**VI. Introduction**

As a software engineer, it is very important to choose the right libraries for our projects. Picking the wrong libraries lead to expensive consequences since we need to invest time into learning how to use these libraries and of course, picking the right library for a certain programming task is not a trivial task. There are many things for us to consider when deciding on a library. One important characteristic of a library is its popularity. Using a popular library is more than just simply following the trend. A popular library means better community support such as bug reporting, feature implementation. A popular library has more learning materials, more people that we can go to and ask questions... Being able to see what libraries are trending and are used by other people will help us better decide what to use for our project.

**How it works into CS**

Dependency management systems

Software projects rarely work in isolation. In most cases, a project relies on reusable functionality in the form of libraries or is broken up into individual components to compose a modularized system. Dependency management is a technique for declaring, resolving and using dependencies required by the project in an automated fashion.

Software projects rarely work alone. Developing software applications are time and money consuming and most projects don't have the resources to invest in the time and money to build all the parts of the system from scratch. Because of this, modern software applications often depend on existing third-party libraries to provide the functionalities that they need.

Even though using third-party libraries provide us many benefits, we still have to be able to manage these libraries and also the libraries that these libraries depend on. Managing a complex and interdependent arrangement is a very difficult task. The problem comes in several forms such as: too many dependencies, long chains of dependencies or where dependencies conflicting with each other. Because of this, most of the complex software uses some kind of package management tools.

A software/package management system is a system that automates the process of installing, configuring and removing of third-party dependencies in a project in a reliable manner. These systems usually keep a database of software libraries and their metadata such as their name, description and version number... In a project that uses these tools, its dependencies information is usually kept in a file.

Graph database

Since our application involves exploring many-to-many relationships, graph data structure is particularly fundamental for my project. By putting these projects in a graph, I was able to compute... (E.g. The most popular libraries, how a library's usage has been changing overtime, how many common users between libraries...) Traditional relational database can simulate graph traversing operation by using foreign-keys and self-joins but the query performance can grow significantly as we increase the records and also it is hard and unintuitive to express these SQL queries.

So what is a graph database?

According to Neo4j website, a graph database is a database designed to treat the relationships between data as a first-class citizen in the data model. By treating *edges* in graph as first-class citizen like *vertices* allows these graph engines to traverse the network of nodes quickly and flexibly allow us to develop our application with less time and effort.

Version control system

*A version control system is a system that saves the modifications to the files that you make over time so that you can get back a specific version later. Git is currently the most popular version control system today. It can be now considered as the standard for version control.*

*So, Git is a version control system (VCS), but what does that mean?*

When software engineers make their software, they make modification to their code continuously, releasing newer versions of their applications.

Using VCSs help developers keep track of these revisions. They are stored in a repository that allows developers to access different versions, make modifications and apply these modifications to the code making a new version. The new changes are available for everyone in the team; they can download them and make contribution to the code.

Git is currently the most popular version control system today and it can be now considered as the standard for version control.

According to GIthub's wikipedia page, "GitHub is a **web-based** Git repository **hosting service**, which offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features"

**VII. Background**

Our project is heavily inspired by the work of OverOps blog or blog.takipi.com. They have been conducting studies on the most popular Java libraries by scraping Github data since 2013. By looking at the pom.xml files of projects that use Maven dependency management system, they generated a large number of data points. Analysing these data points give us helped them rank these libraries by the number of Github projects they appeared in.

Despite the usefulness of these studies, they were just snapshots at the time of conducting the studies. With their approach, our project will allow users to indentify upward/downward trends in usage and how these libraries are associated with each other.

There are existing repositories of software management system like Packagist or search.maven.org where users can get the dependencies of a project. We are different from these websites since we allow users to queries usages of libraries over time, ranking libraries based on their usage, for a given libraries, users can get other libraries that have the most common users with them.... With the same data we get from the repositories, we were able to provide these functionalities by implementing graph data model where projects are stored as nodes of a graph in our database.

Arangodb - graph database

Traditional relational database can be used to represent graph data structure and they are used by many of the big company for example Facebook. But implementing a graph with relational databases has many disadvantages:

* Using foreign-keys and self-join can stimulate graph traversing operation but as the data grows, the latency of these queries raise in a hard to predict manner and along with the memory usage.
* It is also unintuitive and hard to write SQL queries that perform these kinds of path operations.

Using a graph database would help us eliminate these problems since they are faster than relational databases when dealing with data that are connected in many-to-many relationship, modelling and writing data queries are much easier allowing faster and more pleasant development.

There were many available options for a graph database. Neo4j and Arangodb were the two that I've considered. Even though Neo4j is most popular graph database, I've chosen Arangodb for a number of reasons. Arangodb supports storing data on edge documents which is useful for our project. Another reason is that Neo4j is a graph only database while Arangodb is a multi-model database that would fit into other use cases.

Arangodb, like other graph databases, has its own query language called Arangodb Query Language (AQL). Like SQL for relational databases, AQL supports modifying and reading data. It allows us to perform complex graph operation to make use of our data.

Out application interact with Arangodb through the Python client which has a AQLquery() method which sends AQL queries to the Arangodb and get back the result.

Composer - package management system for PHP

PHP applications have been maturing into significantly more and more complex applications than ever before. There are increasingly more frameworks and third-party libraries available and use widely among the developers. The benefits of using third-party tools are undeniable since they reduce the time and cost of building software greatly. These perks of course come with a cost: We have to be able to manage these libraries and also the libraries that these libraries depend on. This is where a dependency management tool like Composer is needed for managing such a complex and interdependent arrangement. For a project that uses Composer, their dependencies are saved in a json file named - composer.json. By inspecting these files, we were able to acquire the data needed to for our graph that connects different PHP projects.

|  |
| --- |
| {  "name": "laravel/laravel",  "description": "The Laravel Framework.",  "keywords": ["framework", "laravel"],  "license": "MIT",  "type": "project",  "require": {  "php": "^7.1.3",  "fideloper/proxy": "^4.0",  "laravel/framework": "5.6.\*",  "laravel/tinker": "^1.0"  },  "require-dev": {  "filp/whoops": "^2.0",  "fzaninotto/faker": "^1.4",  "mockery/mockery": "^1.0",  "nunomaduro/collision": "^2.0",  "phpunit/phpunit": "^7.0"  }, } |

An example of a composer.json ’s content. We only use the “name”, “description” for the library’s data. The “require” and “require-dev” fields are the dependencies of the project which will be used to expand out graph.

**VII. Preparation**

1.

* Architecture diagrams

2.

* Programming language learnt: Python, JavaScript.
* Better understanding of NoSql and graph database. (Arangodb)
* Frameworks learnt: React, Flask

3. We need a graph database for our application where we can connect the libraries. Arangodb was chosen instead of Neo4j because it supports saving data on edge nodes. On the other hand Arangodb is a multi-model and it has arguably better performance than Neo4j.

4. Python:

* Using Python for automation tasks are usually better than PHP. We chose Python over PHP for fetching data.
* Since a lot of the logic is handled by Arangodb's AQL, we only needed a simple json service to call the aql queries. I've decided to use Python for the web service. Flask is the chosen micro framework for the job. It supports routing and jsontify which is all we need from our web service.

5. Front end:

* React is a trending and interesting framework so I've decided to use React
* Simple-react-state: I wanted a state management library where I can have a global state for the whole application. Something like Redux would be overkill for my application.

6. Separation of concern:

* Separation of concern is a trending architecture in the world of web application. Where the backend acts as a Json service that provides the data for the frontend (E.g headless cms).
* The architecture allows me to be flexible with my choices of backend and frontend frameworks.
* In the future, having a json service would allow me to build mobile and desktop apps for my application

**VII. Deliverable**

Web application, arangodb database, Backend

1. Web app

* Allow users to search for libraries with their name, query the most popular libraries at given time, see the usage of libraries overtime, see the libraries that are commonly used with a library.

2. Arangodb database

* Graph of libraries.
* Aql queries that perform many complex graph traversing

3. Back end

* Serve json for the webapp
* whoosh library for indexing the document
* Perform frequently update and data fetching to expand the graph
* Flask cache: file system cache instead of memcached ( does not have to connect to memcached over network, simple to implement and cover all the needs)

**VIII. Evaluation**

Unit tests from CS3270 coursework

* The front end meets most of the specified objectives but it is still relatively simple and does not have many functionality
* We couldn't use Celery in the data fetching process
* We added a cache layer for our project
* Used packagist together with github get projects's data

**REFERENCES**

**APENDIX 2**

**Set up:**

* Install composer, Arangodb
* Cherrypy server or Rocket for Flask deployment