Graph Time

A proof-of-concept data engineering project

Petter Lovehagen

2024-08-29

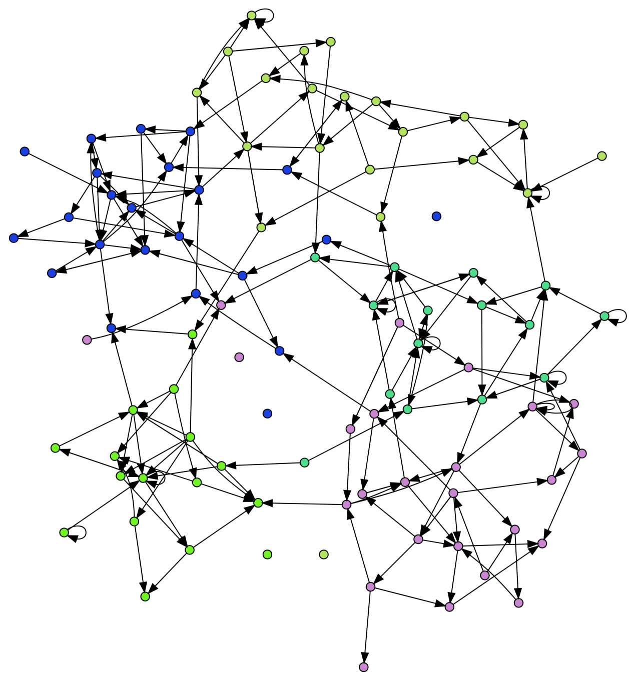
Table of contents

1 Exploring Graph Data Models for Timetabling Insights

A proof-of-concept data engineering project

Supervisor: Xiaodong Li

Programme: MSc Data Science



This is a randomly generated graph for visual purposes only.

See Appendix for graph generator code

2 Introduction

University timetabling is the complex process of scheduling curriculum elements (modules, programmes), rooms, and resources within an academic institution. At its simplest, it involves collecting and combining time slots, rooms, students and other resources while satisfying a multitude of constraints and preferences to achieve a *viable* outcome.

2.1 What is University Timetabling?

But it is never as simple as that…

University timetabling is a critical administrative task that involves creating a learning and teaching schedule for students and staff, respectively. It is undertaken by *schedulers* who are multi-skilled professionals needing to incorporate various skills including data analysis, change management, negotiating, complaint management and optimisation. (Dowland, 2018)

Timetabling complexity can vary from *simple* room bookings on one end of the scale to bespoke, individual schedules. Institutional goals can range from maximising room utilisation to delivering a great student experience. More often than not, university will attempt to deliver **everything** and only realise a fraction of the objectives. There are several reasons for this and they will vary with each institution, but fundamentally, certain objectives are contradictory - if you maximise room utilisation, experience is likely to worsen. The other major reason is that there is no such thing as a *universally* good timetable - what is good for one student will almost certainly not be good for another.

So timetabling and timetablers always find themselves in the firing line of criticism - they cannot win and the old adage 'no news is good news' could not be truer.

2.2 Why is University Timetabling Challenging?

The complexity of university timetabling stems from several factors:

Scale: Universities often have tens of thousands of students, hundreds of programmes (courses), thousands of modules, and limited resources. **Constraints**: There are numerous hard constraints (e.g., room capacity) and soft constraints (e.g., preferences) to consider. **Interdependencies**: Changes in one part of the schedule have cascading effects on other parts. **Diversity of Needs**: Different organisational units (colleges, faculties, schools, departments) have varying requirements and preferences. **Optimisation Goals**: Balancing efficiency, fairness, and quality of education.

2.3 Operational Challenges

Some of the keyy operational challenges in university timetabling include:

- **Resource Utilisation**: Maximising the use of available rooms and equipment.
- **Conflict Resolution**: Avoiding scheduling conflicts for students and staff.
- **Adaptability**: Accommodating last-minute changes or unforeseen circumstances.
- **Equity**: Ensuring fair distribution of desirable time slots and resources.
- **Data Quality and Interoperability**: Ensuring clean, accurate data flows between interconnected systems.
- **Curriculum Complexity**: Managing the intricacies of programme deliveries including excluded combinations, elective choices.
- **Balancing Flexibility and Structure**: Accommodating diverse teaching styles while maintaining a consistent schedule.
- **Stakeholder Communication**: Effectively conveying timetable information and changes to all parties involved.

2.4 The Elusive "Good Timetable"

Despite best efforts, it's impossible to find universal satisfaction with a university timetable. What constitutes a "good" timetable is often subjective and can vary among stakeholders.

Based on surveys across various institutions, students typically prioritise (Dowland, 2018) (Norman, 2022):

- A clash-free timetable
- Early release of timetables
- Clear and accurate information
- Full-year timetable availability
- Minimal changes after publication
- Effective communication of any changes
- Balanced distribution of classes (e.g., no single-hour events on a day, some free days, consecutive hours)

Meanwhile, other stakeholders have different priorities:

- Individual students may prefer learning times that align with their personal commitments such as part-time employment or travel arrangements.

- Individual academic staff may prefer teaching times that align with their research or personal commitments.
- Administrators might focus on efficient resource utilisation and institutional sustainability.
- Departments may have specific needs for specialised rooms or equipment such as laboratories, etc.

This divergence in preferences and the complex interplay of constraints make it challenging to define and achieve a universally “good” timetable, setting the stage for ongoing research and improvement in this field. In the private sector, there are companies which will offer analysis and solutions to address some challenges but these are often blunt and focus on efficiencies (e.g. utilisation) as opposed to pedagogical and experience motivation.

https://www.sciencedirect.com/science/article/pii/S2667305323000789?via%3Dihub

TODO

Add todo box. Write draft intro page from proposal:

- university timetable
  - waht it is and why it is hard
  - no one is happy (refer, seed for quality)
- add references / quote about timetabl
  - update references
- add image of a university timetable

Write background and context - personal motivation

3 Background and Motivation

3.1 Background

Several years ago I was a timetabler (scheduler) and then moved into a different role - a data manager. As such, I became intimate with the data, systems and processes used to support an advanced timetabling unit. I was also very familiar with the challenges and demands that the team faces, as well as the criticisms and complaints. The experiences have left scars and as such, timetabling will always form a part of my professional career. The intense experiences of producing a timetable and shoring up the aftermath will always leave me feeling a little protective of timetablers.

3.2 Motivation

Recognising the many challenges inherent to the process of producing and discussing a timetable, my motivation in this project is to explore ways of adding insight to the conversation rather than relying on anecdotal evidence or samples of n=1. This project is not about timetable production or timetable optimisation or timetabling strategies. It is not about adding to the University Course Timetabling Problem (UTCP) discourse. (Abdipoor et al., 2023) It is about contributing to how we assess, quantify, quality and rate a university timetable. It is about laying the foundations towards data-evenidenced timetabling metrics which can be applied objectively to a collection of individual timetables.

Some of the challenges experienced in timetable are very simple on the face of it. It can be difficult to answer simple questions around timetable 'quality' - the project aims to explore this but from the angle of graph data structures. It is very much a practical project which aims to contribute to the timetabler's toolkit.

1.1 Project Context and Motivation This project is motivated by the recognition that traditional timetabling approaches, while valuable, may not fully capture the nuances and complexities of the modern university environment. In particular, the increasing emphasis on student-centered learning and the need to provide a positive student experience have highlighted the limitations of conventional timetabling methods.

Anecdotal evidence suggests that the relational database structures commonly employed in timetabling systems may hinder the ability to effectively measure and report on the "quality" of a timetable. This limitation can impede efforts to understand the impact of timetables on student experiences and to make data-driven improvements.

Given the highly relational nature of timetables, where courses, instructors, rooms, and time slots are interconnected in intricate ways, it is hypothesized that a graph-based approach could offer a more natural and insightful way to represent and analyze timetable data. Graph databases, designed to handle complex relationships and interconnected data, have the potential to unlock new avenues for understanding and optimizing university timetables.

3.3 Project Context

This project is positioned as a proof of concept and exploratory study. The key aspects of this context include:

- **Proof of Concept**: The project aims to demonstrate the feasibility of using graph databases for timetabling analysis.
- **Exploration**: I will be exploring new methodologies and technologies that have the potential to enhance university timetabling insight and reporting.
- **Learning Opportunity**: This project serves as a platform to xxx
- **Foundation for Future Work**: While the scope is limited to a proof of concept, the findings from this project could lay the groundwork for more comprehensive solutions in the future.

4 Aim and Scope of Project

01-03 Aim and Scope of Project Project Aim The primary aim of this project is to [concise statement of your main goal, e.g., "investigate the viability of using graph data structures for enhanced timetabling analytics and reporting"]. Scope This project will focus on:

[Specific aspect of timetabling you're addressing, e.g., "Developing a graph-based data model for university timetables"] [Another key area of focus, e.g., "Implementing a prototype system for timetable analysis using graph databases"] [Additional area, e.g., "Comparing the analytical capabilities of graph-based and traditional relational database approaches"] Exploring the potential of graph-based approaches to address some of the key challenges in timetabling, such as curriculum complexity and data interoperability.

The project will not include:

[Aspect outside of scope, e.g., "Implementation of a full-scale timetabling system"] [Another limitation, e.g., "Real-time timetable generation or optimization"]

Objectives The key objectives of this project are:

[Specific objective, e.g., "Design an extensible, system-agnostic graph data model for university timetables"] [Another objective, e.g., "Develop a mapping pipeline to transition from relational to graph database representations of timetables"] [Further objective, e.g., "Implement and evaluate a set of analytical metrics leveraging the graph data model"] [Final objective, e.g., "Compare the performance and capabilities of graph-based analytics against traditional relational database approaches"] Investigate how graph-based approaches might contribute to measuring and improving timetable 'quality' from various stakeholder perspectives.

Through these objectives, we aim to explore the potential of graph-based approaches in enhancing our understanding and analysis of university timetables, potentially opening new avenues for timetable quality assessment and optimization. This work aligns with the growing recognition of timetabling as a critical factor in institutional sustainability, student satisfaction, and overall university operations.

1.2 Aim and Scope This project aims to investigate the viability of using graph data structures for university timetables and to explore the potential for enhanced reporting capabilities, including the establishment of measures of "timetable quality." The project will focus on the following key areas:

Foundational Research: The project will contribute to the foundational research in the field of graph-based timetabling analytics, exploring new ways to leverage graph data structures to gain deeper insights into timetabling data. Proof-of-Concept Analytics: A focused graph-based timetabling prototype will be developed to address specific analytical use cases relevant to university timetabling. This prototype will demonstrate the potential of graph-based analytics to go beyond traditional scheduling and optimization benchmarks. Comparative Evaluation: A comparative assessment between graph and relational databases will be conducted to evaluate the effectiveness of graph approaches for focused analytical tasks. This evaluation will highlight the potential advantages of graph-based methods over traditional relational approaches. Metrics for Timetable Quality: The project will develop and expand metrics to measure timetable "quality," moving beyond conventional scheduling and optimization benchmarks. These metrics will enable robust comparisons of different timetable instances or versions, taking into account factors that contribute to a positive student experience. By addressing these key areas, this project aims to lay the groundwork for a new paradigm in university timetabling, one that leverages the power of graph data structures to enhance reporting capabilities, improve decision-making, and ultimately create timetables that better serve the needs of students, instructors, and institutions.

5 Graph Data Model

5.1 Graph Data Model for Timetabling (1000 words)

5.1.1 2.1 Comparison of Relational and Graph Models

- Visual representation of both models - mermaid, or simmilar

5.1.2 2.2 Advantages of the Graph Approach

5.1.3 2.3 Data Augmentation Opportunities

- Room properties example (lat, long)
- Potential for additional data integration (curriculum, student outcomes, etc.)

2 Graph Data Model Given the highly relational nature of timetables, where courses, instructors, rooms, and time slots are interconnected in intricate ways, it is hypothesized that a graph-based approach could offer a more natural and insightful way to represent and analyze timetable data. Graph databases, designed to handle complex relationships and interconnected data, have the potential to unlock new avenues for understanding and optimizing university timetables.

2.1 Graph vs. Relational Data Structures (General) In the realm of data management, two prominent paradigms exist: the relational model and the graph model. The relational model, exemplified by relational databases like SQL Server, MySQL, and PostgreSQL, organizes data into tables with rows and columns. Each row represents an entity, and columns store the attributes of that entity. Relationships between entities are established through foreign keys, creating a structured and rigid schema.

In contrast, the graph model, employed by graph databases like Neo4j, Amazon Neptune, and Microsoft Azure Cosmos DB, represents data as nodes and edges. Nodes are the fundamental entities, and edges define the relationships between them. Both nodes and edges can have properties, allowing for flexible and dynamic schemas.

The choice between relational and graph databases depends on the nature of the data and the types of queries that need to be executed. Relational databases excel at structured data with well-defined relationships, making them suitable for transactional systems and reporting.

However, they can become cumbersome when dealing with complex, interconnected data, as queries often involve multiple joins across tables, leading to performance bottlenecks.

Graph databases, on the other hand, are designed to handle complex relationships and interconnected data naturally. They shine in scenarios where the focus is on exploring relationships, traversing networks, and uncovering patterns in interconnected data. Graph databases offer superior performance for queries that involve traversing multiple relationships, as the relationships are explicitly represented as edges in the graph.

2.2 Pros and Cons of Graph Databases Graph databases offer several advantages over relational databases, particularly when dealing with highly connected data:

Flexibility: Graph databases allow for flexible and dynamic schemas, making it easy to add new types of nodes and edges as the data evolves. This flexibility is crucial in domains like university timetabling, where the relationships between entities can be complex and subject to change. Scalability: Graph databases are designed to scale horizontally, distributing data across multiple machines to handle large datasets and high query volumes. This scalability is essential for university timetabling, as the number of courses, students, and instructors can be substantial. Performance: Graph databases excel at traversing relationships, making them ideal for queries that involve exploring connections between entities. In the context of university timetabling, this translates to faster and more efficient queries for tasks like finding all courses taught by a particular instructor or identifying scheduling conflicts. Expressiveness: Graph query languages, such as Cypher, are designed to express complex relationship-based queries concisely. This expressiveness simplifies the formulation of queries that would be cumbersome in SQL, the query language for relational databases. However, graph databases also have some limitations:

Maturity: Graph database technology is relatively young compared to relational databases, and the ecosystem of tools and libraries is still evolving. Learning Curve: Graph query languages, while expressive, require a learning curve for those familiar with SQL. Limited Support for Aggregation: Graph databases are not optimized for aggregation queries, such as calculating averages or sums, which are common in relational databases. 2.3 Comparison of Timetable Data in Relational and Graph Structures In a relational database, timetable data is typically organized into tables, with each table representing an entity like courses, rooms, instructors, and time slots. Relationships between entities are established through foreign keys, creating a structured schema. For example, a “course” table might have columns for course code, name, and credits, while an “instructor” table might have columns for instructor ID, name, and department. A “schedule” table would then link courses, instructors, rooms, and time slots using foreign keys.

In contrast, a graph database represents timetable data as nodes and edges. Nodes represent entities like courses, rooms, instructors, and time slots, while edges represent relationships between them. For instance, an edge labeled “scheduled_in” could connect a course node to a room node, indicating that the course is scheduled in that room. Similarly, an edge labeled “taught_by” could connect a course node to an instructor node, signifying that the course is taught by that instructor.

6 Data Engineering Overview

⚠ TODO

- add references - neo4j, graphviz

6.1 Overview of the Data Pipeline

The data engineering pipeline is designed to efficiently and securely transfer selected university timetabling data from a relational database (MS SQL) to a graph database (Neo4j), enabling advanced analytics and insights.

This section provides an overview of the pipeline architecture, fundamental design principles, implementation approach and key learning takeaways.

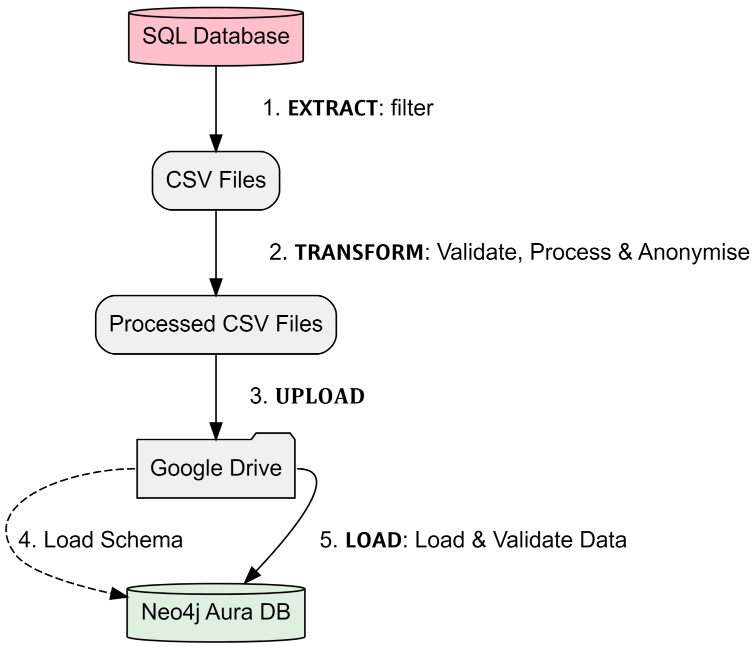
6.1.1 High-level Architecture

The data pipeline consists of these core stages:

1. **Extraction**: Data is extracted from the SQL database and saved into CSV files.
2. **Transformation**: The CSV files are processed, cleaned, transformed, merged, and anonymised using Python code.
3. **Intermediate Storage**: Processed CSVs are saved locally and uploaded to Google Drive (required for Neo4j Aura free instance).
4. **Loading**: Clean data is processed and loaded into Neo4j.

6.1.2 Design Principles

This pipeline represents a comprehensive approach to data engineering, incorporating several best practices in data handling, processing, and database management.



Data Pipeline Overview

Figure 6.1

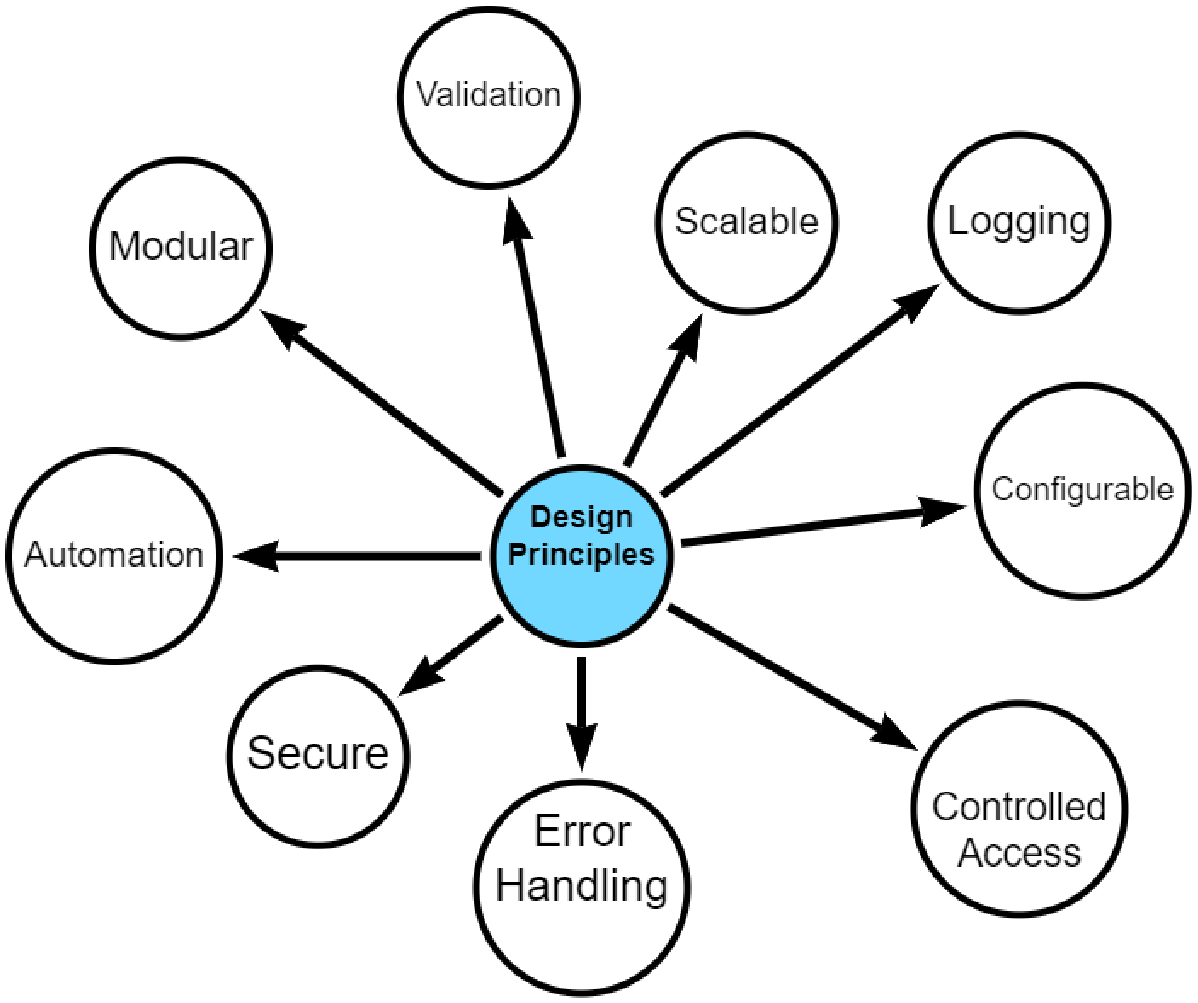


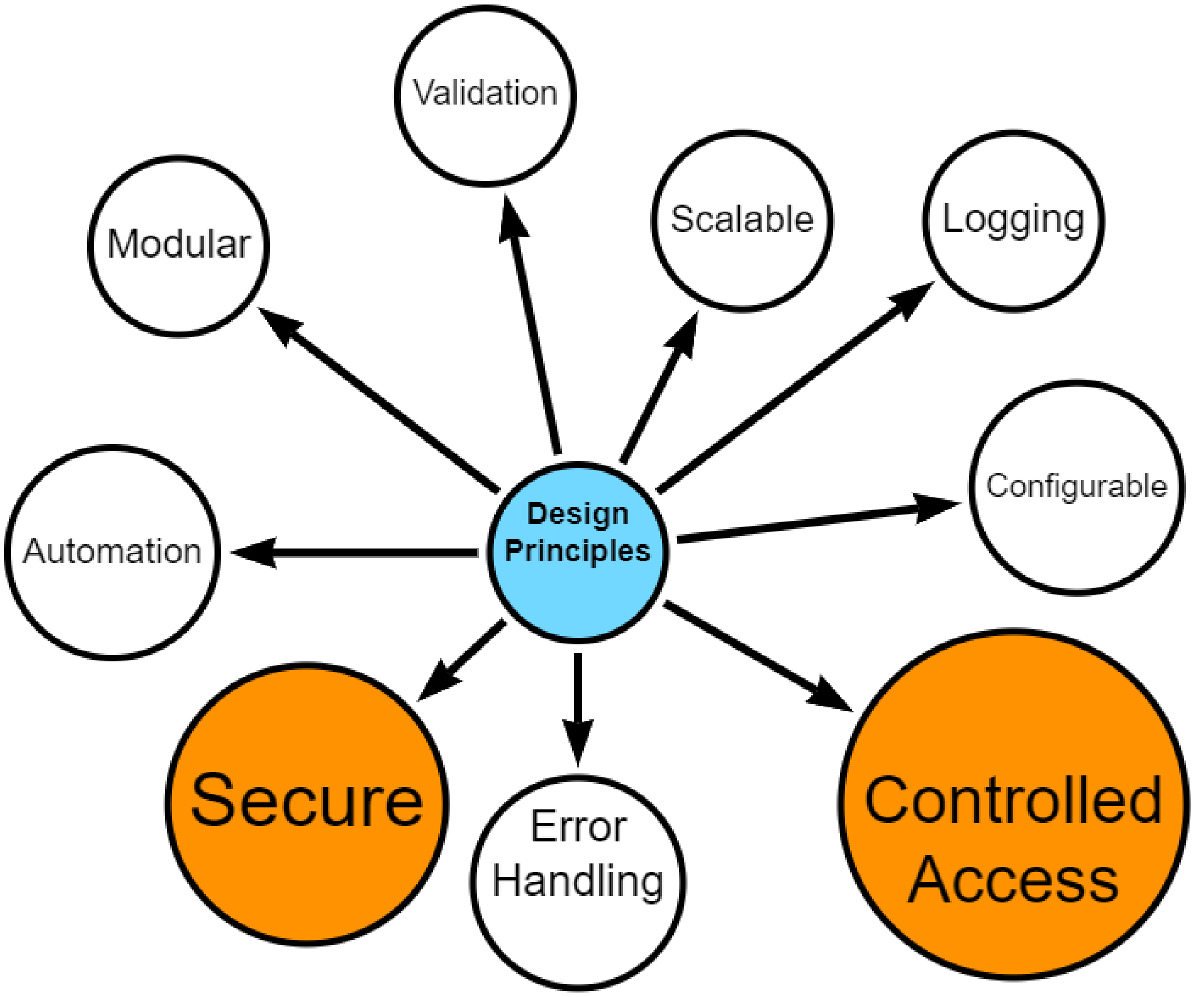
Figure 6.2: Design Principles

The data pipeline is built on several core design principles. I started with a strong sense of what I wanted to achieve - a modular, scalable, secure and configurable design - however, what *exactly* this meant was discovered during the development process.

Given that my project bound by time and word-limits and has additional resource and technology constraints, it was important to make the final artefact one which can be built upon after submission, including potential further development within operational contexts.

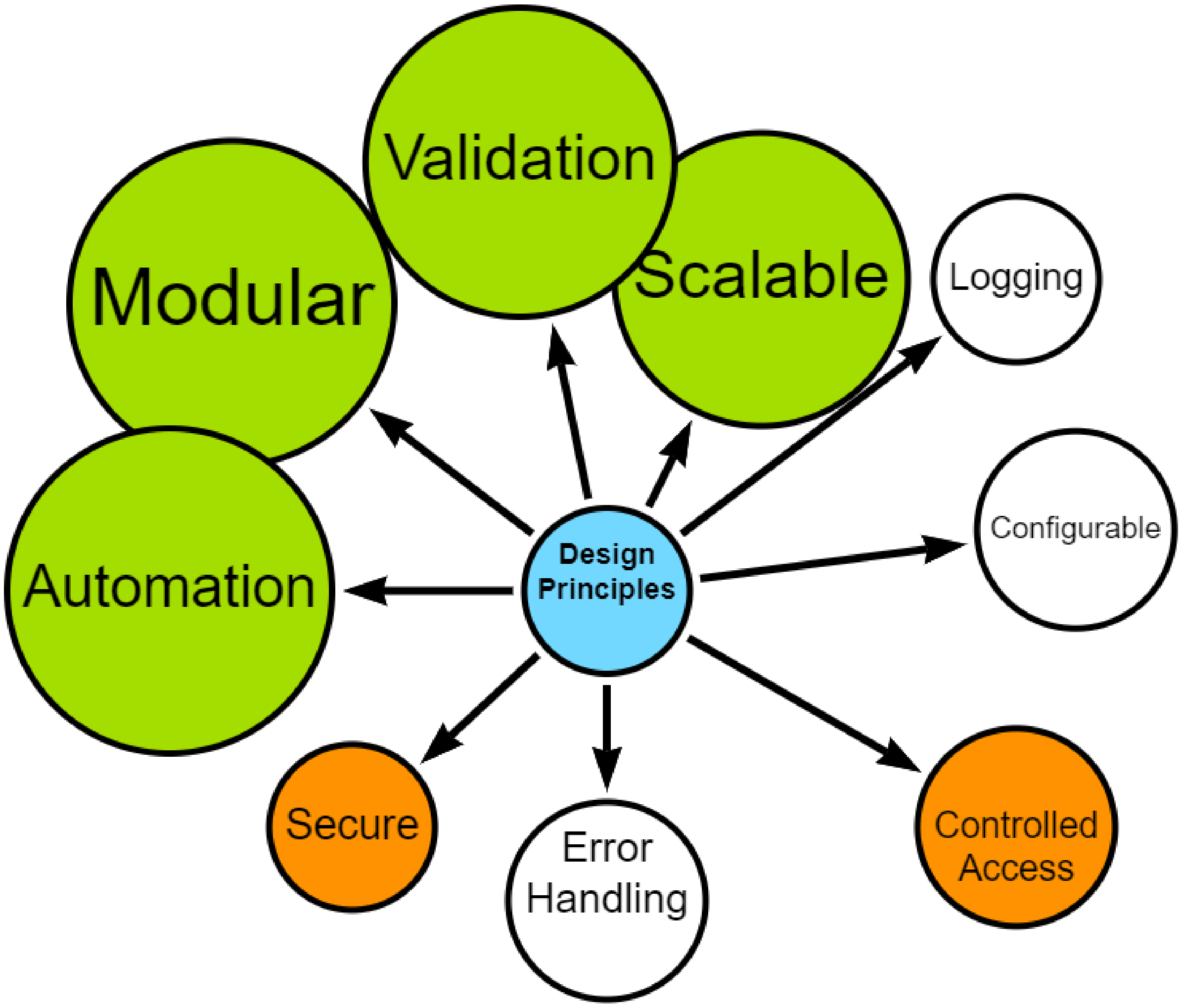
However, the project is also a *proof-of-concept* and as such, some design opportunities were eschewed in favour of simplicity and progress.

6.1.2.1 Security and Data Protection



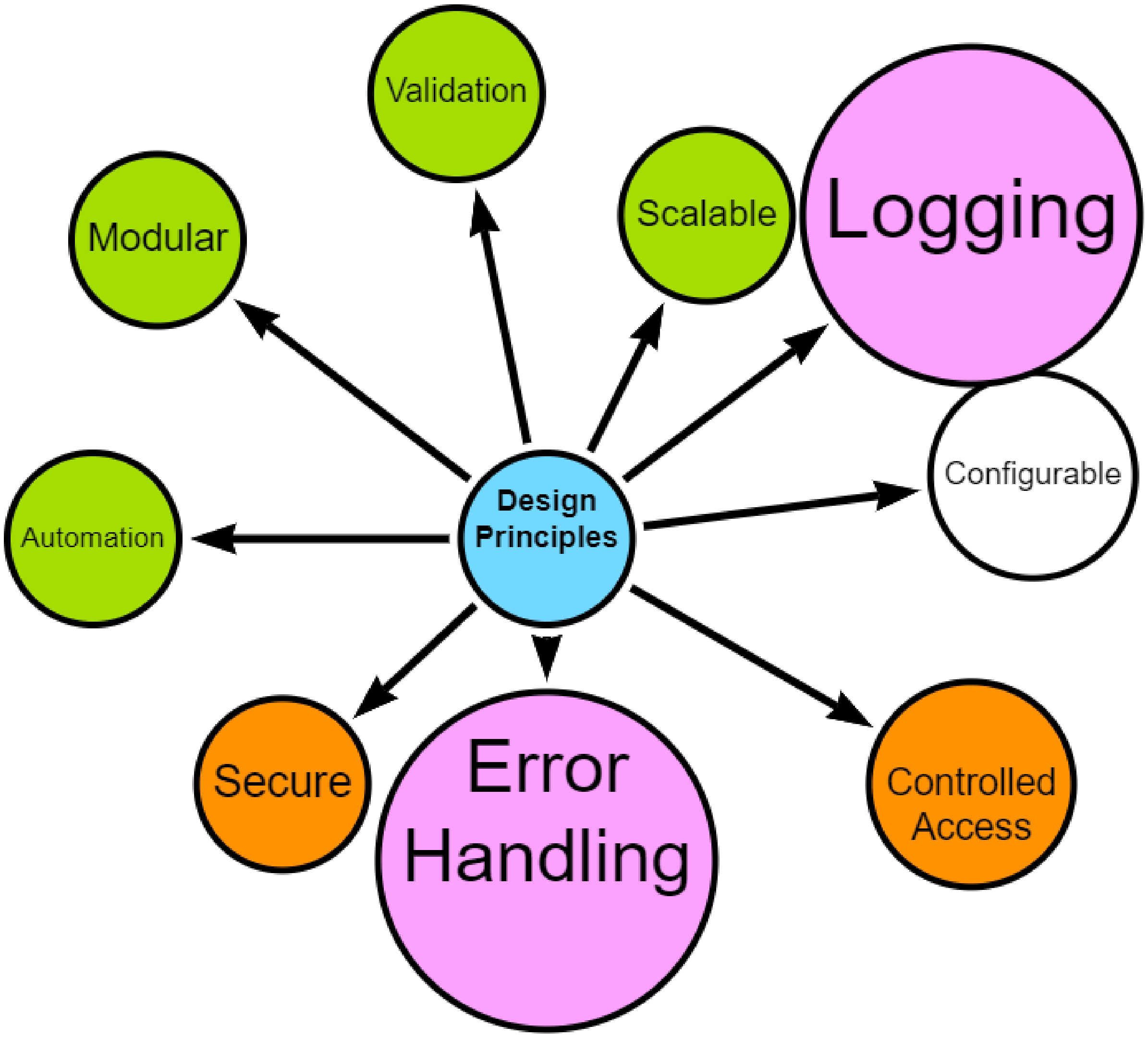
- Secure access controls
- Data anonymisation
- Controlled handling of personally identifiable information

6.1.2.2 Modularity, Scalability and Automation



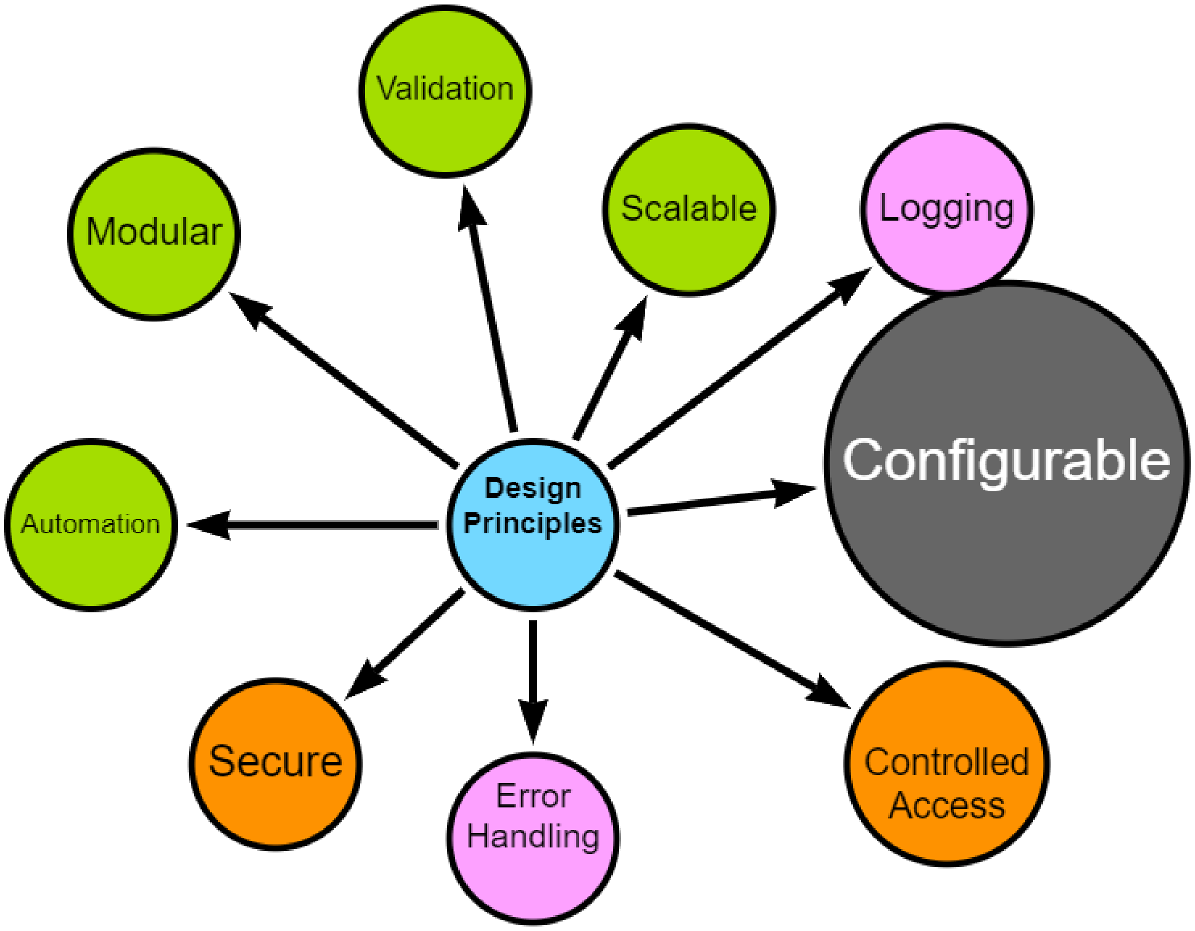
- Distinct, interoperable modules (extract, transform, load)
- Ability to handle increased data volume and complexity
- Automation, where possible
- Configurable data processing options (e.g., data chunking, row processing)
- Optimised, where possible

6.1.2.3 Error Handling and Logging



- Robust error handling
- Comprehensive logging for troubleshooting and auditing

6.1.2.4 User configurable



- Flexible configuration options for data filtering, directory controls, and schema handling

6.1.3 Implementation Approach

The pipeline was developed using an iterative approach, allowing for continuous discovery, refinement and improvement.

Crucial aspects of the implementation include:

- **Technology Stack**: Python for data processing, MS SQL for source data, Neo4j for the target graph database. See Appendix for more details.
- **Cloud Integration**: Utilisation of Google Drive for intermediate storage, compatible with Neo4j Aura.

- **Validation**: Implemented at various stages to ensure data integrity and fitness for processing.
- **Testing**: Continuous simulated unit testing to ensure that componentsare behaving as expected.

6.1.4 Upcoming Sections

The following sections will delve into the specific implementation details of each stage in the pipeline, demonstrating how these principles are put into practice.

I will explore the iterative development process, configuration management, extraction techniques, transformation processes, loading strategies, and automation workflows.

Finally, I will reflect on lessons learned and potential future enhancements to the data engineering components.

7 Data Engineering Approach

⚠ TODO

- Add agile reference

I followed an interative, agile-inspired approach when developing the data pipeline, despite being a team of one. This allowed for flexibility, continuous improvement and the opportunity to adapt to new insights during the process. The bulk of my effort was spent *prototyping*, *testing* and *reviewing* with each iteration resulting in a new challenge, issue, opportunity or occasionally, success.

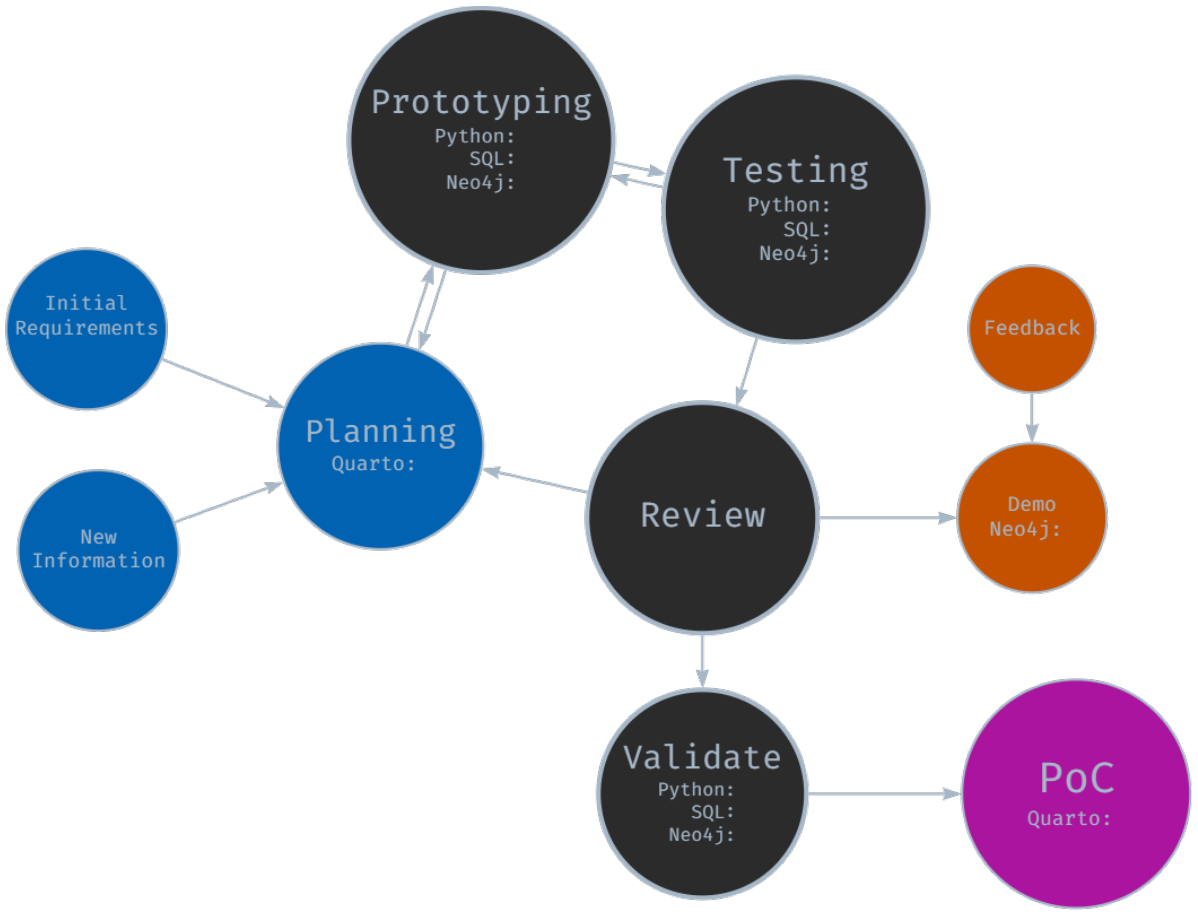


Figure 7.1: Iterative Development Approach

7.0.1 Initial Planning and Requirements Gathering

The development cycle began with initial high-level planning and requirements gathering, where I imagined how each stage should work, trying to bear in mind future-proofing and repeatability principles.

I defined core functionality for each module (extraction, transformation, loading) and outlined initial technical requirements and constraints. The planning documentation was maintained in Quarto and markdown files in a centralised repository for project information.

7.0.2 Prototyping

Following the initial planning, rapid prototyping was undertaken for each module:

- SQL prototyping for data extraction queries
- Python prototyping for data transformation and processing logic
- Neo4j prototyping for graph database schema and loading procedures

This stage allowed for quick exploration of different approaches and early identification of potential challenges as well as giving me the confidence to continue with my exploration.

7.0.3 Component-Based Development and Testing

Development proceeded with a focus on individual components:

- Each module (extraction, transformation, loading) was developed separately with a view to distinct "handovers"
- An iterative, component-based testing approach was employed
- While formal unit tests were not always created, each component was thoroughly tested for functionality

This approach allowed for continuous progress while maintaining a focus on component-level quality. It was during this phase that I started expanding configuration, logging and error-handling options.

7.0.4 Integration -> Review -> Demo -> Feedback -> Repeat

As components reached a (more) stable state, they were integrated and reviewed:

- Components were combined to form larger functional units
- Integrated functionality was occasionally demonstrated to subject matter experts (operational timetablers, timetable manager, data manager)
- Feedback was gathered on functionality, usability, and alignment with requirements

Insights gained from reviews, demonstrations and ongoing development were continuously fed back into the process. New requirements or modifications were documented, for example updates to SQL SELECT statements and data model interpretations.

With each new piece of information or change, decisions were required and made - but not always the right ones.

7.0.5 Version Validation and Documentation

At pivotal junctures, e.g., when a stable version was achieved:

- End-to-end validation of the entire pipeline was performed.
- Results were documented in notebooks, including opportunities for improvement.
- Any issues (or opportunities) identified were logged for the next iteration.

7.0.6 Continuous Learning and Adaptation

Throughout the development process, learning and adaptation became central to the project's evolution. Each iteration brought new insights, often through trial and error and certainly through unintended consequences or unforeseen complications. Early challenges included the need to modularise components *before* they became unmanageable, resisting the temptation to make overly ambitious changes or indeed resisting the temptation to carry on when it would have been better to pause and shore up progress. With practice, I became better at recognising when refactoring was necessary. These experiences underscored the importance of incremental progress and consistent testing in maintaining project stability and direction.

This iterative journey was far from linear. There were many moments of frustration, periods of painstaking troubleshooting, and the constant urge to overdeliver, often exceeding the original proof-of-concept scope. Yet, with each stumble, the process itself became more refined, transforming into a powerful tool for identifying and resolving issues.

While the core MVP (minimum viable product) requirements remained relatively stable (I set them after all!), the iterative approach empowered me to seize opportunities for enhancement. Each chance to modularise, parameterise, or fine-tune sparked an almost compulsive drive for improvement, pushing the pipeline beyond its initial scope.

This dedication to continuous refinement, while time-consuming, ultimately resulted in a robust, flexible solution that can adapt gracefully to unforeseen challenges and serve as the starting point for future opportunities.

However, this rigorous development process naturally led to a greater focus on data engineering rather than delving into the potential insights offered by Neo4j, simply due to the allocation of time and resources.

The iterative approach proved to be more than just a development methodology. It facilitated personal growth, enhanced technical skills, and improved project management capabilities.

8 Configuration and Logging

⚠ TODO

- rewrite detailed options into a summary format
- update if YAML config implemented.
- ☐ revisit approach doc in final script folder

The configuration and logging approach was to centralise configuration parameters into a python scripts in order to allow the user to manage different aspects of the ETL pipeline. The configuration options are as a result of both initial design and discovery during development.

When I set options during prototyping and testing, I considered whether these are worth parameterising in the ETL by weighing up cost versus value within a proof-of-concept scope. In general, my design is set up to run *automatically* or *dynamically* with well structured data, but I included options to override these settings.

8.0.1 Main Configuration options

An YAML file containing configuration options can be viewed in the Appendix.

General settings include being able to filter which data to extract by way of a list of award codes. This is the default folder name for this ETL run, although default folder names, filepaths and directories can be overwritten and customised. This includes source data files and Google Drive folders for processed data.

Credentials to SQL databases, Neo4j instances and Google drives are stored securely via environment variables or keyring secure storage.

There is an option to specify additional data sources, that is data which does not originate in the timetable database. This has been configured for optionally adding additional location data, e.g. latitude and longitude, square meterage, etc.

8.0.2 Logging

I set up a mechanism which can create a separate loggers with ease. This can be expanded or contracted, as required - for example, during development I had sepaparate loggers for each stage of the ETL to allow me to understand errors. However, if the ETL was developed into a stable release, it can be configured to have one log.

Options include customising the log level (DEBUG, INFO, WARNING, ERROR, CRITICAL).

8.0.2.1 Example Extract Log

8.0.2.2 Example Google Drive Log

As part of the logging functionality, I created a timing function which tracks and stores various execution and elapsed times with a view to optimising performance or identifying bottlenecks in future development.

8.0.2.3 Example Process Log

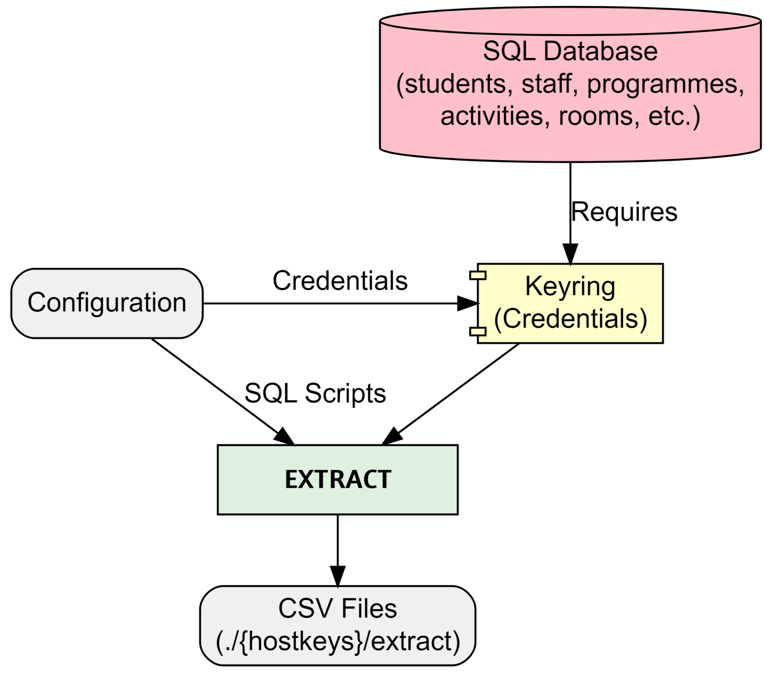
Logging also summarises loading results as can be seen in the snippet below.

8.0.2.4 Example Load Log

9 Extraction

⚠ TODO

-



Extract

The extraction process starts by securely connecting to the specified SQL database using encrypted credentials stored with keyring. The combination of `configuration` and `SQL scripts` determine which data will be extracted by filtering based on programme(s) of study and specifying which nodes, relationships and properties to extract. Additional options include specifying `chunk size` if extracting signficicant amounts of data, for example.

The process performs basic validation at every step ensuring secure connection before running SQL SELECT statements and storing extracted data as csv files locally.

9.0.1 SQL example

```sql
SELECT DISTINCT a.[Id] AS actSplusID,
     CONCAT(a.[Id], '-', adt.[Week], '-', adt.[Day]) AS actGraphID,
     a.[Name] AS actName,
     a.[Description] AS actDescription,
     a.[DepartmentId] AS actDeptSPlusID,
     adt.[StartDateTime] AS actStartDateTime,
     adt.[EndDateTime] AS actEndDateTime,
     adt.[Week] AS actWeekNum,
     adt.[Occurrence] AS actOccurrence,
     a.[ModuleId] AS actModSplusID,
     a.[ScheduledDay] AS actScheduledDay,
     a.[StartDate] AS actFirstActivityDate,
     a.[EndDate] AS actLastActivityDate,
     a.[PlannedSize] AS actPlannedSize,
     a.[RealSize] AS actRealSize,
     a.[Duration] AS actDuration,
     a.[DurationInMinutes] AS actDurationInMinutes,
     a.[NumberOfOccurrences] AS actNumberOfOccurrences,
     a.[WeekPattern] AS actWeekPattern,
     a.[ActivityTypeId] AS actActivityTypeSplusID,
     a.[WhenScheduled] AS actWhenScheduled,
     a.[IsJtaParent],
     a.[IsJtaChild],
     a.[IsVariantParent],
     a.[IsVariantChild]
FROM ##TempActivity a
INNER JOIN ##TempActivityDateTime adt ON a.[Id] = adt.[ActivityID];
```

9.0.2 extract_main.py snippet

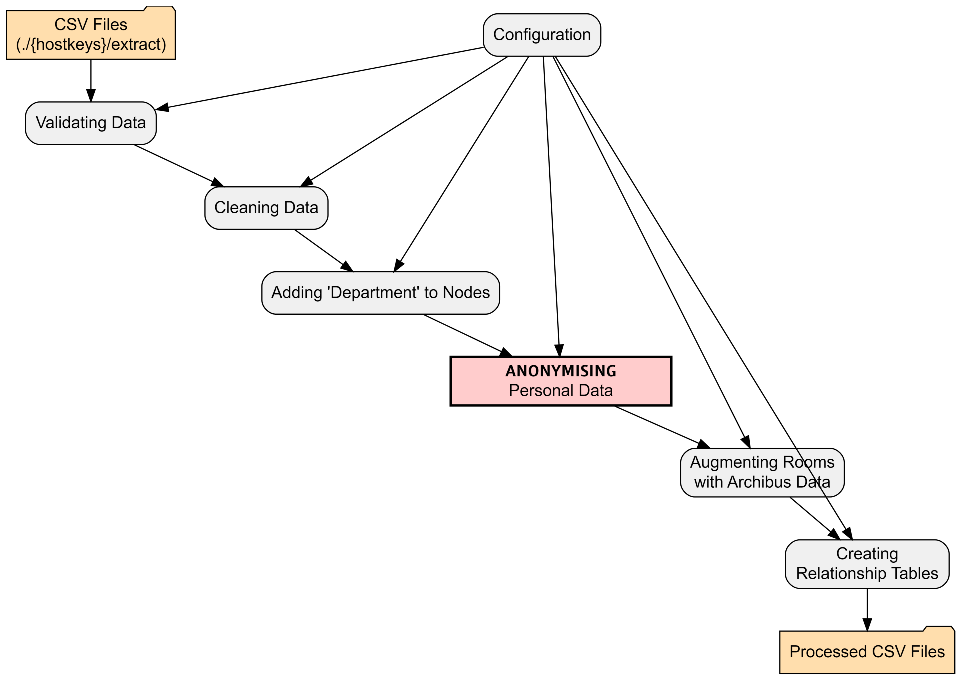
```python
# extract_main.py
from logger_config import extract_logger
from extract_data import main as extract_main
from config import EXTRACT_DIR, HOSTKEYS, CHUNK_SIZE
from utils import execution_times

def run_extraction():
    extract_logger.info("Starting data extraction process")
    extract_logger.info(f"Output Directory: {EXTRACT_DIR}")
    extract_logger.info(f"Hostkeys: {HOSTKEYS}")
    extract_logger.info(f"Chunksize: {CHUNK_SIZE}")

    try:
        extract_main()
    except Exception as e:
        extract_logger.exception("An error occurred during data extraction:")
    finally:
        extract_logger.info("Data extraction completed.")


    # Log the execution times
    extract_logger.info("Extraction Time Summary:")
    for func_name, exec_time in execution_times.items():
        extract_logger.info(f"Function {func_name} took {exec_time:.2f} seconds")


if __name__ == "__main__":
    run_extraction()
```

10 Transformation

TODO

-

The transformation section of the ETL pipeline picks up from where `extract` finished by using the extracted csv files as the source.



CSV Files (./{hostkeys}/extract)
Configuration
Validating Data
Cleaning Data
Adding 'Department' to Nodes
ANONYMISING Personal Data
Augmenting Rooms with Archibus Data
Creating Relationship Tables
Processed CSV Files

The configuration files allows the user to specify which columns should be used as the unique identifier when determining uniqueness, creating relationships between nodes and linking to additional datasets. It is also possible to specify datatypes - the load process will automatically load properties as `string` unless it is well formatted or the datatype is predetermined. The config file allows the user to specify how to handle certain datatypes like dates, times, boolean, etc.

10.1 All data

1. **Validation** - basic validation of the data is performed. Validation is extensible and can be expanded, as requirements are identified.
2. **Cleaned** - basic cleaning of all data is performed by stripping empty space and removing non-printable characters, etc. using regex. The cleaning functionality can be expanded.

With clean data, the transformation proper starts:

10.2 Nodes and relationships

1. **Add Organisational Unit** - where appropriate, the University Organisational Unit (e.g. College, School, Department) is added to the node. This will be picked up as a property during load.

2. **Data Augmentation** - Room data is augmented with additional properties from the location master database, including latitude, longitude, square meterage, etc. Data augmentation is extensible.
3. **Anonymisation** - Personal data is anonymised. An anonymisation function was developed to remove and replace any personally identifiable information (PII). The pipeline extracts minimal PII but this is safely anonymised. The functional also adds fake emails. See Appendix for Anonymisation
4. **Relationships** - Based on requirements in the configuration, relationships are extracted including optional relationship properties.

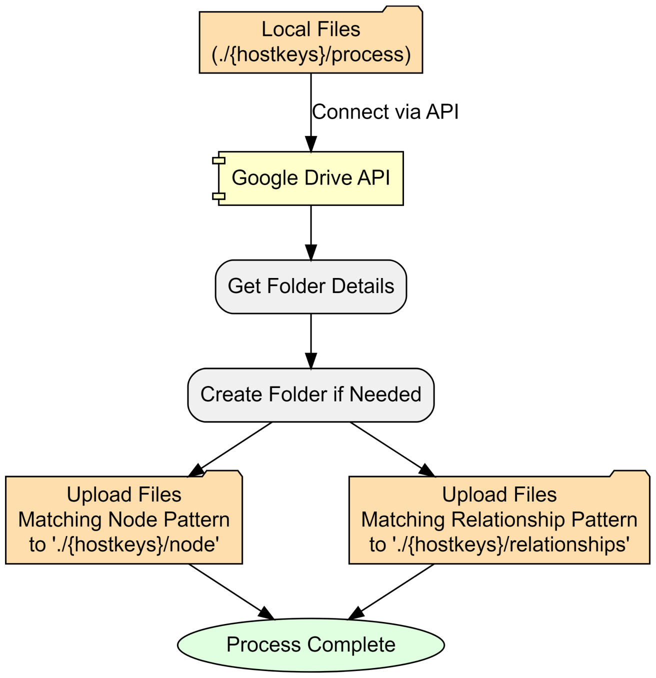
11 Google Load

⚠ TODO

- add log files?
- add screenshot?

As I am using a free instance of Neo4j's graph database - called Neo4j Aura - I needed to overcome some limitations which are not relevant to desktop installations of Neo4j or paid-for cloud instances.

In order to load from csv files, Neo4j Aura requires that the csv files are stored in public cloud storage like Google Drive or Dropbox. Therefore, my project requires this intermediary step.

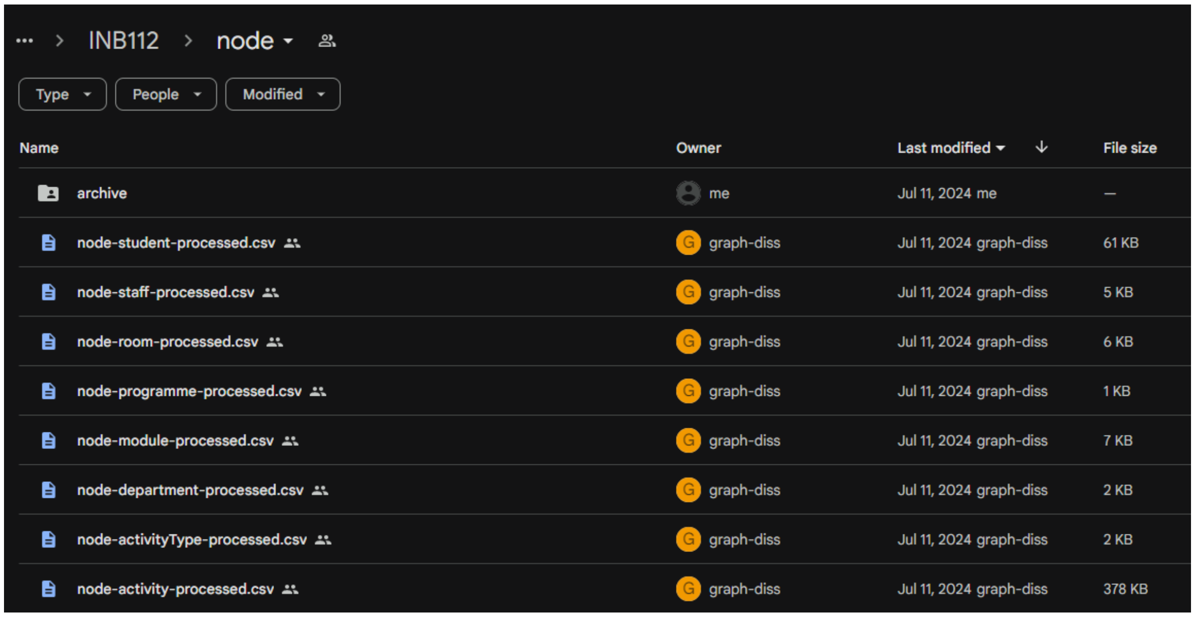


Local Files (./{hostkeys}/process)
Connect via API
Google Drive API
Get Folder Details
Create Folder if Needed
Upload Files Matching Node Pattern to './{hostkeys}/node'
Upload Files Matching Relationship Pattern to './{hostkeys}/relationships'
Process Complete

Configuration settings determine where processed node and relationship files are stored. I have made one folder in my drive public and all ETL files are stored within this root as follows:

- root Google Drive folder

- hostkeys (automatically created, unless override)
    * nodes
    * relationships



Name	Owner	Last modified	File size
archive	me	Jul 11, 2024 me	—
node-student-processed.csv	graph-diss	Jul 11, 2024 graph-diss	61 KB
node-staff-processed.csv	graph-diss	Jul 11, 2024 graph-diss	5 KB
node-room-processed.csv	graph-diss	Jul 11, 2024 graph-diss	6 KB
node-programme-processed.csv	graph-diss	Jul 11, 2024 graph-diss	1 KB
node-module-processed.csv	graph-diss	Jul 11, 2024 graph-diss	7 KB
node-department-processed.csv	graph-diss	Jul 11, 2024 graph-diss	2 KB
node-activityType-processed.csv	graph-diss	Jul 11, 2024 graph-diss	2 KB
node-activity-processed.csv	graph-diss	Jul 11, 2024 graph-diss	378 KB

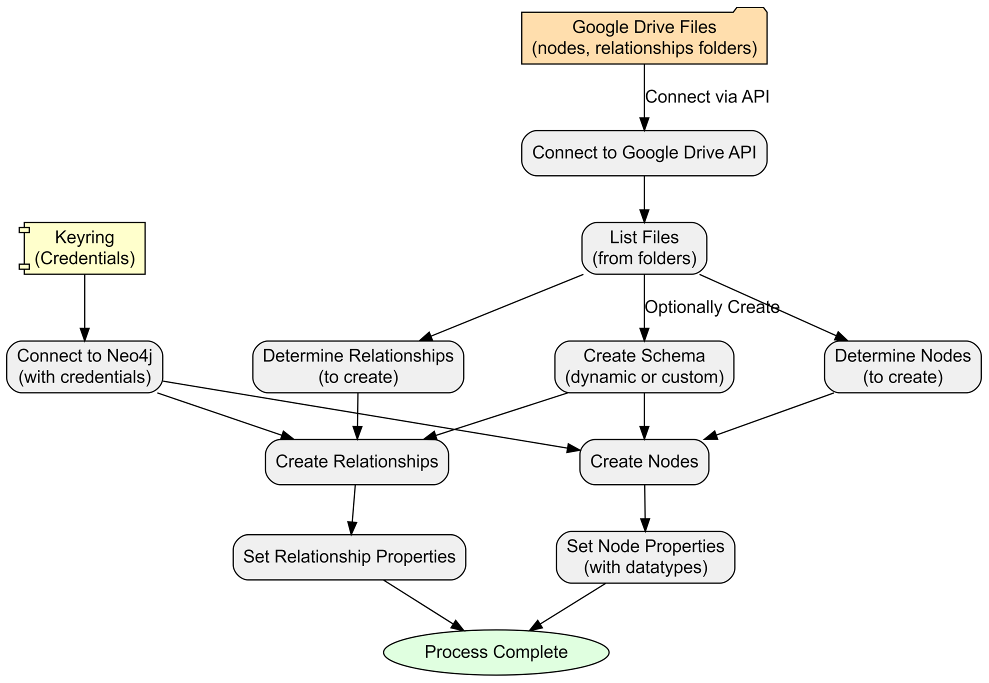
Figure 11.1: Screenshot of Google Drive

12 Neo4j Load

TODO

- add log files?
- add screenshot?
- add neo4j loaded data screenshot

With accessible csv files, the final module of the ETL pipeline creates (or updates) nodes and relationships in the Neo4j instance.



Google Drive Files (nodes, relationships folders)
Connect via API
Connect to Google Drive API
Keyring (Credentials)
List Files (from folders)
Optionally Create
Connect to Neo4j (with credentials)
Determine Relationships (to create)
Create Schema (dynamic or custom)
Determine Nodes (to create)
Create Relationships
Create Nodes
Set Relationship Properties
Set Node Properties (with datatypes)
Process Complete

There are two authentication requirements:

1. **Google Drive** to get node and relationship files and data.
2. **Neo4j Aura instance** is connected to with Keyring encrypted credentials.

The process automatically processes nodes and relationships based on files in the specified folders by using a file-pattern matching approach. However, this can be overridden within configuration.

Also in configuration is the option to create a database schema. There are three options:

1. **No schema**
2. **Dynamic** (default) - creates unique constraints based on nodes
3. **Custom** - allows the user to specify specific constraints prior to loading.

At this point, the ETL loads data on a row-by-row basis, reading the public csv files. Columns become properties with data types cross-referenced from a data-mapping dictionary in the configuration.

If there have been no errors - we should have data in our Neo4j Aura instance!

13 Reflections

From the outset, I recognised that this data engineering project was ambitious in both scale and scope. However, the reality of its magnitude became increasingly apparent as development progressed. Despite my initial awareness, I found myself continually expanding the project's boundaries, often pushing for a "gold-plated" solution rather than acknowledging when certain aspects were "good enough." This tendency towards scope creep, while driven by a desire for excellence, has significantly increased the project's complexity and time requirements.

The learning curve has been exceptionally steep. I've had to rapidly acquire proficiency in a diverse range of technologies and tools: Python, Neo4j, Google APIs, Quarto, and GraphViz. This intensive learning process, while challenging, has been incredibly rewarding, expanding my technical toolkit far beyond my initial expectations. However, it has also contributed to the project's expanding scope, as each new skill acquired opened up possibilities for further enhancements.

Unexpected challenges have been a constant companion throughout this process. From deleted servers and access issues to discrepancies between development environments (such as missing user certificates), I've encountered a wide array of unforeseen obstacles. These issues have necessitated the development of strong troubleshooting skills and a flexible approach to problem-solving. While often frustrating, these challenges have also provided valuable learning opportunities, pushing me to deepen my understanding of the systems and technologies I'm working with.

13.1 Lessons Learned

1. **Scope management is crucial**: Work on recognising when a solution is "good enough" and resist the urge to continually expand scope. Set clear boundaries at the start and be prepared to reassess and adjust plans when necessary.
2. **Embrace modularisation from the beginning**: Avoid the temptation to create oversized code blocks. Maintain a list of "future enhancements" to prevent immediate implementation of every idea.
3. **Balance documentation with development**: Document sufficiently during the development process, but save comprehensive documentation for appropriate milestones. This approach maintains progress while ensuring proper record-keeping.

4. **View obstacles as learning opportunities**: Embrace continuous learning and see challenges as chances to grow. Invest time in understanding the right technologies and approaches, particularly focusing on modularisation.
5. **Celebrate incremental progress**: Recognise and appreciate small achievements throughout the development process. This helps maintain motivation and provides a clearer sense of overall progress.

14 Timetable Metrics

14.1 Timetable Quality Metrics and Insights (1500-2000 words)

14.1.1 4.1 Defining Timetable Quality

14.1.2 4.2 Implemented Metrics

- Constraint violations (max hours per day, days per week, lunch breaks, etc.)
- Distance-based metrics using room properties

14.1.3 4.3 Aggregation Methods

- Student-level, programme-level, and other relevant groupings

14.1.4 4.4 Cypher Queries for Metric Calculation

- Example queries with explanations

14.1.5 4.5 Visualization of Results

- Bloom visualisations or other relevant charts

15 Future Opportunities

15.1 Future Opportunities and Potential Insights (500 words)

- Discussion of potential analyses (module combinations, student clustering, etc.)
- Integration of additional data sources

16 Conclusion

16.1 Conclusion (500 words)

- Summary of key achievements
- Reflection on the project's impact and potential for timetabling processes
- Future work and recommendations

17 Random Graph Generator

The function below generates a random graph (dot file) using Graphviz.

To render, ensure that graphviz is installed or save to file and render within documents using Quarto or similar.

```python
import graphviz
import random
import string
from collections import defaultdict

def generate_random_graph(num_nodes=50, num_edges=100, num_clusters=5, colors=None):
    """Generates a random Graphviz graph with clusters and random colours.

    Args:
        num_nodes: Number of nodes in the graph.
        num_edges: Number of edges in the graph.
        num_clusters: Number of clusters to create.
        colors: List of colours to use for clusters (optional). If not provided, random colo
    """

    dot = graphviz.Digraph("G")
    dot.attr(fontname="Helvetica,Arial,sans-serif")
    dot.attr(layout="neato")
    dot.attr(start="random")
    dot.attr(overlap="false")
    dot.attr(splines="true")
    dot.attr(size="8,8")
    #dot.attr(dpi="300")

    # nodes to clusters, random colours if not provided
    cluster_assignments = {}
    if colors is None:
        colors = ["#%06x" % random.randint(0, 0xFFFFFF) for _ in range(num_clusters)]

    for i in range(num_nodes):
        cluster_assignments[i] = random.randint(0, num_clusters - 1)

    # random node names, colouur assignment
    nodes = []
    for i in range(num_nodes):
        node_name = ''.join(random.choices(string.ascii_lowercase + string.digits, k=8))
        nodes.append(node_name)
        cluster_id = cluster_assignments[i]
        color = colors[cluster_id]
        dot.node(node_name, label="", shape="circle", height="0.12", width="0.12", fontsize="


    # random edges (with a higher probability of staying within clusters)
    edges = []
    for _ in range(num_edges):
        src_cluster = random.randint(0, num_clusters - 1)
        dst_cluster = src_cluster if random.random() < 0.8 else random.randint(0, num_cluste
        src_node = random.choice([node for i, node in enumerate(nodes) if cluster_assignments
        dst_node = random.choice([node for i, node in enumerate(nodes) if cluster_assignments
        edges.append((src_node, dst_node))

    #  edges to the graph
    for edge in edges:
        dot.edge(*edge)

    return dot
```

18 Technology Stack

18.1 Technology Stack

TO ADD:

- Technology Stack
  - Python used
  - Python packages used and why
  - Neo4j used
  - Google API
  - VS code
  - Quarto
  - SQL
- graphviz
- arrows
- mermaid
- git hub, git

19 Configuration YAML

19.1 Config

The below is an example of configuration options configured in more human readable YAML format.

```yaml
# ETL Pipeline Configuration

general:
  hostkeys:
    - INB112
    # - N420
  folder_name: '' # default to hostkey if empty

file_paths:
  root_dir: '.'  # default to current working directory
  nodes_folder_url: # (Optional) override for dynamic lookup) eg "https://drive.google.com/di
  relationships_folder_url: # (Optional) override for dynamic lookup) eg."https://drive.googl
  gdrive_root_folder_url: "1iWkeTubJ0xZ6I728emoj9BkqZm7dL2fq"
  gdrive_folder_name: # Leave commented out to use default (hostkey)
  google_credentials_path: 'credentials/graph-diss-dbbdbb5e5d00.json'
  department_source: 'node-dept-all.csv'
  archibus_source: 'archibus.csv'

data_processing:
  chunk_size: 20000
  temp_tables_sql_file: "create_temp_tables.sql"
  node_output_filename_template: "node-{node}-processed.csv"
  rel_output_filename_template: "rel-{relationship}-processed.csv"

neo4j:
  #max_connection_retries: 5
  #max_transaction_retry_time: 30
  schema:
    apply: True
    type: 'dynamic' # Options: 'dynamic', 'custom'
    custom_path: ''
  batch_size: 1000

logging:
  log_level: "INFO" # Options: DEBUG, INFO, WARNING, ERROR, CRITICAL

nodes:
  department:
    filename_pattern: "node-dept-all*.csv"
    dept_join_col: null
    node_suffix: 'dept'
    node_id: "deptSplusID"
  module:
    filename_pattern: "node-module-by-pos-temp*.csv"
    dept_join_col: "modSplusDeptID"
    node_suffix: "mod"
    node_id: "modSplusID"
  room:
    filename_pattern: "node-room-by-pos-temp*.csv"
    dept_join_col: null
    node_suffix: 'room'
    node_id: "roomSplusID"
  programme:
    filename_pattern: "node-pos-by-pos-temp*.csv"
    dept_join_col: "posSplusDeptID"
    node_suffix: "pos"
    node_id: "posSplusID"
  activityType:
    filename_pattern: "node-activitytype-by-pos-temp*.csv"
    dept_join_col: 'actTypeDeptSplusID'
    node_suffix: 'actType'
    node_id: 'actTypeSplusID'
  staff:
    filename_pattern: "node-staff-by-pos-temp*.csv"
    dept_join_col: "staffDeptSplusID"
    node_suffix: "staff"
    dtype:
      staffSplusID: str
      staffID: str
    node_id: "staffSplusID"
  student:
```

```yaml
    filename_pattern: "node-student-by-pos-temp*.csv"
    dept_join_col: "stuDeptSplusID"
    node_suffix: "stu"
    dtype:
      stuSplusID: str
      studentID: str
    node_id: "stuSplusID"
  activity:
    filename_pattern: "node-activity-by-pos-temp*.csv"
    dept_join_col: null
    node_suffix: null
    dtype:
      actSplusID: str
      actTypeSplusID: str
      actRoomSplusID: str
      actStaffSplusID: str
      actStuSplusID: str
      actStartDateTime: str
      actEndDateTime: str
      actFirstActivityDate: str
      actLastActivityDate: str
      actWhenScheduled: str
    node_id: "actGraphID"

relationships:
  activity_module:
    filename_pattern: "rel-activity-module-by-pos-temp*.csv"
    node1_col: "actSplusID"
    node2_col: "modSplusID"
    relationship: "BELONGS_TO"
  activity_room:
    filename_pattern: "rel-activity-room-by-pos-temp*.csv"
    node1_col: "actSplusID"
    node2_col: "roomSplusID"
    relationship: "OCCUPIES"
  activity_staff:
    filename_pattern: "rel-activity-staff-by-pos-temp*.csv"
    node1_col: "staffSplusID"
    node2_col: "actSplusID"
    relationship: "TEACHES"
  activity_student:
    filename_pattern: "rel-activity-student-by-pos-temp*.csv"
    node1_col: "stuSplusID"
    node2_col: "actSplusID"
    relationship: "ATTENDS"
  activity_activityType:
    filename_pattern: "relActivityActType*.csv"
    node1_col: "actSplusID"
    node2_col: "actActivityTypeSplusID"
    relationship: "HAS_TYPE"
  module_programme:
    filename_pattern: "rel-mod-pos-by-pos-temp*.csv"
    node1_col: "modSplusID"
    node2_col: "posSplusID"
    relationship: "BELONGS_TO"
    properties:
      - "modType"

data_type_mapping:
  activity:
    actStartDateTime: ['datetime', '%Y-%m-%d %H:%M:%S']
    actEndDateTime: ['datetime', '%Y-%m-%d %H:%M:%S']
    actFirstActivityDate: ['date2', '%Y-%m-%d']
    actLastActivityDate: ['date2', '%Y-%m-%d']
    actPlannedSize: 'int'
    actRealSize: 'int'
    actDuration: 'int'
    actDurationInMinutes: 'int'
    actNumberOfOccurrences: 'int'
    actWhenScheduled: ['datetime', '%Y-%m-%d %H:%M:%S']
    actStartDate: ['date', '%Y-%m-%d']
    actEndDate: ['date', '%Y-%m-%d']
    actStartTime: 'time'
    actEndTime: 'time'
    actScheduledDay: 'int'
  room:
    roomCapacity: 'int'

display_name_mapping:
  activity: "actName"
```

20 Anonymisation

TODO

- wrtie a summary
- show before and after
- staff/student
- columns
- consistent changing - uses random seed to ensure

```python
import random
import hashlib
from faker import Faker
import pandas as pd


def anonymise_data(df):
    """
    Anonymises a DataFrame by generating fake names, emails, and IDs.
    """
    process_logger.info("Starting anonymisation")
    process_logger.info(f"Columns in dataframe: {df.columns.tolist()}")

    # Determine if it's staff or student data
    if 'staffSplusID' in df.columns:
        process_logger.info("Processing staff data")
        id_col = 'staffID'
        prefix = 'staff'
        columns_to_remove = ['staffFullName', 'staffLastName', 'staffForenames', 'staffID']
    elif 'stuSplusID' in df.columns:
        process_logger.info("Processing student data")
        id_col = 'studentID'
        prefix = 'stu'
        columns_to_remove = ['stuFullName', 'stuLastName', 'stuForenames', 'studentID']
    else:
        process_logger.error("Neither 'staffSplusID' nor 'stuSplusID' found in columns.")
        return df  # Return original dataframe if required columns are missing

    # Create a dictionary to store anonymised data
    anon_data = {}

    # Generate anonymised data for each unique ID
    for unique_id in df[id_col].unique():
        # Create a seed based on the unique_id
        seed = int(hashlib.md5(str(unique_id).encode()).hexdigest(), 16) & 0xFFFFFFFF
        fake = Faker()
        fake.seed_instance(seed)
        random.seed(seed)

        first_name = fake.first_name()
        last_name = fake.last_name()
        full_name = f"{first_name} {last_name}"
        email = f"{first_name.lower()}.{last_name.lower()}@fakemail.ac.uk"
        anon_id = f"{prefix}-{random.randint(10000000, 99999999):08d}"

        anon_data[unique_id] = {
            f'{prefix}FirstName_anon': first_name,
            f'{prefix}LastName_anon': last_name,
            f'{prefix}FullName_anon': full_name,
            f'{prefix}Email_anon': email,
            f'{prefix}ID_anon': anon_id
        }

    # Create a new DataFrame with anonymised data
    df_anon = pd.DataFrame.from_dict(anon_data, orient='index')

    # Reset the index and rename it to match the original ID column
    df_anon = df_anon.reset_index().rename(columns={'index': id_col})

    try:
        # Merge anonymised data with the original DataFrame
        df_result = pd.merge(df, df_anon, on=id_col)

        # Remove columns that should be anonymised
        columns_to_remove = [col for col in columns_to_remove if col in df_result.columns]
        df_result = df_result.drop(columns=columns_to_remove)

        process_logger.info("Anonymisation completed successfully")
        return df_result

    except Exception as e:
        process_logger.error(f"Error during anonymisation: {str(e)}")
        return df  # Return original dataframe if an error occurs
```

21 References