD

Anyone can create their own package for Alpine Package Keeper. To generate a package for Alpine Linux, an APKBUILD file is used. It serves as a specification for creating one or multiple packages. [12] APKBUILD is a shell script that adheres to a specific structure. More details of this structure are further explained later.

For the purpose of storage, management, and versioning of individual APKBUILDs, a Git repository called *aports* exists. This repository or a so-called *tree* contains the corresponding APKBUILD file for every package.

Packages in Alpine Linux are organized into separate directories based on the specific repository they belong to.[7] Versioning for specific releases (and the edge branch) is managed using git branches. The *master* branch is considered to be the edge. [11]

Additionally, there is an *unmaintained* directory for packages that are no longer supported. Most importantly, unmaintained packages are no longer built. Thus, they do not appear in the Alpine primary mirror.[8]

Brief structure[9] of APKBUILD [13]:

maintainer Package maintainer.

pkgname Name of the packaged software.

pkgver Version of the packaged software.

pkgrel Version of the APKBUILD file, starting from 0, is incremented with each change. This value gets reset when changing `pkgver`.

arch What architectures to build for. This can also be *noarch* in case it is architecture-independent or *all*. Architectures can be negated using the "!" character to exclude them from the list of supported architectures.

license License reflects licensing policy of the packaged software.

[7]As mentioned previously – main, community, and eventually testing in the edge.

[8]Packages in this folder may show signs of errors, and their APKBUILD may not be valid.

[9]Some less properties significant are omitted.

subpackages Multiple packages can be built from one recipe. For example, *doc* and *dev* are the most common *subpackages* we see. However, defining custom subpackages is a possibility.

source List of local files or remote sources to fetch. Checksum calculation is performed on listed files.

secfixes Map of security vulnerabilities (CVE identifier) fixed in each version of the APKBUILD's package(s)

Dependencies are listed in Section 1.2.2.3.

Another component of APKBUILD are functions that run commands to build and test the packaged software and create the resulting packages. Their description is not essential for the further procedure and therefore omitted in this work. APKBUILD is consumed by a tool called *abuild.* This tool is then used to compile and build the resulting binary package(s), which can be further distributed. The binary file is referred to as a *package* from now on and is described in more detail in Section 1.2.2.2.

1.2.2.2 Package

Although the resulting package is generated from the APKBUILD file, it is impossible to rely solely on the data obtained by parsing this file, as not all necessary data can be obtained through static analysis. It is important to note that some data are generated or extended during the build phase of packages, most notably auto-discovered dependencies and providers. The abuild tool inspects ELF headers in binaries to discover dynamically linked libraries, inspects *shebangs* in scripts etc. Therefore, they can only be obtained from the resulting distributed package.

The current version of the binary package file format, *apkv2*, is based on the *gzip* format. The package file consists of three gzip streams concatenated together, each containing a TAR segment [14]:

1. Digital signature,
2. Control segment,
3. Package data.

The critical component for metadata analysis is the control segment which includes installation scripts and, most importantly, the so-called PKGINFO file containing the required metadata.

The PKGINFO file is a plain-text file with key-value pairs. Its content is based mainly on the previously mentioned APKBUILD, but it contains some additional data (according to Alpine Linux Development Team [14]):

size Size of the installed package in bytes.

builddate Time stamp of the date and time when the package was built.

packager Identifier of the builder of the package.

origin Name of the origin package (useful for subpackages).

commit Commit hash of the APKBUILD from which the package was built.

On the other hand, compared to APKBUILD, some data are missing, such as build dependencies, check dependencies, subpackages, security fixes or property `pkgrel`. To be precise, `pkgrel` becomes part of the `pkgver` as a suffix.

Apart from PKGINFO metadata, information about files (more precisely, their paths) from the package data segment is also essential for analysis since we can further investigate possible file conflicts between individual packages.

1.2.2.3 Dependencies

Dependencies between packages are fundamental relationships that need to be explored. This section aims to explain all types of relationships and their connotations thoroughly (according to Alpine Linux Development Team [13]):

checkdepends Dependencies required by the package only during the check phase (i.e. for running tests).

depends Dependencies required by the package during the runtime.

makedepends Dependencies required by the package only during the build.

Dependencies are listed as the name of the required package or the required provider and the optional version constraint, as shown in Code 1.1. Several operators can be used to define the version constraint: `<`, `<=`, `<~`, `=`, `~`, `~`, `>~`, `>=`, `>`, `><`. If no version constraint is specified, any version of the package satisfies the constraint.

```
makedepends="gperf>=0.2.0 autoconf automake libtool"
depends="ethtool wireless-tools iw sqlite grep>=5.8.1-r1"
checkdepends="coreutils"
```

Code 1.1: Example of dependency properties in APKBUILD.

Although the following relationships are not dependencies, they are essential for dependency resolution and are therefore mentioned here:

provides List of provider (package) names (and optionally version info) this package provides. Provider is like a virtual package.

replaces The packages whose files this package is allowed to overwrite (i.e both can be installed even if they have conflicting files).

install_if A set of dependencies that, if installed, trigger installation of this package.

Additionally, the resulting package may contain so-called auto-detected provides:

- `cmd:<name>` Command on `PATH` (e.g. `cmd:git=2.36.3-r0`).
- `pc:<name>` *pkg-config* file (e.g. `pc:shared-mime-info=2.2`).
- `py<abiver>:<name>` Python module (e.g. `py3.10:pygments=2.11.2-r0`).
- `so:<name>` shared library (e.g. `so:libruby.so.3.1=3.1.3`).

It is possible that the provided package does not have a version specified. In this case, the dependency resolution process is based on the comparison of the `provider_priority` attribute with higher `provider_priority` having increased precedence. Later, concerns about this solution were raised in the Alpine community. [15]

The second situation concerns the `replaces` property. Multiple packages may provide a file with the same path. Listing the second package in the `replaces` property allows the first package to be installed simultaneously as the listed package and overwrite conflicting files of the second package. Similar to the `provider_priority` property, the decision is based on the `replaces_priority` property with higher priority having increased precedence.

1.2.2.4 Mirror

As mentioned in Section 1.1.6, a mirror is a server used to distribute packages in specific repositories. In the case of Alpine Linux, the resulting packages are uploaded to the mirror by an automated process after building from aports. Anyone can set up their own mirror and synchronize it with the primary mirror, for example, using the *rsync* tool. Changes to the original mirror, such as adding or removing packages, are reflected depending on the chosen synchronization period. [16]

Each architecture in a given repository has an index of packages. This APKINDEX file is used by tools such as apk-tools to improve searching efficiency. However, this file only contains a subset of the PKGINFO metadata (e.g. it does not contain a list of package files), so it cannot be fully utilized for the metadata analysis mentioned above.

1.2.3 Existing Tools

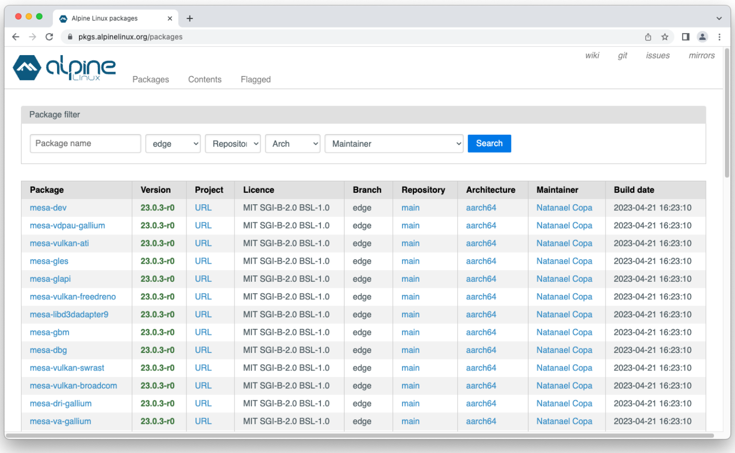
This chapter describes the current tools available to the Alpine Linux community.

1.2.3.1 aports-turbo

aports-turbo is a server-side web application accessible via pkgs.alpinelinux.org. [17] It is written in Lua language and uses the embedded DBMS SQLite. Its database is periodically updated by scripts that are run against repositories on a mirror.

This web application is used by the community and developers to search for information about packages. Users can *flag* packages as outdated to inform their maintainers that a new version of the packaged software is available. However, this is usually not needed as aports-turbo is integrated with the Anitya release monitoring service. [18]

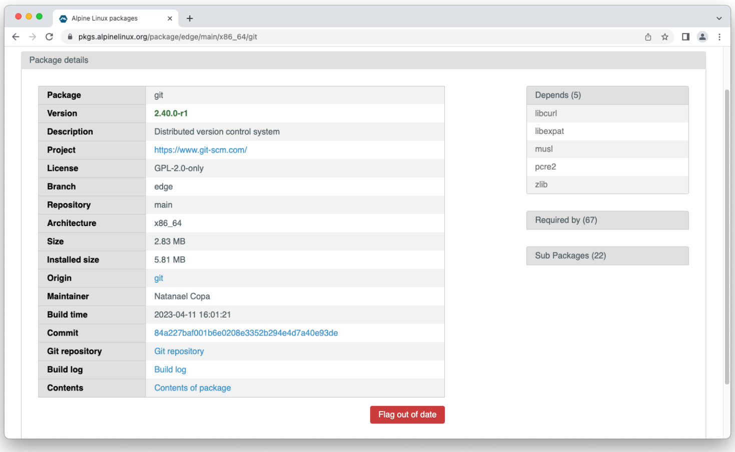
For querying from the user's perspective, this tool allows users to search packages individually based on their name, branch, repository, architecture, and maintainer. It is also possible to search for files based on the file path and file name. It provides an overview of package metadata and shows direct dependencies.



Package	Version	Project	Licence	Branch	Repository	Architecture	Maintainer	Build date
mesa-dev	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-vdpau-gallium	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-vulkan-ati	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-gles	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-glapi	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-vulkan-freedreno	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-libd3dadapter9	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-gbm	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-dbg	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-vulkan-swrast	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-vulkan-broadcom	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-dri-gallium	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10
mesa-va-gallium	23.0.3-r0	URL	MIT SGI-B-2.0 BSL-1.0	edge	main	aarch64	Natanael Copa	2023-04-21 16:23:10

Figure 1.1: aports-turbo search screen.

Initially, aports-turbo operated over a single database but the performance became insufficient as the dataset grew. Therefore, the dataset was divided into individual databases by package branches. Individual branches can be considered disjoint, as discussed in Section 1.2.1.



Package	git
Version	2.40.0-r1
Description	Distributed version control system
Project	https://www.git-scm.com/
License	GPL-2.0-only
Branch	edge
Repository	main
Architecture	x86_64
Size	2.83 MB
Installed size	5.81 MB
Origin	git
Maintainer	Natanael Copa
Build time	2023-04-11 16:01:21
Commit	84a227baf001b6e0208e3352b294e4d7a40e93de
Git repository	Git repository
Build log	Build log
Contents	Contents of package

Depends (5)
libcurl
libexpat
musl
pcre2
zlib

Figure 1.2: aports-turbo package detail screen.

However, advanced queries are not possible.[10] This tool does not meet the requirements stated in Section 2.1.

1.2.3.2 apkbrowser

apkbrowser [19] is a reimplementation of the *aports-turbo* project in Python. The set of functionalities remains the same.

1.2.3.3 Security Issue Tracker

Alpine Linux has its own *Security Fixes Database*, which obtains data from the `secfixes` property inside the APKBUILD file. The database[11] provides structured data in JSON/YAML format, extracted from the comments in APKBUILDs. This data contains information about the CVEs fixed in each version of each aport or package. [20]

[10] E.g. listing transitive dependencies, searching packages based on their dependencies, working with file paths, etc.

[11] The database is available at secdb.alpinelinux.org.

Another tool[12] is the *Security Issue Tracker*, which monitors security issues affecting the Alpine Linux distribution. This tracker is derived from the CVE dictionary, the NVD database and Alpine's own security fixes database. It infers which CVEs have been fixed and which remain unfixed. The data are provided in JSON-LD format. [21]

1.2.3.4 apk-tools

The package manager apk-tools [22] was explicitly designed for Alpine Linux. Its core features include searching, installing, updating, and removing packages.

This tool initially consisted of a group of shell scripts, but it was reimplemented in C. From the perspective of package metadata analysis, it is possible to read package information, including its dependencies or search for packages by name.

Similarly to aports-turbo, advanced queries are not supported by this tool. Neither this tool meets the requirements stated in Section 2.1.

1.2.3.5 abuild

abuild [23] is a tool used for building packages in Alpine Linux. It provides a framework for building packages from source code and is used to generate binary packages in the *.apk* format. Abuild handles dependencies, configures the build environment and the source code, and creates the final package.

It is integrated with the Alpine Linux package repository and allows creation of custom packages for use within the Alpine Linux ecosystem. Package maintainers and developers typically use it to build and distribute packages for Alpine Linux.

1.2.3.6 GitLab

Alpine's GitLab [24] provides a collaborative platform for software development. It is the primary tool the Alpine Linux community uses to manage their packages and source code, allowing users to manage and review code and track bugs, requests, and issues. The Alpine community uses GitLab as a central hub for their package development and maintenance.

[12]The tool is available at security.alpinelinux.org.

1.3 Graph theory

Graphs can be found all over the world. Thanks to them, it is possible to reduce almost every real-world problem to a graph using edges and vertices. These problems can then be solved by using graph theory, which often gives us a set of tools to solve some of these problems efficiently and elegantly.

Let us take the world-famous case of the electrification of Moravia around 1925, where the objective was to design the power lines between villages with minimal usage of power lines for the electric company. [25] This problem led to the discovery of one of the fundamental algorithms in graph theory.[13]

Unless stated otherwise, the following definitions have been adjusted from Diestel [26]:

Undirected graph An *undirected graph* is a pair $G = (V, E)$ of finite sets such that $E \subseteq \{\{u, v\} \mid u, v \in V\}$; thus, the elements of E are 2-element subsets of V. The elements of V are the *vertices* of the graph G, the elements of E are its *edges*.

Directed graph A *directed graph* is a pair $G = (V, E)$ of finite sets such that $E \subseteq \{(u, v) \mid u, v \in V\}$; thus, the elements of E are *ordered pairs* of V. The elements of V are the *vertices* of the graph G, the elements of E are its *edges*. [27]

Multigraph A *multigraph* is a graph $G = (V, E)$ together with map $E \to V^2$.

Subgraph Let $G = (V, E)$. A graph $G' = (V', E')$ is a *subgraph* of G (and G a *supergraph* of G'), if $V' \subseteq V$ and $E' \subseteq E$, written as $G' \subseteq G$.

Path A *path* of length $k \geq 0$ is a graph $P = (V, E)$ of the form:

$$V = \{x_0, x_1, ..., x_k\} \quad E = \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\},$$

where the x_i are all distinct. The vertices x_0 and x_k are *linked* by P and are called its *ends*; the vertices $x_1, ..., x_{k-1}$ are the *inner* vertices of P.

Incident vertex A vertex v is *incident* with an edge e if $v \in e$; then e is an edge at v. the two vertices incident with an edge are its *ends* and an edge *joins* its ends. An edge $\{x, y\}$ is usually written as xy (or yx). If $x \in X$ and $y \in Y$, then xy is an X–Y *edge*.

[13]The algorithm known as Boruvka's algorithm, later simplified and known as Kruskal's algorithm for finding a minimum spanning tree of an undirected edge-weighted graph.

Property graph Property graph G is a tuple $(V, E, \rho, \lambda, \sigma)$, where [28]:

1. V is a finite set of *vertices.*
2. E is a finite set of *edges* such that $V \cap E = \emptyset$.
3. $\rho : E \to (V \times V)$ is a total function. Intuitively, $\rho(e) = (v_1, v_2)$ indicates that e is a directed edge from vertex v_1 to vertex v_2 in G.
4. $\lambda : (V \cup E) \to Lab$ is a total function with Lab a set of labels. Intuitively, if $v \in V$ (respectively, $e \in E$) and $\lambda(v) = l$ (respectively, $\lambda(e) = l$), then l is the label of vertex v (respectively, edge e) in G.
5. $\sigma : (V \cup E) \times Prop \to Val$ is a partial function with $Prop$ a finite set of properties and Val a set of values. Intuitively, if $v \in V$ (respectively, $e \in E$), $p \in Prop$ and $\sigma(v, p) = s$ (respectively, $\sigma(e, p) = s$), then s is the value of property p for vertex v (respectively, edge e) in the *property graph G*.

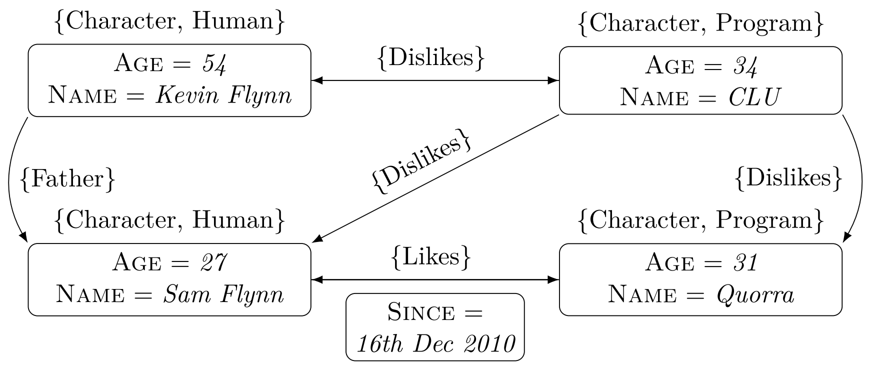


Figure 1.3: Property graph example (including labels and properties).

1.3.1 Storing graphs

There are two commonly utilized approaches for representing a directed multigraph $G = (V, E)$ in a computational context:

Adjacency matrix Let $V = \{v_1, v_2, ..., v_n\}$. The *adjacency matrix* $M(G) = [m_{ij}]$ of a multigraph G is an $n \times n$−matrix such that m_{ij} = number of v_i−v_j edges. [27]

Adjacency list consists of an array *Adj* of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G. [29]

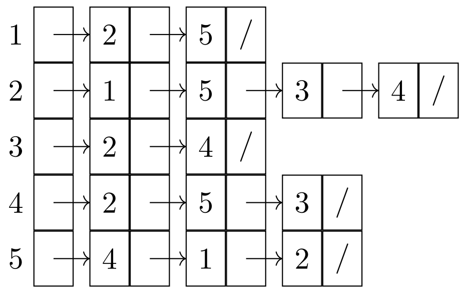


Figure 1.4: Example of an adjacency list.

1.4 Databases

A database is a data collection that is stored electronically in a structured manner for easy retrieval and manipulation of the data. It is typically managed by a database management system (DBMS) that acts as an interface between users and the database.

Since the 1970s, the well-known type of database has been a relational database, which stores data in tables with a defined schema and is designed to adhere to the use of transactions. [30]

A transaction involves a series of operations treated as a single unit of work. The fundamental characteristic of a transaction is that all its constituent actions are executed as a complete set, and if any part of the transaction fails, the engine rolls the database back to its initial state. [31] In other words, a transaction can have only two possible outcomes: successful completion and committing all of its changes or a failure that leads to undoing all its changes.

The ACID properties represent a set of vital transactional features essential to ensure data consistency of state despite concurrent accesses and failures. These properties include *Atomicity*, *Consistency*, *Isolation*, and *Durability*. [32]

Presently, there are discussions among data engineers regarding the limitations of relational databases and exploring alternative non-traditional approaches, known as NoSQL databases, to overcome them. Graph databases are often classified as one type of NoSQL database.

1.4.1 Graph databases

Graph databases are becoming increasingly popular due to their ability to capture and analyse complex relationships between data entities. They provide a more natural way of modelling and querying data, making it easier to manage large interconnected datasets.

As previously mentioned, many structures can be represented using edges and vertices. However, in the field of graph databases, the terms *relationships* and *nodes* are more commonly used. In the following chapters, this terminology is used exclusively.

In contrast to traditional relational databases, traversing highly interconnected data is generally more efficient in graph databases. Traversing relationships in relational databases is achieved using JOIN operations, the complexity of which increases directly with the size of the dataset. In comparison, the complexity of such queries in a graph database should remain roughly constant regardless of the total size of the dataset. [33]

The key features of graph databases can be classified as follows [33]:

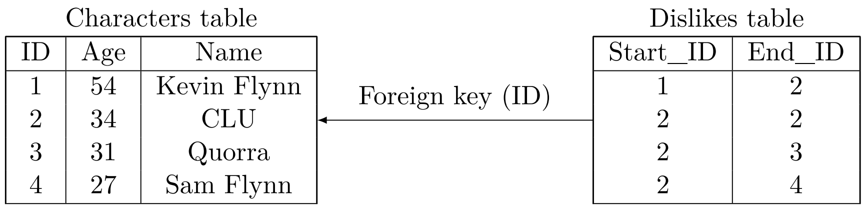
Native processing We say that a graph database uses *native processing* if it exhibits a property so-called *index-free adjacency.* Native graph processing offers faster traversal performance but can make non-traversal queries more difficult and memory-intensive to execute. E.g. Neo4j, TigerGraph.

Native storage Certain graph databases employ a mechanism called *native graph storage*, which has been specifically engineered and optimised to store and manage graphs. The actual implementation varies depending on the specific database used. These are usually modifications of the structures listed in Section 1.3.1.[14] On the other hand, graph databases using native graph storage may encounter challenges in terms of performance when handling *supernodes*, which are nodes with a high degree of relationships, or in terms of scalability. [34]

Non-native storage In contrast, *non-native graph* storage uses non-graph backends, usually well-established relational databases. These are reliable and scalable and are well-understood by operations teams. This provides a more accessible option than native graph processing, which may require more specialised knowledge to implement and maintain. An example of such a database is AgensGraph.

[14]For example, in Neo4j, a doubly linked list is used for representing relationships. [33]

With *index-free adjacency*, each node in the graph contains direct references to its adjacent nodes. This eliminates the need for global indexing (hence *index-free*), as each node acts as a local index of adjacent nodes. Searching indices typically takes $O(\log n)$ in time complexity[15] as shown in Figure 1.5, whereas traversing relationships in index-free adjacency is $O(1)$ in time complexity as shown in Figure 1.6, where n is the size of the dataset. This approach is beneficial for local graph queries, as it reduces the need for expensive indexing operations and allows for faster traversal through direct pointer dereferencing. [35] [36]



Characters table

ID	Age	Name
1	54	Kevin Flynn
2	34	CLU
3	31	Quorra
4	27	Sam Flynn

Dislikes table

Start_ID	End_ID
1	2
2	2
2	3
2	4

Figure 1.5: Graph processing using a table as an index to traverse between nodes. Searching for characters disliked by Kevin Flynn requires searching through *Dislikes table* in $O(\log n)$.

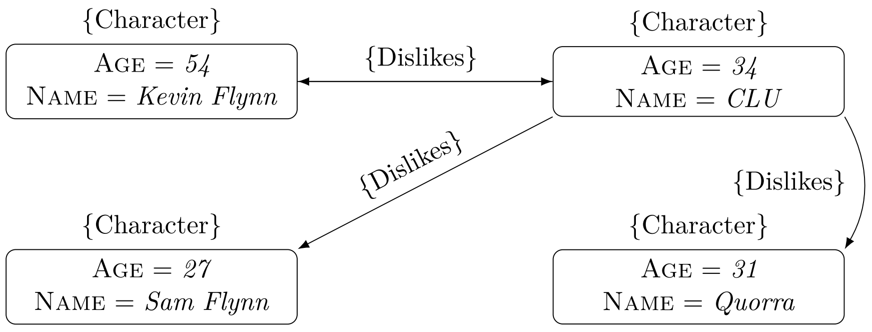


Figure 1.6: Graph processing using index-free adjacency to traverse between nodes. Searching for characters disliked by Kevin Flynn requires searching only through neighbouring nodes in $O(1)$.

For purposes of this thesis, any database engine that behaves like a graph database from an interface perspective (i.e., exposes a graph data model) qualifies as a graph database.

[15]Time complexity for searching in a balanced tree.

Graph databases can also be evaluated based on the representation of the graph itself. The following two methods are among the most popular:

Resource Description Framework In *Resource Description Framework* (RDF), data are represented as a graph that has directed edges and labelled nodes. [37]

Property graphs In the context of property graphs, the conventional representation is a *directed property multigraph*, as previously introduced in Section 1.3. For purposes of this thesis, the term property graph is used as a synonym.

The subject matter of this thesis is limited to the exploration of property graphs since property graphs are a relatively newer concept than RDF. Examples include AgensGraph, Amazon Neptune, ArangoDB, JanusGraph, Neo4j, and TigerGraph. [38]

1.4.1.1 Neo4j

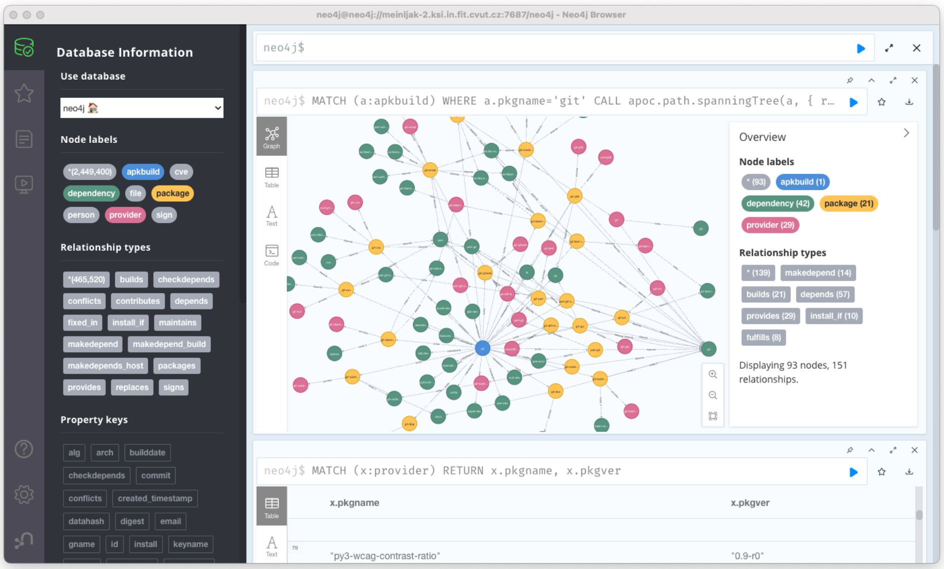
Neo4j is currently the most popular graph database. [39] Like many NoSQL databases, it is schema-free but allows working with constraints. It represents graph structures using property graphs, which can be viewed as *bidirectional* for data querying. Although many NoSQL stores are not transactional, Neo4j is fully ACID-compliant and provides support for transactions. [33]

In contrast to many graph databases such as AgensGraph, Neo4j is considered a graph database with native processing thanks to its index-free adjacency and native storage.

The software is distributed in two editions, the Community Edition and the Enterprise Edition. The former is an open-source version distributed under the terms of the GNU General Public License v3.

Neo4j utilizes the Java Virtual Machine and has various libraries, providing additional functionalities for working with graphs and extending the query language, such as APOC and Graph Data Science (GDS).

In conclusion, Neo4j offers its users a wide range of tools, such as Neo4j Browser, which provides an interactive interface for querying and visualizing data stored in the database, Neo4j Desktop for managing databases, and Neo4j Bloom for graph exploration.



```
neo4j$
```

```
neo4j$ MATCH (a:apkbuild) WHERE a.pkgname='git' CALL apoc.path.spanningTree(a, { r…
```

```
neo4j$ MATCH (x:provider) RETURN x.pkgname, x.pkgver
```

	x.pkgname	x.pkgver
79	"py3-wcag-contrast-ratio"	"0.9-r0"

Figure 1.7: Neo4j Browser tool interface.

1.4.1.2 AgensGraph

AgensGraph is a database management system that combines two data models – graph model and a relational model. Thus it is called *multi-model.* It is a *fork* of the well-known PostgreSQL database management system and adopts both SQL and Cypher languages even within one single query. Same as Neo4j, it uses property graph structure, it is schema-free and it is ACID-compliant thanks to the underlying relational database, which is then considered non-native.

Although, in 2017, the vendor of AgensGraph claimed that their database outperformed Neo4j in all aspects based on the *LDBC*[16] benchmark with an average 50×improvement in time complexity, a peer-reviewed study from 2020 contradicts this statement using the same LDBC benchmark methodology. [40] [41]

According to the same study, AgensGraph timed out even under a small dataset[17] while finding paths by length from 3 to 5. Overall, AgensGraph outperforms Neo4j in SQL-accompanied workload and simple update and query operations while performing badly in processing complex queries and managing large datasets. [41]

[16]The Linked Data Benchmark Council, ldbcouncil.org.
[17]Including 3.182 mil. Nodes and 17.256 mils. relationships.

1.4.1.3 Query Languages

Both Neo4j and AgensGraph database management systems utilize the Cypher language, respectively OpenCypher[18], for property graph querying. Other systems using Cypher include, for example, Memgraph, RedisGraph and TuGraph. The language was initially introduced in 2011 by Neo4j, and its syntax draws inspiration from SPARQL. Cypher's pattern-matching functionality is a crucial feature that allows users to define a pattern of nodes and relationships to match in a graph. This feature has a shallow learning curve, making it relatively easy for new users to learn and leverage in their data analysis and interpretation tasks.

Over time, the absence of a standardized query language for property graphs has become increasingly apparent. In 2015, as a response to this issue, Neo4j launched the OpenCypher initiative, enabling Cypher to adopt as a general language for property graphs. While this project has led to the growing industrial use of Cypher, it has not been able to establish a universal standard on its own. [38]

```
MATCH (a:Character)-[:DISLIKES]->(b:Character)
WHERE a.name = "Kevin Flynn"
RETURN b.age
```

Code 1.2: Example of Cypher query that returns a list of the ages of characters disliked by Kevin Flynn.

To address this challenge, the Joint Technical Committee 1 of ISO/IEC, which defines information technology standards for the International Organization for Standardization and International Electrotechnical Commission, approved the development of a new property graph query language in 2019. This new query language, called GQL, aims to extend SQL with the pattern-matching paradigm of Cypher and thus provide a standardized query language for property graphs. [42] The expected declaration as an international standard is April 2024. [43]

Another widely used query language is Gremlin, which is used in Apache TinkerPop. This language is utilized by many graph DBMS such as JanusGraph. It is a Turing-complete traversal language incorporating functional programming elements and does not rely on pattern matching. ArangoDB Query Language (AQL) is another query language.

```
g.V().has("name","Kevin Flynn").out("dislikes").values("age")
```

Code 1.3: Example of Gremlin query that returns a list of the ages of characters disliked by Kevin Flynn.

[18]The terms Cypher and OpenCypher may be used interchangeably in this thesis.

Chapter 2

Analysis and Design

In continuation of Chapter 1, this chapter aims to identify and consolidate all the functional and non-functional requirements, constraints, and specifications for the proposed solution. To achieve this goal, domain knowledge is utilized to transform the problem into a graph model. Additionally, the chapter considers the estimated size of the dataset and explores potential *sharding* options for future scalability.

Finally, a suitable graph database management system is selected based on the given parameters. Through this process, this chapter lays the foundation for the subsequent development and implementation stages of the proposed solution.

2.1 Requirements Analysis

In this section, representatives of the Alpine Linux community were contacted to provide specific requirements, including queries they cannot evaluate with current tools. As is common in software engineering, requirements are typically divided into two categories:

Functional requirements describe what the system should do and usually refer to the application's features.

Non-functional requirements describe how the system should perform and are not directly related to the application's functionality.

Collecting requirements is crucial for selecting a suitable graph DBMS and creating an appropriate data model. Understanding the functional and non-functional requirements helps to make an informed decision and avoid potential issues in the future.

ID	Description
FR1	Members of the Alpine community are allowed to write their own queries for reading data and expect an easily understandable query language.
FR2	Regularly monitor package changes and incrementally store them while preserving previous data.
FR3	Enables the control of the constraints Alpine imposes on its packages.
FR4	Collect package metadata for multiple repositories, architectures and release branches.
FR5	Collect available package metadata from both binary packages and APKBUILDs.
FR6	Allow queries across branches.

Table 2.1: Functional requirements

ID	Description
NR1	Read transactions are a priority from a performance perspective; they are more common than write transactions.
NR2	Used license must comply with Alpine policies.
NR3	Code base must be open-source.
NR4	Runs on Alpine Linux.
NR5	Performs well on a dataset with a **minimum** of 4 mils. highly connected nodes.
NR6	Access control is necessary, thus normal users can perform *read-only* operations only.
NR7	DBMS can limit the size of the query in order to prevent DOS attacks.

Table 2.2: Non-functional requirements. Computation for *NR5* is performed in Section 2.3.

Together with both functional and non-functional requirements, several actors have been identified:

1. Package Maintainers,
2. Consistency Checker,
3. Regular Users.

As one of the actors is the so-called Consistency Checker, the database must allow storing data that violates the schema given by Alpine Linux in order to be able to detect these violations using queries. A total of 16 use cases were collected, which are part of Appendix D and their evaluation is detailed in Chapter 4.

2.2 Data Modeling

This section presents the domain modelling of the Alpine Linux package system, previously described in Section 1.2. It starts by using an E-R diagram to describe the essential entities, properties, and relationships in the conceptual modelling approach. Core entities, such as APKBUILD, Package, File, Dependency and Provider, are introduced. The following subsection focuses on the relationships between the entities, outlining the various associations among them.

Additionally, the section describes how the conceptual model was transformed into a graph model. It highlights the benefits of graph databases over traditional relational databases in terms of modelling and explains the considerations made during the modelling process.

2.2.1 Conceptual schema

In Section 1.2, two data sources for the database were identified – APKBUILD and the resulting binary package. As mentioned earlier, the resulting packages are generated using the abuild tool from the APKBUILD files. Despite overlapping data and certain duplications, the model treats these data sources as independent entities. This approach is adopted to avoid reliance on their proper synchronization. Therefore, when designing the data model, it is necessary to consider this limitation and consider such anomalies.

The following chapters primarily focus on the properties and relationships of the package entity, with a similar approach taken for APKBUILD.

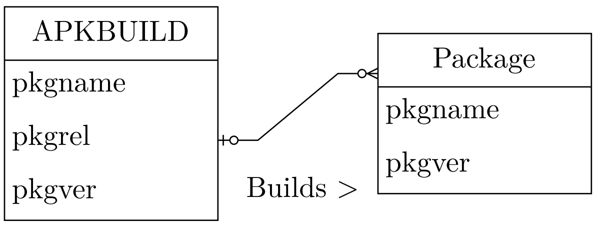


Figure 2.1: E-R diagram of APKBUILD and Package.

A combination of properties `pkgname` and `pkgver` (respectively `pkgrel`) serves as a unique identifier.

2.2.1.1 Repository

According to the domain of Alpine Linux packages, a package and its corresponding APKBUILD should belong to one repository. However, it is possible for a package to appear in two repositories simultaneously, either due to an error or natural development, such as during the transition from testing to community. Therefore, the conceptual model is designed to be more flexible in this regard, allowing for the detection of potential errors or inconsistencies in the data.

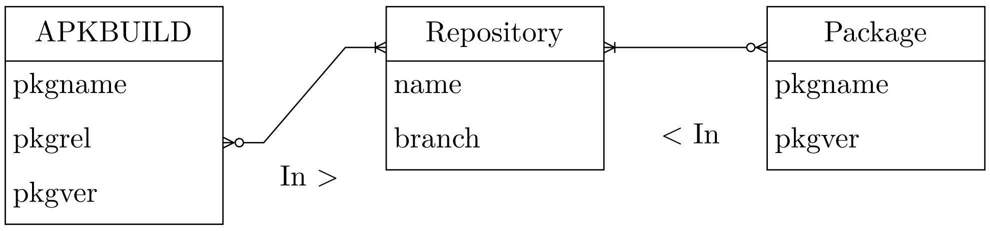


Figure 2.2: E-R diagram of APKBUILD, Repository and Package.

2.2.1.2 Architecture

The role of entity Architecture differs depending on whether it is in the context of an APKBUILD or a resulting package. In the APKBUILD specification, a list of architectures for which the resulting package is to be built can be specified. This list may also include the keyword *all* along with individual negated architectures. For such negated architectures, the package is not built.

The edge between the APKBUILD and a specific Repository indicates for which architectures the package is built. The absence of this edge expresses negation in the APKBUILD.

```
arch="all !armhf !s390x !ppc64le !mips !mips64"
```

Code 2.1: Example of architectures property in *aports/main/gnu-efi/APKBUILD*. [44]

The entity Package is then built for individual architectures. The edge between the Package and a specific Architecture indicates for which the Package is built. The Package is built for only one architecture.

The architecture named *noarch* represents a particular type of architecture. This refers to a package that is not architecture-dependent.

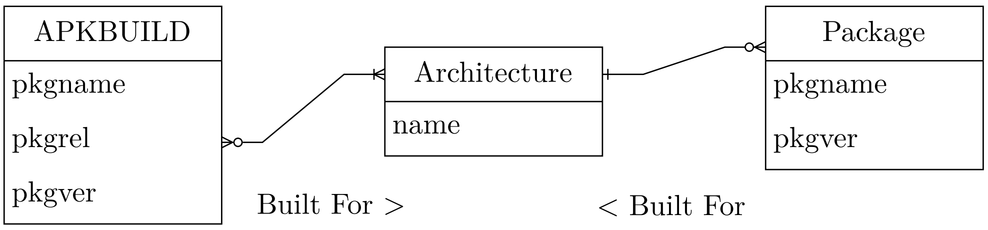


Figure 2.3: E-R diagram of APKBUILD, Architecture and Package.

2.2.1.3 Provides

As mentioned in Section 1.2.2.3, each package creates its own virtual packages called providers (via property `provides`). Additionally, the package itself is also considered as a provider for the purposes of this model. This modification is utilized in Section 2.2.1.4.

A combination of properties `pkgname` and `pkgver` serves as a unique identifier. It should be noted that, unlike packages, providers may not be versioned.

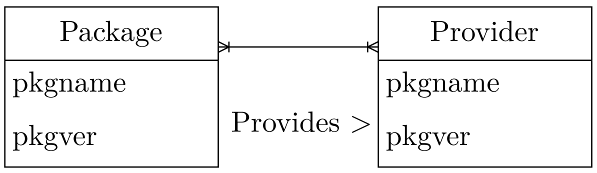


Figure 2.4: E-R diagram of Package and Provider.

2.2.1.4 Dependencies

As mentioned in Section 1.2.2.3, each package has various types of dependencies. However, all dependencies can be modelled as a single entity, as they share a common principle. The role of each dependency in a given package is distinguished by its type of relationship. This applies to both the package and APKBUILD data sources. A combination of dependency entities `pkgname` and `ver_constraint` serves as a unique identifier.

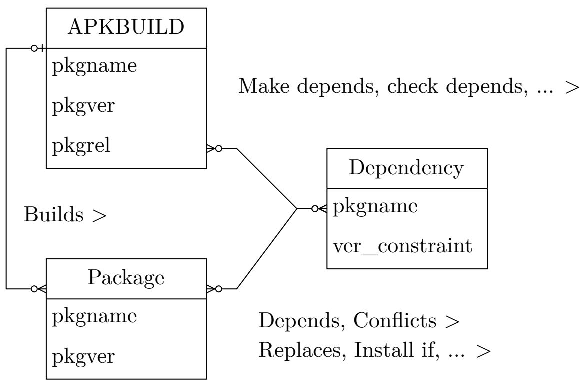


Figure 2.5: E-R diagram of APKBUILD and Package together with Dependency satisfied by Provider.

One of the primary roles of a dependency is to determine whether it has been satisfied. Dependency entities can only be satisfied by a provider entity. The process of satisfying dependencies is described in detail in Section 1.2.2.3. In essence, a provider entity can satisfy a dependency only when its `pkgname` and `pkgver` match the `ver_constraint` requirements specified by the dependency.

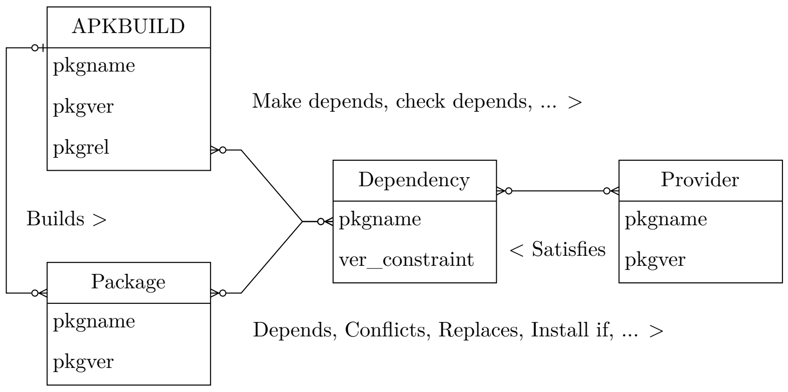


Figure 2.6: E-R diagram of APKBUILD and Package together with Dependency.

2.2.1.5 Files

As mentioned in Section 1.2.2.3, each package can provide multiple files identified solely by the `path` property.

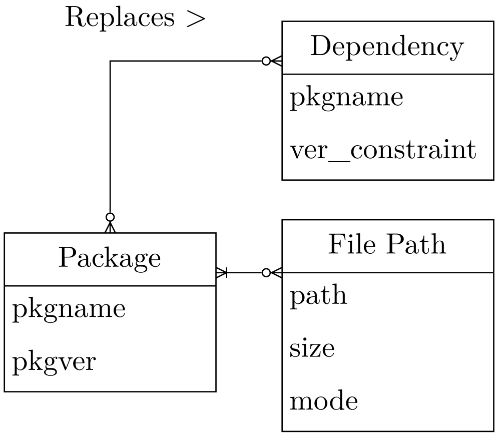


Figure 2.7: E-R diagram of Package and File.

However, it is possible that multiple packages provide a file with the same path. While this scenario is undesirable, it is allowed if the package providing the file with the same path is listed as a dependency in the `replaces` relationship. In case the `replaces` relationship is not listed among these two packages, they cannot be installed at the same time.

This situation is not a problem when the mutual installation of packages is prevented by other criteria. For instance, if one package explicitly lists another as a conflicting dependency.

2.2.2 Graph Model

According to Pokorný and Valenta [31], in the case of a relational model, the conceptual model needs to be transformed into a relational one in several steps (simplified):

1. Representation of entity types: This is typically done by directly converting entities into tables.

2. Representation of relationship types: The process varies depending on the relationship's cardinality – 1:1, 1:N, and M:N, which must be decomposed. This results in the creation of additional tables to represent some types of relationships.

3. Normalization.

However, in the case of a graph model (according to Robinson et al. [33]) this entire process is straightforward and does not require any particular conversion of the conceptual model. On the contrary, it is possible to convert the conceptual model into a property graph directly, but the following techniques should be kept in mind:

1. Represent entities as nodes.
2. Relationships and their orientation represent the relations between entities.
3. Entity attributes are represented as node properties.
4. Labels typically represent the role in the domain, and a single node may belong to multiple (sub)domains.
5. Relationships can also be viewed as connecting different domains.
6. Details of the relationship between entities can be specified as the properties of the relationship.

Although converting a conceptual schema to a graph model (more precisely, a property graph) is straightforward compared to converting to a relational model, some techniques from graph theory have been utilized. They have been mainly used for query optimization and simplification of relationships.

Regarding files, all of their attributes except for the `path` attribute have been separated and added as relationship properties. As a result, the nodes representing files are uniquely identified by the `path` attribute, significantly simplifying any operations or manipulations involving this entity. To speed up searching, folders are also stored in the same way as file nodes. This optimization is used, for example, in query D.1.5. Further optimization can be achieved by representing the folder structure as a tree using relationships between file nodes.

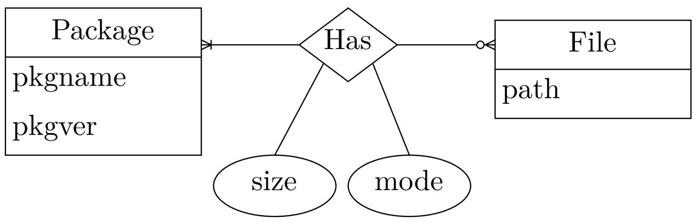


Figure 2.8: Graph diagram of Package and File.

According to Robinson et al. [33], a structure called *linked list* is suitable for the rapid traversal of time-ordered events. One of the suitable applications could be the attribute `pkgver` (i.e. version) of individual packages. These can be distinguished from each other using the relationship type *previous*, where we consider this relation as a *strict order* [45].

Definition. *Let $\prec$ be a relation on a set S. Then $\prec$ is a strict ordering on S if and only if $\prec$ satisfies the strict ordering axioms:*

1. Irreflexivity: $\forall a \in S : \neg(a \prec a)$
2. Asymmetry: $\forall a, b \in S : a \prec b \Rightarrow \neg(b \prec a)$
3. Transitivity: $\forall a, b, c \in S : (a \prec b) \wedge (b \prec c) \Rightarrow (a \prec c)$

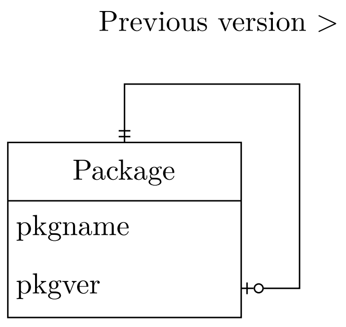


Figure 2.9: Diagram of the relationship between Package and its previous versions.

Using a versioned graph allows the recovery of a specific state of the graph. However, most graph databases do not support versioning as a primary concept. Creating a versioning scheme within the graph model by timestamping nodes and relationships is possible, but this adds complexity to queries written against the graph. [33]

2.2.3 Incremental changes

Functional requirement *FR2* stated in Section 2.1 demands that the modelled graph should be capable of incrementally adding new data, for instance, when a new version of a package is released. For this reason all changes are resolved additively, i.e. there is no deletion of old data.

Graphs possess an additive nature, allowing for introducing new nodes, labels and relationships into an existing structure without impacting the existing structure. [33] In contrast, relational databases, which are not schema-free, encounter more challenges when dealing with structural changes in package management in future versions of Alpine Linux.

2.2.4 Subgraphs

This section explores possible dataset partitioning methods, specifically identifying suitable subgraphs for potential sharding. This may be necessary if the dataset in the database becomes too large or grows over time, thereby reducing the DBMS's ability to execute transactions within a reasonable time or leading to high memory or storage usage.

As revealed by the aports-turbo tool described in Section 1.2.3.1, the dataset can be divided to some extent based on branches. While this division may result in specific duplicates, such as in the case of entities like a person who acts as a package maintainer, it is feasible.

However, it is essential to note that dividing the dataset based on branches is not suitable for analyzing the history of packages in general. This limitation arises from the release cycle of Alpine Linux itself. New development always occurs in the edge branch, while for each release, the edge branch is branched off into its release branch (e.g. v3.17). Consequently, when discussing the package history, all packages would be present in the edge branch, while the history of packages in a particular release branch would only begin with its branching. Therefore, when considering potential divisions, it is necessary to acknowledge that each decision carries inevitable trade-offs.

As discussed in Section 1.2.1, another way to partition the dataset is based on package architecture, as they must be disjointed in dependencies. However, as elaborated in Section 2.3, this may not necessarily be a reasonable approach. This decision is endorsed in the Section 4.3.

It is impossible to partition the dataset based on repositories from a dependency perspective, as explained in Section 1.2.1, as packages in the community repository may have dependencies on packages in the main repository.

While various methods of dataset partitioning can be identified, it is essential to consider that one of the fundamental requirements of the tool being developed is the ability to analyze dependencies.

2.3 Dataset Estimates

In order to perform requirements engineering and subsequently select an appropriate database, it is necessary to make a rough estimate of the dataset size.

Using the tools mentioned in Section 1.2.3 and [16], the following values were collected[19]:

- *Number of architectures*: 8
- *Number of stable releases per year*: 2
- *Number of stable APKBUILDS in latest release*[20]: 6608
- *Number of x86_64 stable (sub)packages*[21]: 17811
- *Average number of (runtime) dependencies per (sub)package*: 3.6
- *Average number of providers per (sub)package*: 1.09
- *Average number of files per (sub)package*[22]: 105

It can be inferred that the majority of the dataset will consist of files. However, analysis of packages across two architectures revealed that roughly 85 % of file paths are shared, significantly reducing the number of file nodes. This finding should be taken into account when estimating the dataset size. Moreover, as an optimization strategy, storing packages for different architectures in the same database may be advisable.

The following estimates apply only to a single stable release and define the rough minimum size of the dataset stored in the database.

- *Package nodes*: ≈200,000
- *Provider nodes*: ≈220,000
- *Dependency nodes*: ≈700,000
- *File nodes*[23]: ≈3,000,000

A rough estimate reveals that representing a single stable branch and all of its architectures will require several million nodes, approximately 4 million nodes. This calculation, for example, does not account for shared dependency nodes and completely disregards relationships or nodes whose number is negligible in these orders of magnitude.

[19] Measurements were taken for release 3.17.
[20] 1558 in main and 5050 in community.
[21] 4989 in main and 12822 in community.
[22] Measured main and community repository for *x86_64*.
[23] ≈1.8 mil. nodes for a single architecture.

2.4 Graph DBMS Selection

The first criterion for selecting a Graph DBMS was the query language. Functional requirement *FR1* states that more comprehensive community can perform queries without requiring in-depth knowledge of graph databases.

According to community size, as shown in table 2.3, and a good learning curve[24], the choice of the query language was Cypher, previously described in Section 1.4.1.3.

Further formal requirements from a licensing standpoint and the demand for open-source code were placed on the Graph DBMS. Thus, the following graph DBMS were excluded from the selection: Amazon Neptune, MemGraph and TigerGraph.

The narrow selection included AgensGraph, Neo4j, and TuGraph. In Section 2.3, the estimated dataset size was taken into account, resulting in non-functional requirement *NR5.* Based on the study [41], AgensGraph, which had an inadequate performance on a dataset of comparable size, was removed from the selection. Another disadvantage of AgensGraph was its small community, as shown in Table 2.4, and a lack of tools compared to Neo4j.

Its underlying engine (PostgreSQL), widely popular with its multi-model approach, could be considered an advantage. Although the multi-model approach can be considered an advantage, replacing it with more single-model DBMSs and then orchestrating them within the application layer is usually possible.

Undoubtedly, concerns with larger datasets arose with Neo4j as well. However, these concerns were limited to *write transactions* only which do not have a performance requirement according to Section 2.1. Although TuGraph appears to be a more powerful DBMS from the perspective of the mentioned study, both remaining DBMSs, Neo4j and TuGraph, meet the non-functional requirements regarding performance.

Name	Stack Overflow tags
AQL	440
Cypher	9,624
Gremlin	3,434

Table 2.3: Comparison of query languages for graph DBMS. Performed on April 15th 2023.

[24]From the perspective of the author of this thesis.

However, limited support for Cypher language and its relatively new open-source status in September 2022 are some of TuGraph's drawbacks, together with *NR7*, which could not be confirmed since its limited documentation. It has a tiny community and some documentation is in Chinese. Non-functional requirement *NR7* can be satisfied in Neo4j using configuration by limiting memory per transaction. Since Alpine Linux provides a package for Neo4j, the non-functional requirement *NR4* is also satisfied.

Name	Stack Overflow tags	GitHub stars	License	Query language	Type
AgensGraph	109	1,300	Apache 2.0	Cypher + SQL	Non-native
ArangoDB	1,954	12,900	Apache 2.0	AQL	Hybrid
JanusGraph	818	4,800	Apache 2.0	Gremlin	Hybrid
neo4j	22,488	11,300	GPL v3	Cypher	Native
TuGraph	0	580	Apache 2.0	Cypher (limited)	Native

Table 2.4: Comparison of graph DBMS. Performed on April 15th 2023. GitHub star numbers are approximate. Type is according to [41].

Based on the above, Neo4j was selected as the Graph DBMS, which has the largest community, extensive documentation and user-friendly tools. Neo4j satisfies all the requirements placed on graph DBMS regarding both functional and non-functional requirements.

Chapter 3

Implementation of Proof-of-Concept

This chapter provides insight into the implementation aspect of the developed proof-of-concept. The chapter begins with a high-level overview of the chosen software architecture, explaining the different layers and their interaction. Subsequently, it describes the utilized design patterns and highlights key design decisions.

At the lowest level, it focuses on the chosen technologies, describing their benefits and drawbacks while justifying specific choices. Additionally, it addresses the more specific implementation of significant classes and mentions some technical challenges or complex solutions.

3.1 Architecture

The significance of software architecture cannot be underestimated. Well-designed architecture can facilitate software development and improve its sustainability and testability, naturally leading to higher reliability. Architecture provides a high-level view of software and primarily focuses on fulfilling non-functional requirements instead of software design, which mainly deals with functional requirements.

According to Martin [46]: *"The goal of software architecture is to minimize the human resources required to build and maintain the required system."*

Although there are numerous architectural design patterns, the vast majority of them strive for the same goal: separation of concerns. The so-called multi-layered architectures attempt to achieve this goal by dividing software into several layers.

For this thesis, architecture inspired by the concept of The Clean Architecture, as shown in Figure 3.1, was selected. This concept was developed by Robert C. Martin, also known as Uncle Bob. Other popular approaches include Hexagonal Architecture and Onion Architecture. [46]

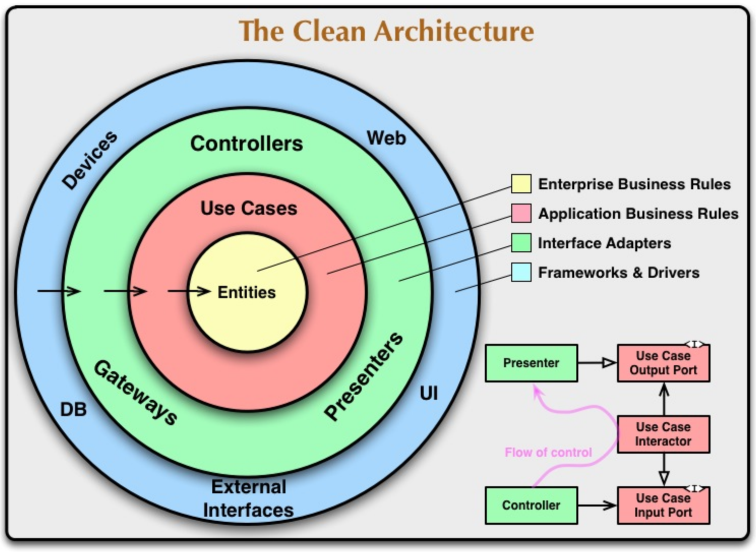


Figure 3.1: The Clean Architecture [46].

The resulting application can be divided into following parts (as shown in Figure 3.2):

Controllers This layer processes and handles input from the user and presents output.

Interface Adapters This layer mainly contains implementations of *Repositories* whose contract was defined in lower layers, on which this layer depends. The task of this layer is to mediate access to data sources, specifically the Neo4j database. The DTO pattern is utilized for the purpose of converting data between entity representation and database representation, which is presented in detail in Section 3.2.

Use cases Use cases are the core part of the application's business logic. Nevertheless, to allow use cases to access database data, this layer would have to become dependent on the Interface adapters which is prohibited in the proposed architecture. Therefore, the *Dependency Inversion Principle* is utilized, described in detail in Section 3.2. Thus, use cases use *Repository interfaces* as a form of contract rather than a direct dependency.

Entities Entities are representing the essential and abstract concepts of the application's business logic.

The architecture of the application, which was inspired by The Clean Architecture, is illustrated in Figure 3.2. Some less significant parts have been omitted for the sake of clarity.

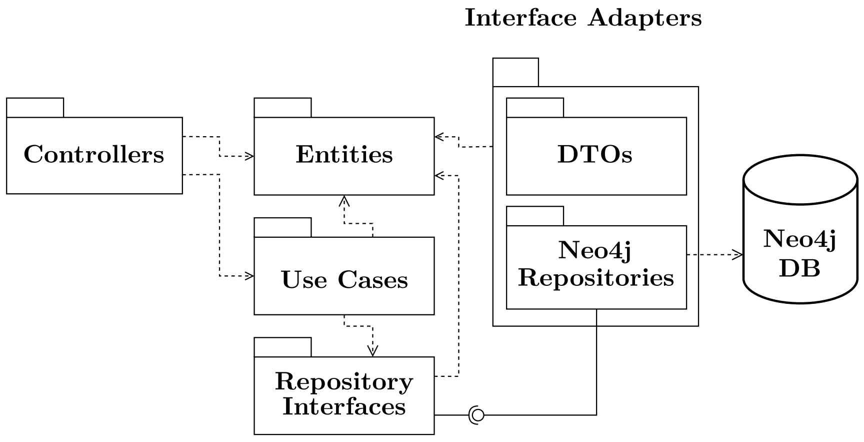


Figure 3.2: Architecture of the application.

3.2 Design Principles

As software architecture provides a high-level view, software design deals with the design of specific methods and classes, and how these classes should be interconnected. Software design can be viewed as a set of principles and patterns. According to Martin [46], the most significant principles can be considered the so-called SOLID principles:

Single-responsibility principle Each class should have only one responsibility.

Open–closed principle *"Software entities ... should be open for extension, but closed for modification."* [47]

Liskov substitution principle *"Build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another."* [46]

Interface segregation principle *"Clients should not be forced to depend upon interfaces that they do not use."* [48]

Dependency inversion principle *"The code that implements high-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies."* [46]

Recurring issues in software design are commonly addressed by utilizing so-called *design patterns*. One of the most significant sets of design patterns is the *Gang of Four* (GoF). In the implementation of the proof-of-concept, several patterns were used, such as the Data Transfer Object (DTO), which serves to transfer data between the database and the entity, or the Repository pattern, which provides an interface between the database and the domain. [49]

Neo4j provides the library *Neo4j-OGM* for *Object-Graph Mapping* (OGM), which is the equivalent of well-known *Object-Relational Mapping* (ORM) in the world of relational databases. Considering the fact that this work is a proof-of-concept, the author decided not to utilize this library.

From the perspective of the developed application, there will be more emphasis on larger queries for bulk writes to the database. Therefore, using such a library at this stage of the application would rather add complexity than being beneficial.

3.3 Technologies

This section focuses on the specific technologies that were utilized in the implementation.

3.3.1 Neo4j Integration

Selected Graph DBMS Neo4j provides several officially supported drivers for Java, JavaScript, Python, .NET, and Go programming environments. In addition to the drivers, this DBMS provides a range of libraries, such as the GDS, as mentioned above. It also offers a variety of connectors, including integration with GraphQL, which opens up possibilities for potential future development.

Neo4j is written in the Java and Scala programming languages. Therefore, the driver for Java is by far the most widely used and offers features that other drivers do not have, such as the Traversal Framework. This framework has broader graph traversing capabilities than Cypher. This is because, instead of Cypher, the framework can dynamically make choices on how to traverse the graph at each step of traversal. Its expressive power is then higher than the expressive power of Cypher. In addition to drivers, the Traversal framework can also be utilized in procedures, which are created using plugins in Neo4j. [50]

Although Neo4j is schema-free, it is still possible to ensure data consistency, to some degree, by using constraints. In this project, constraints are used to ensure the uniqueness of primary keys. Indices are also utilized for query performance optimization.

3.3.2 TypeScript

TypeScript is a high-level programming language developed by Microsoft, a syntactic superset of the JavaScript language. [51] It is transpiled into JavaScript during compilation. Unlike JavaScript, TypeScript has a powerful type system and static typing that is evaluated during compile time. It is a multi-paradigm language that is primarily oriented towards functional and object-oriented paradigms.

Since TypeScript is a superset of JavaScript, its type system can be disabled in cases where it may be too restrictive, which can be especially useful in creating a proof-of-concept.

Although the previous Section 3.3.1 suggests that the Java environment is more appropriate for developing an enterprise-level application using Neo4j, the author chose TypeScript for its lower complexity. This was mainly because the emerging application is in the proof-of-concept phase. The primary goal of this work is to verify whether graph representation is suitable for this domain at all. Any further development towards a fully usable tool may be the subject of follow-up work.

3.3.3 Node.js

Node.js is an open-source JavaScript runtime environment built on Chrome's V8 JavaScript engine, using an event-driven, non-blocking I/O model. It allows developers to execute JavaScript code on the server-side, outside of a web browser. Node.js was initially released in 2009 and it is currently maintained by the OpenJS Foundation.

3.4 Background

This section dives into implementation details, describing specific types of classes, such as Entities or Repositories, and builds upon the forenamed layering approach using *mocking* techniques. Subsequently, it discusses more complex technical solutions, such as dependency tree traversal.

3.4.1 Package Metadata Extraction

As mentioned in Section 2.2.1, two types of files need to be used as data sources. Extracting metadata from these formats is non-trivial, so an external package called *alpkit* was used to extract this data to JSON format. The alpkit tool was originally developed for the needs of this thesis.

3.4.2 Controllers

Controllers serve as an interface between users and business logic, their only task is to process the user input and pass the control to the business logic. Since the purpose of the application at this time is importing data into the database, only a command-line interface is available.

```
apk-importer bootstrap v3.16 x86_64 main
```

Code 3.1: Example of a command to run the application.

Controllers are implemented as higher-order functions processing the data from the CLI, preparing the dependencies, and passing the control to a specific use case.

3.4.3 Repositories

As mentioned earlier, repositories (according to the Repository pattern described by Fowler [49]) serve as an interface between the database and business logic. In other words, other layers should not be aware of whether data was retrieved from the database or from memory. An example of a repository was provided in Listing 3.2.

```
export class CveNeo4jRepository
    implements ICreateRepository<ICve> {
  constructor (private _session: Session) {}

  public async create({ id, fixedIn }: ICve): Promise<void> {
    await this._session.run(`
      MERGE (c:cve { id: $id }) WITH c
      MATCH (a:apkbuild) WHERE a.pkgname=$pkgname AND
        a.pkgver=$pkgver AND a.pkgrel=$pkgrel
      MERGE (c)-[:fixed_in]->(a)
    `, { id, pkgname: fixedIn.pkgname, pkgver: fixedIn.pkgver,
        pkgrel: fixedIn.pkgrel });
  }
}
```

Code 3.2: Example of a Repository class.

3.4.4 Use Cases

Use cases encapsulate a specific user requirement, incorporating the essential logic and operations necessary to fulfil one particular functionality, and they are fundamental in implementing the system's business logic.

Use cases are implemented as a higher-order functions where the outer function serves for passing dependencies.

```
export const apksBootstrapUseCaseFactory = (
  logger: ILogger,
  parser: IFileParser<IApk>,
  // ...
) => async (): Promise<void> => {
  const filePaths = await filePathsGetter.getFilePaths();
  const apks = await parser.parse(filePaths);

  if (!conf.disableFiles) {
    await filesImportUseCase(apks);
  }

  await archRepository.createBulk(apks.map(({ arch }) => arch));
  await createApkRepository.createBulk(apks);
  await dependencySatisfaction();
  await vdependencyRepository.create();
}
```

Code 3.3: Sample of a use case implementation (adjusted).

3.4.5 Entities and DTOs

The entities are almost identical to the entities modelled in Section 2.2. For simplicity, they are usually defined in the form of an interface. In more complex cases, for example, when derived properties are required, they are defined as a class with derived properties in the form of methods.

```
export const mapFileEntityToDto =
    (entity: entities.IFile): IFileDto => {
      try {
        return {
          path: entity.path,
          type: entity.type,
          link_target: entity.link_target || null,
          uname: entity.uname || null,
          gname: entity.gname || null,
          size: entity.size || null,
          mode: entity.mode,
          device: entity.device || null,
          digest: entity.digest || null,
          xattrs: entity.xattrs
            ? JSON.stringify(entity.xattrs)
            : null,
        }
      } catch(e) {
        const message = 'Cannot convert File entity to DTO';
        // ...
        throw new Error(message);
      }
    }
```

Code 3.4: Mapper from Entity to DTO Sample Sample of a mapper from an Entity to DTO (adjusted).

Data Transfer Objects (DTOs) transfer entities between different representations. They are used to represent data from the alpkit tool and to represent data in the database since they are not directly stored as entities. One of the reasons why entities cannot be stored directly in the database is, for example, the representation of numbers in Neo4j. This database represents numbers as 64-bit, while Node.js represents numbers in 32-bit. Therefore, it is necessary to ensure adequate conversion when reading and writing. A simple mapper is used for such conversion between DTOs and entities, represented as a function, as shown in Code 3.4.

```
export interface IFileDto {
  path: string;
  type: string;
  mode: string;
  link_target: string | null,
  uname: string | null,
  gname: string | null,
  device: string | null,
  digest: string | null,
  xattrs: string | null,
  size: number | null,
}
```

Code 3.5: FileDTO Sample.

3.4.6 Mocking

The technique called *mocking* is widely used in testing and development. Its principle is to enable the isolated development or testing of a particular class (or a package) by replacing its specific dependencies with a *fake* substitute considered functional and therefore not subject to testing.

```
export class ApkTestingParser extends ApkParser
    implements IFileParser<IApk> {
  public override async parse(path: string): Promise<IApk> {
    const buffer = await readFile(path);
    const dto: apkInspectMappers.IApkDto = JSON.parse(
        buffer.toString()
    );

    return apkInspectMappers.mapApkDtoToEntity(dto);
  }
}
```

Code 3.6: Testing Parser class used for mocking (adjusted).

Thanks to the layered architecture, as shown in Section 3.1, and adherence to SOLID principles, as shown in Section 3.2, especially the Single-Responsibility and Interface Segregation principles, it is possible to easily replace the individual layers and dependencies of specific classes, making it easy to perform unit testing of specific classes or methods.

As package metadata extraction takes a significant amount of time to run, pre-processed data in the form of JSON files were used during the development and testing of other layers instead of repeated data extraction. Similarly, classes for working with aports repository or Alpine mirror were mocked.

3.4.7 Dependency satisfaction

One implementation detail worth mentioning is the potential satisfaction of dependencies. It is important to mention right at the beginning that this work does not aim to create a full-fledged dependency solver in a graph database. For example, proposed solution does not consider `provider_priority`. That may be the purpose of future development. This work is a proof-of-concept to answer whether this domain and its issues are representable and solvable using a graph database.

For these reasons, only a simplified version of dependency resolving has been implemented based on version matching. Its purpose is to limit the space of searchable dependencies for query search. More specific dependency traversal is up to the user and their query.

Although the full dependency resolution algorithm is already implemented in the apk-tools, unfortunately, it does not provide a sufficient application interface for the needs of this work. As a apart of future development, it would be advisable to consider how to extend the interface provided by apk-tools for these purposes appropriately.

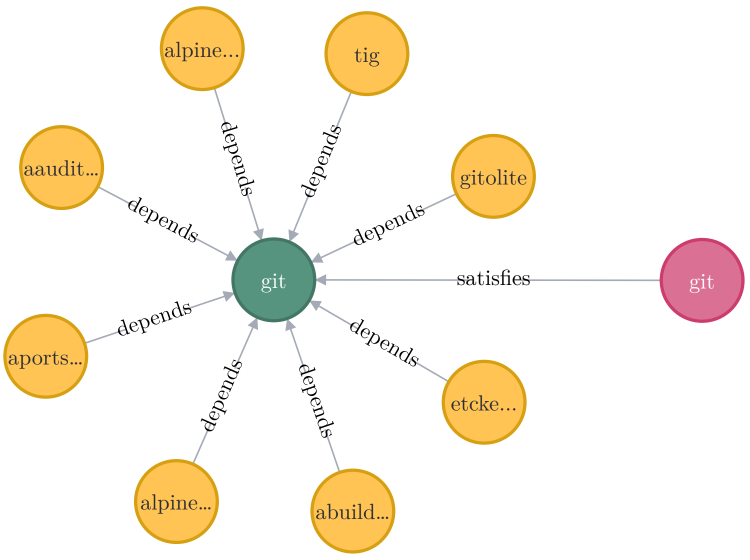


Figure 3.3: Dependency subgraph diagram from Neo4j browser. Circles represent nodes. Yellow – packages, green – dependency, pink – provider.

As already outlined in the conceptual model, in Section 2.2.1.4, dependencies are represented by the Dependency entity. Dependency can only be satisfied by the Provider entity. Whether a particular dependency is satisfied by a given provider is determined by the *satisfies* edge. Since dependency itself is not domain-dependent on any particular Package that requests it, it can be reused for multiple Packages. This sharing of the intermediate node saves significant computational resources and is a natural optimization. As shown in Figure 3.3, green git node is shared among multiple packages, although it had to be resolved only once.

For comparison, the aports-turbo tool only considers the name when working with dependencies, it does not compare versions.

3.4.8 Dependency tree traversal

One of the fundamental queries is the dependency tree traversal. In other words, to list all packages on which the selected package depends directly or transitively. If there are no cycles in the graph, it will be a subgraph of the tree type. It is also a query suitable for demonstrating the ability of the graph approach to data representation.

The representation of dependencies between packages is problematic in this case. It is not a direct connection but a path consisting of multiple intermediate nodes. Querying transitive dependencies is beyond the expressive power of the Cypher language in this scenario, as it is necessary to decide which edge to follow at each step of traversing the graph.

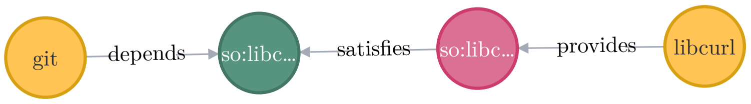


Figure 3.4: Dependency satisfaction between git and libcurl packages.

One solution is the Traversal framework mentioned in Section 3.3.1, which, however, adds further complexity to the project. There is an alternative solution that is more optimal for read operations, in line with *NR1* requirement. This solution involves maintaining an auxiliary structure in the graph, a derived edge called *vdepends*.

The relationships vdepends mentioned in Code 3.7 also reflect the domain restriction that packages in main repository can only depend on packages in the same repository. Packages in the community repository can depend on packages in both the main and community repositories.

With this modified graph, it is now easy to traverse it using common algorithms such as BFS. Example 3.8 returns the dependency tree for the package *git* and assigns the individual dependencies into levels according to BFS. Although Neo4j offers a feature called virtual graphs, these cannot be used for further traversal, only for visualization.

```
MATCH
    (pa:package)-->(a:arch),
    (pa)-->(ra:repository {name: "community"}),
    (pa)-[:depends]->(de:dependency),
    (de)<-[:satisfies]-(pr:provider),
    (pr)<-[:provides]-(pb:package),
    (pb)-->(a),
    (pb)-->(rb:repository)
WHERE
    ra.branch = rb.branch AND
    rb.name IN ["main", "community"]
WITH pa, pb
MERGE (pa)-[:vdepends]->(pb)
MERGE (pa)-[:vdepends_any]->(pb)
```

Code 3.7: Example of query for simplifying intermediate nodes by using an auxiliary relationship.

Technically, several types of these auxiliary relationships are used, depending on the type of dependency (*depends*, *makedepends*, etc.). It is also important to keep in mind that the *install_if* relationship reverses the direction of the dependency. If at least one of these edges exists, a relationship *vdepends_any* is created simultaneously for easier querying.

```
MATCH
    (p:package {pkgname: "git"})-->(a:arch {name: "x86_64"})
CALL apoc.path.spanningTree(p, {
    relationshipFilter: "vdepends_any>"
})
YIELD path
RETURN path, length(path) as hops
```

Code 3.8: Example of query for obtaining a dependency tree of the given package.

As already mentioned, this is not exact dependency resolution, but version matching only. A comparison with the actual solver in apk-tools has been made. The result of the actual dependency resolution performed by apk-tools is shown in Code 3.9. The result of version matching in the proposed tool is shown in Figure 3.5, generated by the query in Code 3.8.

As can be seen from the example, the proposed tool included more packages in the dependency tree. The reason for this is that all packages potentially satisfying the dependency are marked using version matching. Thus, it is not one specific dependency tree like the one generated by apk-tools, but rather a union of several dependency trees.

To be specific, in the discussed example, the proposed tool identified three possible ways to satisfy the `/bin/sh` dependency from ca-certificates package – either with package busybox, dash or yash. The apk-tools chose busybox from these options since it has the highest `provider_priority`.

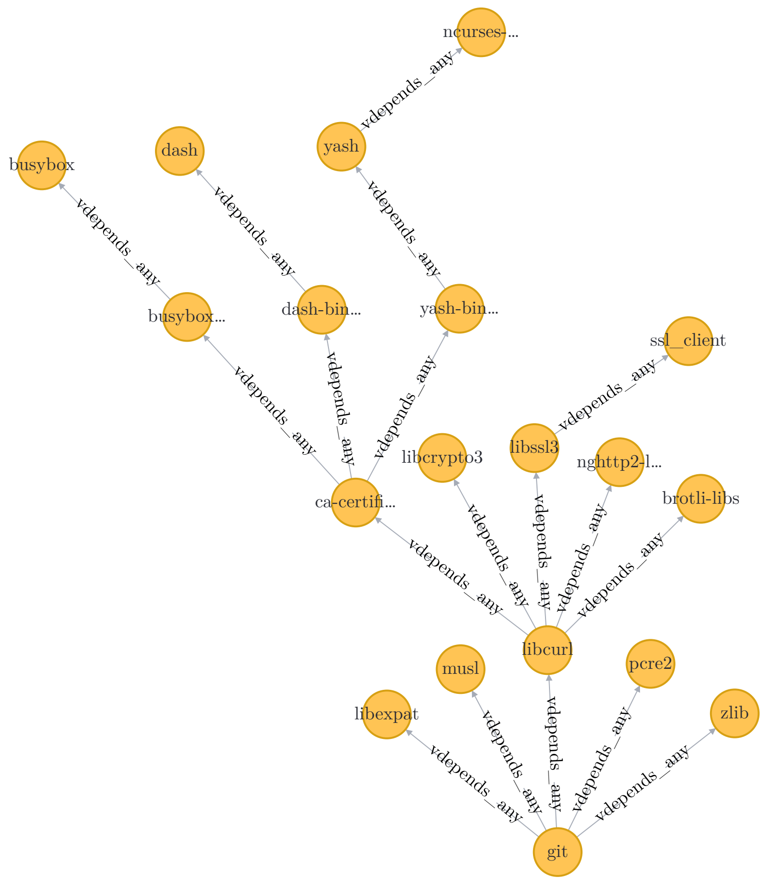


Figure 3.5: Dependency tree of git package version 2.38.5-r0 in v3.17 according to the proposed tool.

```
(1/14) Installing musl (1.2.3-r4)
(2/14) Installing busybox (1.35.0-r29)
(3/14) Installing busybox-binsh (1.35.0-r29)
(4/14) Installing libcrypto3 (3.0.8-r4)
(5/14) Installing ca-certificates (20220614-r4)
(6/14) Installing brotli-libs (1.0.9-r9)
(7/14) Installing nghttp2-libs (1.51.0-r0)
(8/14) Installing libssl3 (3.0.8-r4)
(9/14) Installing ssl_client (1.35.0-r29)
(10/14) Installing zlib (1.2.13-r0)
(11/14) Installing libcurl (8.0.1-r0)
(12/14) Installing libexpat (2.5.0-r0)
(13/14) Installing pcre2 (10.42-r0)
(14/14) Installing git (2.38.5-r0)
```

Code 3.9: Dependency tree of git package version 2.38.5-r0 in v3.17 according to apk-tools.

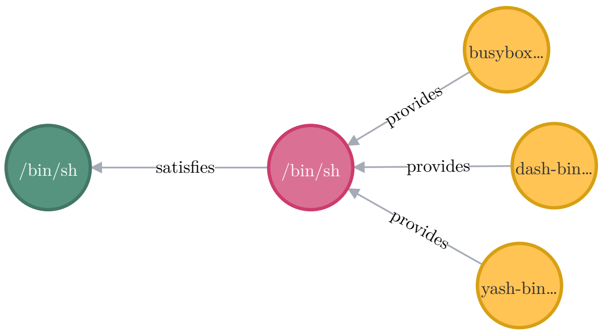


Figure 3.6: Dependency /bin/sh being satisfied by packages busybox, dash and yash.

Just to compare with the traditional relational approach, a query similar to Code 3.8 could be done in the SQL language using *recursive* queries. However, as illustrated in Code 3.10, such a query for a graph traversal is very complicated, both for humans to comprehend and for the DBMS to execute.

```sql
WITH RECURSIVE dependency AS (
    SELECT
        vdepends_any.parent_id AS parent_id,
        0 AS level
    FROM vdepends_any
        JOIN package
            AS parent ON vdepends_any.parent_id = parent.id
        JOIN package
            AS child ON vdepends_any.child_id = child.id
        JOIN arch ON parent.arch_id = arch.id
    WHERE child.pkgname = "git"
        AND child.arch_id = parent.arch_id
        AND arch.name = "x86_64"
    UNION ALL
    SELECT
        vdepends_any.parent_id,
        level + 1
    FROM dependency, vdepends_any
    WHERE dependency.parent_id = vdepends_any.child_id
)
SELECT * FROM dependency
```

Code 3.10: Traversing dependency tree using SQL recursive query.

3.5 Commands

The application offers following commands:

Bootstrap command creates the database in bulk. This command is expected to take a long time to complete, usually several hours.

Update command updates an already bootstrapped database with new changes in the repositories. This command is expected to take no more than a few minutes to complete and is usually run on a populated database. The implementation of this command fulfils functional requirement FR2, which requires the ability to track and incorporate incremental changes.

Rebuild command is used to generate dependency satisfaction and auxiliary vdepends relationships. It is automatically executed after bootstrap and update.

Both bootstrap and update modes share a large portion of the functionality but may differ in some cases, mainly for performance reasons. Proper import order of individual entities must be maintained during the execution of these commands. For example, dependency resolution, mentioned in Section 3.4.7, must be executed only after all Dependency and Provider entities are present in the database.

Running the tool requires a locally downloaded repository of aports and a local Alpine mirror.[25] In the case of update mode, the application synchronizes changes in the aports repository and Alpine mirror by itself. Based on the logged data and standard output, it processes affected packages. This command can be run automatically after connecting to a CRON-like service or a publish-subscribe protocol such as MQTT.

[25]It should be noted that the entire Alpine mirror takes up units of TB (according to Alpine Linux Development Team [16]).

Chapter 4

Evaluation

The application allows for the creating and updating of a graph database. The purpose of this chapter is to evaluate the established objectives. However, the main objective was to determine the new possibilities for querying and analyzing data that the proposed tool provides.

Firstly, this chapter compares the tool with the existing aports-turbo tool which was described in previous chapters. Secondly, representatives from the Alpine Linux community were approached to provide queries that would be beneficial for their work, but which they are currently unable to execute. Finally, this chapter addresses the actual measured size of the dataset.

It is worth noting that the process of developing the application alone was a significant contribution to the Alpine Linux community, as the author identified and reported several bugs in the APKBUILDs during development, resulting in a prompt fix.

The testing was conducted on a virtual machine[26] running Alpine Linux v3.17. The Neo4j DBMS version 5.4.0 was launched on the same machine in a container environment using the Podman tool[27].

During the testing of the requested queries, no performance issues were noted that would significantly limit the functionality of the proposed tool. The only observed performance degradation was encountered during the search for cycles with a length of four or more, referring to the use case CC3 in Appendix D.2.3. Worsened performance was also observed for queries using string operations. This problem is, however, solvable to some extent by using better indexing or improving the graph structure.

[26]Four CPUs, 16 GB of RAM, and SSD storage were available.

[27]The Neo4j package for Alpine Linux was not yet in existence at the time of the application development.

4.1 Comparison with aports-turbo

As stated in Section 1.2.3.1, it is based on a relational database and was therefore selected for comparison with the tool created in this work.

As a result of the evaluation, it was confirmed that the proposed tool allows the same set of queries to be performed as aports-turbo. Additionally, a comparison was made between existing SQL queries and the same queries written in Cypher language, which proved to be more natural and more straightforward for graph querying. Some examples of unusual queries are mentioned in Appendix C.

4.2 Community Questionnaire

During the evaluation, all 16 queries provided by the Alpine Linux community were successfully implemented and evaluated using the Cypher language. Few of them required features from the Neo4j APOC library. The detailed analysis of each query is presented in Appendix D. The developed tool meets all functional and non-functional requirements specified in Section 2.1.

4.3 Dataset size

This chapter deals with the evaluation of the total size of the dataset over which the database operates. The original estimates were expressed in Section 2.3. These were in the order of millions of nodes for a single branch.[28] In the non-functional requirements, there was a requirement for at least 4 million nodes. In fact, fewer nodes were needed to represent this single branch. This difference is mainly due to the fact that the original estimate was based on the *x86_64* architecture, for which the largest number of packages is available. But it is still units of millions of nodes:

- Nodes: 2,262,672
- Relationships: 7,652,959
- Total time of bootstrapping[29] approximately 12 hours.

For the following tables, only the *x86_64* and *aarch64* architectures were selected since they are containing the largest number of packages. All testing was done over branch v3.16, its raw data size is approximately 200 GB.

[28]Including both repositories and all architectures.

[29]Including parsing both APKBUILDS and resulting binary files.

Repository	Architecture	Nodes	Relationships
main	x86_64	384,707	686,370
main	aarch64	336,443	656,481
community	x86_64	1,745,538	2,270,696
community	aarch64	1,483,708	2,157,162

Table 4.1: Comparison of datasets for single repositories. Performed on May 5th 2023.

Table 4.2 shows how the size of dataset is affected by the combination of architectures, or repositories, within a single database. This way leads to more efficient usage since a significant part of the nodes is being shared. This applies, for example, to the files and dependencies nodes.

Repository	Architecture	Nodes	Relationships
main	x86_64 ∪ aarch64	394,312	724,710
comm.	x86_64 ∪ aarch64	1,762,218	2,918,577
main ∪ comm.	x86_64 ∪ aarch64	2,135,174	3,819,912

Table 4.2: Comparison of datasets for combined architectures or repositories. Performed on May 5th 2023.

Table 4.3 shows the number of shared file nodes across architectures and within the repository. Total files represent all non-unique files identified by its path.

Repository	Architecture	Total files	File nodes
main	x86_64	342,248	294,875
main	aarch64	390,823	343,270
main	x86_64 ∪ aarch64	733,071	352,661
comm.	x86_64	1,612,817	1,416,280
comm.	aarch64	1,828,106	1,646,219
comm.	x86_64 ∪ aarch64	3,440,923	1,686,705
main ∪ comm.	x86_64 ∪ aarch64	4,173,994	2,037,781

Table 4.3: Comparison of shared nodes for combined architectures. Performed on May 5th 2023.

Conclusion

The main goal of this thesis was to provide the Alpine Linux community with a tool for querying information about packages, which would allow for their in-depth analysis. The thesis aimed to explore solutions to this problem using graph theory and graph databases.

Initially, this thesis described the problem of package management in Alpine Linux. The chosen domain was converted into a structure called property graph with the help of graph theory.

Several graph DBMSs were compared based on the gathered requirements, created graph model, and the analysis as mentioned earlier. Eventually, Neo4j was identified as the most suitable solution for this purpose.

Using the modelled property graph and the Neo4j graph DBMS, a tool that converts package metadata and their relationships into a graph database was created. The Node.js runtime environment and the TypeScript language were chosen for the implementation of the application logic. The resulting solution is a proof-of-concept that meets all functional and non-functional requirements.

Compared to the relational approach, this method of representing package data has proven to be both viable and simple. In particular, it has proved to be more straightforward in terms of data modelling, storing and in terms of querying.

The proposed data representation allows the domain to be queried and analysed in ways that were previously impossible. The chosen graph DBMS meets the performance requirements and does not suffer from the limitations of the described solutions based on a relational database. The only significant performance degradation occurred when searching for cycles of length four or more.

Regarding the practical application of the results of this work, the new solution has the potential to completely replace the current application aports-turbo, if an appropriate API and web interface are developed. However, it would be necessary to consider a high number of concurrent users and to address the potential scalability issues that may arise.

During the problem solving process within this domain, several areas were identified as having potential for further development. These include integrating with CVE security trackers, implementing a full dependency solver within the Neo4j DBMS, or collecting additional metadata such as from git logs or files. In addition, data retention issues need to be addressed as the database continues to grow with new data.

Bibliography

1. PREINING, Norbert. *Analysing Debian packages with Neo4j - Part 1 - Debian* [online]. 2018. [visited on 2023-05-06]. Available from: `https://www.preining.info/blog/2018/04/analysing-debian-packages-with-neo4j-part-1-debian/`.
2. DEBIAN. *What is a package manager?* [online]. 2023. [visited on 2023-04-14]. Available from: `https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html`.
3. WIKIPEDIA CONTRIBUTORS. *List of software package management systems* [online]. 2023. [visited on 2023-04-16]. Available from: `https://en.wikipedia.org/w/index.php`.
4. DECAN, Alexandre; MENS, Tom; CONSTANTINOU, Eleni. On the impact of security vulnerabilities in the npm package dependency network. In: 2018. Available from DOI: `10.1145/3196398.3196401`.
5. WIKIQUOTE. *Brian Kernighan* [online]. 2023. [visited on 2023-05-06]. Available from: `https://en.wikiquote.org/w/index.php?title=Brian_Kernighan&oldid=3273358`.
6. ALPINE LINUX DEVELOPMENT TEAM. *Working with the Alpine Package Keeper* [online]. 2019. [visited on 2023-04-15]. Available from: `https://docs.alpinelinux.org/user-handbook/0.1a/Working/apk.html`.
7. ALPINE LINUX DEVELOPMENT TEAM. *Alpine Package Keeper* [online]. 2022. [visited on 2023-04-14]. Available from: `https://wiki.alpinelinux.org/wiki/Alpine_Package_Keeper`.
8. ALPINE LINUX DEVELOPMENT TEAM. *About Alpine Linux* [online]. www.alpinelinux.org, 2022. [visited on 2023-04-17]. Available from: `https://www.alpinelinux.org/about/`.

9. ALPINE LINUX DEVELOPMENT TEAM. *Alpine Linux Trivia* [online]. www.alpinelinux.org, 2022. [visited on 2023-04-17]. Available from: `https://wiki.alpinelinux.org/wiki/Alpine_Linux:Trivia`.
10. COPA, Natanael. *Re: [leaf-devel] 2.6.x kernel support?* [online]. sourceforge.net, 2005. [visited on 2023-04-19]. Available from: `https://sourceforge.net/p/leaf/mailman/message/12731159/`.
11. ALPINE LINUX DEVELOPMENT TEAM. *Repositories* [online]. www.alpinelinux.org, 2023. [visited on 2023-04-17]. Available from: `https://wiki.alpinelinux.org/wiki/Repositories`.
12. ALPINE LINUX DEVELOPMENT TEAM. *Creating an Alpine package* [online]. 2023. [visited on 2023-04-20]. Available from: `https://wiki.alpinelinux.org/wiki/Creating_an_Alpine_package`.
13. ALPINE LINUX DEVELOPMENT TEAM. *APKBUILD Reference* [online]. 2023. [visited on 2023-04-20]. Available from: `https://wiki.alpinelinux.org/wiki/APKBUILD_Reference`.
14. ALPINE LINUX DEVELOPMENT TEAM. *Apk spec* [online]. 2022. [visited on 2023-04-14]. Available from: `https://wiki.alpinelinux.org/wiki/Apk_spec`.
15. CONILL, Ariadne. *Spelunking through the apk-tools dependency solver* [online]. ariadne.space, 2021. [visited on 2023-04-29]. Available from: `https://ariadne.space/2021/10/31/spelunking-through-the-apk-tools-dependency-solver/`.
16. ALPINE LINUX DEVELOPMENT TEAM. *How to setup a Alpine Linux mirror* [online]. www.alpinelinux.org, 2023. [visited on 2023-04-19]. Available from: `https://wiki.alpinelinux.org/wiki/How_to_setup_a_Alpine_Linux_mirror`.
17. ALPINE LINUX DEVELOPMENT TEAM. *aports-turbo* [online]. [N.d.]. [visited on 2023-04-26]. Available from: `https://gitlab.alpinelinux.org/alpine/infra/aports-turbo`.
18. RED HAT, INC. AND OTHERS. *Anitya* [online]. release-monitoring.org, 2022. [visited on 2023-05-09]. Available from: `https://release-monitoring.org/`.
19. POSTMARKETOS.ORG CONTRIBUTORS. *postmarketOS - apkbrowser* [online]. [N.d.]. [visited on 2023-04-26]. Available from: `https://gitlab.com/postmarketOS/apkbrowser`.
20. ALPINE LINUX DEVELOPMENT TEAM. *secdb* [online]. [N.d.]. [visited on 2023-04-26]. Available from: `https://gitlab.alpinelinux.org/alpine/security/secdb`.

21. ALPINE LINUX DEVELOPMENT TEAM. *Security Issue Tracker* [online]. [N.d.]. [visited on 2023-04-26]. Available from: `https://security.alpinelinux.org`.
22. ALPINE LINUX DEVELOPMENT TEAM. *apk-tools* [online]. [N.d.]. [visited on 2023-04-26]. Available from: `https://gitlab.alpinelinux.org/alpine/apk-tools`.
23. ALPINE LINUX DEVELOPMENT TEAM. *abuild* [online]. [N.d.]. [visited on 2023-04-26]. Available from: `https://gitlab.alpinelinux.org/alpine/abuild`.
24. ALPINE LINUX DEVELOPMENT TEAM. *alpine* [online]. [N.d.]. [visited on 2023-04-26]. Available from: `https://gitlab.alpinelinux.org/alpine`.
25. BORŮVKA, Otakar. Příspěvek k otázce ekonomické stavby elektrovodných sítí. *Elektrotechnický obzor 15*. 1925.
26. DIESTEL, Reinhard. *Graph Theory*. 5th. Prentice Hall, 2017. ISBN 9783662536216.
27. BANG-JENSEN, Jørgen; GUTIN, Gregory. *Digraphs: theory, algorithms and applications*. Springer-Verlag, 2007.
28. ANGLES, Renzo; ARENAS, Marcelo; BARCELÓ, Pablo; HOGAN, Aidan; REUTTER, Juan; VRGOČ, Domagoj. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys*. 2017, vol. 50, no. 5. Available from DOI: `10.1145/3104031`.
29. LEISERSON, Charles E.; CORMEN, Thomas H.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms*. 3rd. The MIT Press, 2009. ISBN 9780262033848.
30. RAMAKRISHNAN, Raghu; GEHRKE, Johannes. *Database Management Systems*. 3rd. McGraw-Hill, 2003. ISBN 9780072465631.
31. POKORNÝ, Jaroslav; VALENTA, Michal. *Databázové systémy*. 2. přepracované vydání. Česká technika - Nakladatelství ČVUT, 2020. ISBN 9788001066966.
32. LITTLE, Mark. Transactions and Web services. *Communications of the ACM*. 2003, vol. 46, no. 10, pp. 49–54. Available from DOI: `10.1145/944217.944237`.
33. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. *Graph Databases*. 2nd. O'Reilly Media, Inc., 2015. ISBN 9781491932001.
34. MCCREARY, Dan. *The Neighborhood Walk Story* [online]. 2018. [visited on 2023-04-14]. Available from: `https://dmccreary.medium.com/how-to-explain-index-free-adjacency-to-your-manager-1a8e68ec664a`.

35. POKORNÝ, Jaroslav. Graph Databases: Their Power and Limitations. In: SAEED, Khalid; HOMENDA, Wladyslaw (eds.). *Computer Information Systems and Industrial Management*. Springer-Verlag, 2015, pp. 58–69. ISBN 978-3-319-24369-6. Available from DOI: `10.1007/978-3-319-24369-6_5`.

36. VILHENA, Thomas. *Index-free adjacency* [online]. 2019. [visited on 2023-04-14]. Available from: `https://thomasvilhena.com/2019/08/index-free-adjacency`.

37. W3C. *RDF 1.1 Concepts and Abstract Syntax* [online]. 2014. [visited on 2023-04-15]. Available from: `https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`.

38. DEUTSCH, Alin; FRANCIS, Nadime; GREEN, Alastair; HARE, Keith; LI, Bei; LIBKIN, Leonid; LINDAAKER, Tobias; MARSAULT, Victor; MARTENS, Wim; MICHELS, Jan; MURLAK, Filip; PLANTIKOW, Stefan; SELMER, Petra; REST, Oskar van; VOIGT, Hannes; VRGOČ, Domagoj; WU, Mingxi; ZEMKE, Fred. Graph Pattern Matching in GQL and SQL/PGQ. *Proceedings of the 2022 International Conference on Management of Data*. 2022. Available from DOI: `10.1145/3514221.3526057`.

39. SOLIDIT CONSULTING & SOFTWARE DEVELOPMENT GMBH. *DB-Engines Ranking* [online]. 2023. [visited on 2023-04-15]. Available from: `https://db-engines.com/en/ranking/graph+dbms`.

40. BITNINE GLOBAL INC. *Graph DBMS Performance Comparison AgensGraph vs. Neo4j* [online]. bitnine.net, 2017. [visited on 2023-04-15]. Available from: `https://bitnine.net/blog-agens-solution/blog-agensgraph/graph-dbms-performance-comparison-agensgraph-vs-neo4j/`.

41. WANG, Ran; YANG, Zhengyi; ZHANG, Wenjie; LIN, Xuemin. An Empirical Study on Recent Graph Database Systems. *Knowledge Science, Engineering and Management* [online]. 2020, vol. 12274, pp. 328–340 [visited on 2023-04-15]. Available from DOI: `10.1007/978-3-030-55130-8_29`.

42. ALASTAIR, Green. *GQL Is Now a Global Standards Project alongside SQL* [online]. Neo4j Graph Data Platform, 2019. [visited on 2023-04-15]. Available from: `https://neo4j.com/blog/gql-standard-query-language-property-graphs/`.

43. ASSOCIATION OF ISO GRAPH QUERY LANGUAGE PROPONENTS. *Graph Query Language GQL - What is a GQL Standard?* [online]. JCC Consulting, Inc., www.gqlstandards.org, 2022 [visited on 2023-04-16]. Available from: `https://www.gqlstandards.org/what-is-a-gql-standard`.

44. ALPINE LINUX DEVELOPMENT TEAM. *Aports tree* [online]. 2022. [visited on 2023-04-20]. Available from: `https://wiki.alpinelinux.org/wiki/Aports_tree`.

45. CONRADIE, Willem; GORANKO, Valentin. *Logic and Discrete Mathematics*. John Wiley & Sons, 2015. ISBN 9781119000099.

46. MARTIN, Robert C. *Clean Coder Blog* [online]. Cleancoder.com, 2019. [visited on 2023-04-21]. Available from: `https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html`.

47. MARTIN, Robert C. *The Open-Closed Principle* [online]. 2006. [visited on 2023-04-20]. Available from: `https://web.archive.org/web/20150905081105/http://www.objectmentor.com/resources/articles/ocp.pdf`.

48. OBJECT MENTOR, INC. *The Interface Segregation Principle* [online]. 1996. [visited on 2023-04-24]. Available from: `https://web.archive.org/web/20150905081110/http://www.objectmentor.com/resources/articles/isp.pdf`.

49. FOWLER, Martin. *Patterns of enterprise application architecture*. Addison-Wesley, 2015.

50. NEO4J, INC. *Traversal Framework - Java Reference* [online]. Neo4j Graph Data Platform, 2023. [visited on 2023-05-03]. Available from: `https://neo4j.com/docs/java-reference/current/traversal-framework/`.

51. MICROSOFT. *TypeScript - JavaScript that scales.* [online]. Typescriptlang.org, 2015. [visited on 2023-04-24]. Available from: `https://www.typescriptlang.org/`.

Appendix A

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
APOC	Awesome Procedures On Cypher (Neo4j library)
APT	Advanced Package Tool
AQL	ArangoDB Query Language
BFS	Breadth-first search
BSD	Berkeley Software Distribution
CLI	Commands Line Interface
CPU	Central Processing Unit
CTU	Czech Technical University in Prague
CVE	Common Vulnerabilities and Exposures
DBMS	Database Management System
DFS	Depth-first search
DTO	Data Transfer Object
ELF	Executable and Linkable Format
GDBMS	Graph Database Management System
GoF	Gang of Four (set of design patterns)
GQL	Graph Query Language

FIT Faculty of Information Technology

IEC International Electrotechnical Commission

ISO International Organization for Standardization

JSON JavaScript Object Notation

LDBC The Linked Data Benchmark Council

NVD National Vulnerability Database

OGM Object-Graph Mapping

ORM Object-Relational Mapping

RAM Random-access memory

RDF Resource Description Framework

SSD Solid-state Drive

SQL Structured Query Language

TAR Tape Archive (file format)

UC Use Case

YAML YAML Ain't Markup Language (markup language)

Appendix **B**

Attachment Contents

```
 README.adoc ........................................... brief description
 paper
   BP_Meinlschmidt_Jakub_2023..... source code of this thesis in LaTeX
   BP_Meinlschmidt_Jakub_2023.pdf . text of this thesis in PDF format
 project ............................. source code of the practical part
   README.adoc ................. brief description of running the project
```

Appendix C

Queries for aports-turbo

The queries runtime was tested over the entire branch v3.16, i.e. main and community repositories and all architectures.

C.1 Search

The tool allows searching for specific packages by repository, architecture and name.

```
SELECT
    packages.*,
    datetime(packages.build_time, 'unixepoch') as build_time,
    maintainer.name as mname, maintainer.email as memail
FROM packages
LEFT JOIN maintainer ON packages.maintainer = maintainer.id
WHERE
    packages.repo = :repo AND
    packages.arch = :arch AND
    packages.name = :pkgname
```

Code C.1: Query source code for package search in SQL [17].

```
MATCH
    (p:package {pkgname: $pkgname}),
    (p)-->(r:repository {name: $repository_name}),
    (p)-->(a:arch {name: $arch})
OPTIONAL MATCH (m:person)-[:maintains]->(p)
RETURN p, m.email, m.name
```

Code C.2: Query code for package search in Cypher.

This query took approximately 1 ms to complete.

C.2 Dependencies

The tool allows searching for packages by dependencies.

```
SELECT DISTINCT
    pa.repo,
    pa.arch,
    pa.name,
    MAX(pa.provider_priority)
FROM depends de
LEFT JOIN provides pr ON de.name = pr.name
LEFT JOIN packages pa ON pr.pid = pa.id
WHERE pa.arch = :arch AND de.pid = :id
GROUP BY pr.name
ORDER BY pa.name
```

Code C.3: Query source code for dependencies search [17].

The difference is that in the proposed tool the primary key ID is not used, but `pkgname` and `ver_constraint` is used to identify the dependency.

```
MATCH (de:dependency)
OPTIONAL MATCH
    (pa:package)-->(de),
    (pr:provider)-->(de),
    (pa)-->(r:repository),
    (pa)-->(a:arch)
WHERE pa.arch = $arch AND de.pkgname = $pkgname
RETURN
    DISTINCT r.name,
    a.name,
    pa.name,
    MAX(pa.provider_priority)
```

Code C.4: Query source code for dependencies search in Cypher.

This query took approximately 50 ms to complete.

Appendix D

Queries Evaluation

Explanation for types of dependencies:

R depends (runtime)

B makedepends (buildtime)

C checkdepends (for tests)

I install_if

The query runtime was tested over the entire branch v3.16, i.e. main and community repositories and all architectures.

D.1 Queries for Package Maintainers

D.1.1 Use Case PM1

Description: *Find aports maintained by "Jakub Jirutka" that depend on "cargo".*

```
MATCH
    (a:apkbuild)<-[:maintains]-(pe:person),
    (a)-->(d:dependency {pkgname: "cargo"})
WHERE pe.name = "Jakub Jirutka"
RETURN a
```

This query took approximately 5 ms to complete.

D.1.2 Use Case PM2

Description: *Find aports with "ppc64le" arch enabled that depend (RBC) on "luajit" package, directly and transitively.*

```
MATCH
    (a:apkbuild)-[:vdepends|vmakedepends|vcheckdepends*]->
        (d:package),
    (a)-[:built_for]->(c:arch)
WHERE
    d.pkgname = "luajit" AND
    c.name = "ppc64le"
RETURN a
```

This query took approximately 10 ms to complete.

D.1.3 Use Case PM3

Description: *Find aports that transitively depend (RBC) on "cargo" or "rust" and are disabled on "riscv64".*

```
MATCH
    (a:apkbuild)-[:vdepends|vmakedepends|vcheckdepends*1..]->
        (d:package)
WHERE
    d.pkgname IN ["cargo", "rust"] AND
    NOT (a)-[:built_for]->(:arch {name: "riscv64"})
RETURN a
```

This query took approximately 20 ms to complete.

D.1.4 Use Case PM4

Description: *Find aports that depend (B) on "cargo" and "zstd-dev".*

```
MATCH
    (a:apkbuild),
    (a)-[:vmakedepends]->(:package {pkgname: "cargo"}),
    (a)-[:vmakedepends]->(:package {pkgname: "zstd-dev"})
RETURN a
```

This query took approximately 1 ms to complete.

D.1.5 Use Case PM5

Description: *Find aports that depend (B) on “ruby-dev” or depend (R) on “ruby” or “so:libruby.so.3.1” or installs any file with the given path.*

Path: */usr/lib/ruby/gems/3.1.0/**

```
MATCH (a:apkbuild)-[:depends]->(d:dependency)
WHERE
    d.pkgname IN ["ruby-dev", "ruby", "so:libruby.so.3.1"]
RETURN a
UNION
MATCH (a:apkbuild)-[:builds]->(p:package)
WHERE (p)-[:file_rel]->
    (:file {path: "/usr/lib/ruby/gems/3.1.0/"})
RETURN a
```

This query took approximately 2 s to complete.

D.1.6 Use Case PM6

Description: *Find packages with at least one file with the “setuid” bit set.*

```
MATCH (p:package)-[r:file_rel]->(f:file)
WITH p, count(f) as files, r, left(r.mode, 1) as mode
WHERE
    files > 0 AND
    mode <> "0"
WITH apoc.bitwise.op(toInteger(mode), "&", 4) as setuid, r, p
WHERE
    setuid = 4
RETURN p.pkgname, r.mode, setuid
```

This query took approximately 11 s to complete.

D.1.7 Use Case PM7

Description: *Find packages with at least one file with file capabilities in "xattrs".*

```
MATCH (p:package)-[r:file_rel]->(f:file)
WITH p, count(f) as files, r
WHERE
    files > 0 AND
    NOT r.xattrs IS NULL
RETURN p.pkgname
```

This query took approximately 8 s to complete.

D.1.8 Use Case PM8

Description: *Find packages whose name ends with "-doc" and that do not contain any file with path "/usr/share/man/*".*

The following approach is ineffective[30], since it creates a cartesian product of packages and files:

```
MATCH (p:package), (f:file)
WHERE
    p.pkgname ENDS WITH "-doc" AND
    f.path STARTS WITH "/usr/share/man/" AND
    NOT (p)-[:file_rel]->(f)
RETURN p
```

More efficient approach can be used:

```
MATCH (p:package)
WHERE
    p.pkgname ENDS WITH "-doc" AND
    NOT (p)-[:file_rel]->(:file {path: "/usr/share/man/"})
RETURN p.pkgname
```

This query took approximately 5 s to complete.

[30]Takes more than minutes to complete.

D.1.9 Use Case PM9

Description: *Find packages whose name ends with "-doc", contain at least one file outside of the "/usr/share/man/" directory, and the size of the installed files is over 256 kiB.*

```
MATCH (p:package)-[r:file_rel]->(f:file)
WITH p, count(f) as man_files, r, f
WHERE
    p.pkgname ENDS WITH "-doc" AND
    NOT f.path STARTS WITH "/usr/share/man/" AND
    // Ends with `/` <=> is a directory
    NOT f.path ENDS WITH "/" AND
    man_files > 0
WITH p, sum(r.size) as total_size
WHERE total_size > 256*1024
RETURN p.pkgname, total_size
```

This query took approximately 13 s to complete.

D.2 Queries for Consistency Checks

D.2.1 Use Case CC1

Description: *Find packages for "x86_64" with conflicting files (i.e. two packages provide the same path that is not a directory) that do not declare the other package in "replaces".*

```
MATCH
    (a:arch {name: "x86_64"}),
    (a)<--(pa:package)-[:file_rel]->
        (f:file)<-[:file_rel]-(pb:package)-->(a)
WHERE
     // Ends with `/` <=> is a directory
    NOT f.path ENDS WITH "/" AND
    NOT (pa)-[:replaces]->(:dependency)
        <-[:satisfies]-(:provider)<--(pb)
RETURN pa.pkgname, pb.pkgname, f.path
```

This query took approximately 200 ms to complete.

D.2.2 Use Case CC2

Description: *Find aports and packages with unsatisfied dependencies (RBCI).*

```
MATCH
    (a:apkbuild|package)-->(d:dependency)
WHERE NOT (d)<-[:satisfies]-(:provider)
RETURN a
```

This query took approximately 20 ms to complete.

D.2.3 Use Case CC3

Description: *Find aports with cyclic dependencies (RBC).*

```
MATCH
    path=(p:package)-[:vdepends|vmakedepends|vcheckdepends*]
        ->(p)
RETURN path
```

The time for the query to complete varied depending on the length of the searched path:

- Path of length 2 took 900 ms
- Path of length 3 took 4 s
- Path of length 4 took 13 s
- Path of length 5 took 42 s
- Path of length 6 took 121 s
- Path of length 6 took 307 s

D.2.4 Use Case CC4

Description: *Select two APKBUILDs of the same name in a single branch but in two repositories.*

```
MATCH
    (a:apkbuild),
    (a)-[:in]->(ra:repository)
WITH
    a.pkgname as aport,
    ra.branch as branch,
    count(ra.name) as repositoriesInSingleBranch
WHERE repositoriesInSingleBranch > 1
RETURN aport
```

This query took approximately 50 ms to complete.

D.3 Queries for Users

D.3.1 Use Case U1

Description: *Find packages for architecture "x86_64" that provide file "ioctl.h".*

```
MATCH
    (p:package),
    (p)-[:built_for]->(:arch {name: "x86_64"}),
    (p)-[:file_rel]->(f:file)
WHERE f.path ENDS WITH "/ioctl.h"
RETURN p
```

This query took approximately 800 ms to complete.

D.3.2 Use Case U2

Description: *Find out in which version of packages (aports) "CVE-2022-37434" was fixed.*

```
MATCH (c:cve {id: "CVE-2022-37434"})-->(a:apkbuild)
RETURN a
```

This query took approximately 1 ms to complete.

D.3.3 Use Case U3

Description: *Find packages for architecture "x86_64" whose pkgname contains "postgresql".*

```
MATCH (:arch {name: "x86_64"})<--(p:package)
WHERE p.pkgname CONTAINS "postgresql"
RETURN p
```

This query took approximately 20 ms to complete.