UPPSALA
UNIVERSITET

IT 23 128

Degree project 30 credits

November 2023

Lakehouse architecture for simplifying data science pipelines

data engineering and graph data mining explorations in Trase.earth for the traceability of supply chains driving deforestation

Nicolás Martín

Master's Programme in Data Science

UPPSALA
UNIVERSITET

Lakehouse architecture for simplifying data science pipelines: data engineering and graph data mining explorations in Trase.earth for the traceability of supply chains driving deforestation

Nicolás Martín

Abstract

Data management and pre-processing often consume the majority of time spent by data scientists. The data architecture and the configuration of data pipelines significantly influence the efficiency of this work. An emerging 'Lakehouse' architecture combines the features of both a Data Lake and a Data Warehouse, eliminating the need to manage a two-tier system. This allows for the storage and processing of raw, structured, and semi-structured data on a unified platform, offering higher performance and decoupling computing from storage. The capabilities of this architecture are explored within Trase.earth, a leading initiative in commodity supply chain transparency that focuses on agricultural products driving deforestation. This thesis demonstrates that the Lakehouse architecture can simplify intricate data pipelines while enabling new functionalities. It also shows that this transition can be made backwards-compatible, rely on open standards, and reduce costs. The enhancements analyzed include data ingestion from heterogeneous sources, data discoverability, metadata management, data sharing, and pipeline management with the integration of data quality expectations. As an additional case study, graph data mining techniques are applied to the beef supply chain in the state of Pará, Brazil, using a dataset of sanitary records for animal transportation. Various methods for deriving and analyzing paths of indirect sourcing are employed, facilitating the identification and characterization of the most frequently traveled routes, trade communities, and node centrality.

The code related to this thesis can be found in: https://github.com/nmartinbekier/ds_de_thesis .

Faculty of Science and Technology

Uppsala University, Uppsala

Supervisors: Harry Biddle and Vivian Ribeiro
Reviewer: Raazesh Sainudiin

Examiner: Matteo Magnani

Dedication

To Nathalie, the love of my life, guiding force, and anchor.
To Celeste and Elías, the greatest joy of our lives. Unconditional love to both of you.

Acknowledgements

Very special thanks to Raaz for his continuous guidance, dedication, and patience. Thank you also for the support through Combient Mix and the Combient Competence Centre for Data Engineering Sciences of Uppsala University for providing office space and a wide range of technology infrastructure to carry out the thesis work (Databricks, AWS, and Ericsson's Research Data Center).
Thanks to the Trase team, especially to my supervisors Harry and Vivian, and to Toby for opening the door and warm disposition. My admiration for what you do, and gratitude for your deep involvement, support, and feedback. I hope this thesis can contribute to your important work. It's been a privilege to learn, explore, and work with this aim.
Thanks to David for having started to take some of this work further.
Thanks to Oskar Åsbrink for the foundational work from which this thesis builds on, and for your insights and knowledge sharing. Thanks to Lovisa Eriksson for the comprehensive and generous comments and correction suggestions.
Thanks to the Neo4j Sweden office team for the availability, advice and feedback.

Table of Contents

List of Figures

1 Introduction

Data management and preparation typically consumes the majority of time and effort spent by data scientists [1]. Considerations in data engineering, particularly the setup of data pipelines —end-to-end systems that facilitate the flow and transformation of data— can make the work of data scientists in data preparation easier, faster, and overall more effective. For the purposes of this thesis, a data pipeline refers to a series of connected and managed data processing steps. Within a data pipeline, the architecture of the data is a critical factor affecting its manageability, simplicity, and cost. One architecture that has gained popularity since the early 1990s and is currently used by virtually all Fortune 500 companies [2] is the data warehouse. Data warehouses serve as time-variant, non-volatile central repositories that support analytical queries [3]. The most popular current warehouse solutions [4], such as Google BigQuery [5], Amazon Redshift [6], Snowflake [7], and Microsoft Synapse [8], are cloud-enabled. They allow for native connections to data lakes (centralized repositories where both unstructured and structured data are stored) through two-tier architectures. However, this two-layered structure introduces complexities, costs, and vendor lock-ins that can be eliminated by a single-tier lakehouse architecture, as proposed in the paper "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics" [2] in 2021. A lakehouse stores and processes raw, structured, and semi-structured data on a unified platform built on top of a data lake.

This thesis analyzes the data pipelines of Trase.earth as a case study to answer the research question: How can a lakehouse architecture simplify complex data pipelines used for analytical workloads? Moreover, can a transition to this architecture be made in a backward-compatible manner, while also relying on open standards and reducing existing technology costs? Trase [9], an initiative by the Stockholm Environment Institute and Global Canopy, is the leading endeavor for supply chain data transparency, focusing on mapping agricultural commodities that drive deforestation. Since its inception in 2015, Trase has evolved from mapping a few commodities to an increasingly diverse range of commodities and countries. This expansion has brought an increase in the number of domain experts and data scientists involved, thereby complicating the setup and management of data pipelines. Such complexity affects the ability to deploy new commodities or update existing ones. While a warehouse solution can assist with various management and processing needs, it can also introduce additional complexities, require extra onboarding efforts for contributing researchers, and potentially create vendor lock-in, as opposed to relying on open standards.

While addressing the main research questions, a range of solutions explored in conjunction with the lakehouse architecture are explored and developed, including:

- Ingestion of information from heterogeneous sources.
- Data discoverability and metadata management.
- Data pipeline definition, execution, and integration of quality expectations.
- Data sharing and connection with different systems.

While these solutions and the lakehouse architecture implementation are based on Trase's case study, explorations and findings should also apply to various contexts with heterogeneous data needs. Although most of the study cases regarding lakehouse architectures are focused on big data settings involving hundreds of computing nodes and managing petabytes of information [10], the current exploration makes a case of its usefulness for simplifying the management of data pipelines in small-to-medium sized data contexts cost-effectively. The greater separation of storage and computing needs from a traditional warehouse architecture makes it easier to upscale or downscale resources as needed and make the most of them while keeping costs down. These solutions are reviewed in the light of a "Data Engineering Lifecycle" approach, based on the framework presented in "Fundamentals of Data Engineering" [11]. As data architectures in particular and data engineering in general have undergone big changes in the past decade and will continue to do so in the near future, it is especially useful to ground the current work within wider and more stable points of reference.
Within the explorations of Trase's work making the supply chains of commodities driving deforestation more transparent, this thesis also explores possibilities of using graph theory and network analysis to extract insights from the beef supply chain in Brazil. The network of animal movements between farms up to slaughterhouses based on publicly available sanitary information is modeled, building upon previous analysis made by Trase's team while enabling more fine-grained analysis of animal movements between farms. This allows the creation of a detailed picture of possible indirect suppliers of the slaughterhouses, making it easier to identify their potential of indirectly sourcing from regions and suppliers linked with deforestation or human rights abuses.
The thesis is structured as follows:

- In Chapter 2, "Trase: trade intelligence for sustainable trade," the Trase initiative is introduced along with its data requirements and challenges.
- In Chapter 3, "Data engineering and pipelines exploration: lakehouse architecture," a data engineering framework is outlined. This chapter discusses the importance of data architectures, data pipelines, and specifically the lakehouse architecture. Proposals for its implementation within Trase are presented. These proposals

are designed to be backwards compatible and allow for incremental transitions. Key aspects of data pipelines relevant to Trase, including data ingestion, pipeline management, data discovery, metadata management, and data sharing, are also explored.

- In Chapter 4, "Graph data mining for the traceability of the beef supply chain in Brazil," a graph model is constructed to depict the indirect supply links within the beef supply chain. Algorithms for community detection, centrality, and shortest paths are applied to the model to generate insights. This analysis is based on 2 million records of animal movements in the region of Pará, Brazil.
- In Chapter 5, "Conclusions and future Work," a summary of the results, cost considerations, challenges, and directions for future work is provided. This encompasses both the data engineering and the graph data mining sections.

The work done in this master's thesis builds upon the work carried out by Oskar Åsbrink and Nicolás Martín on 1DL507 Project in Data Science course (1DL507) [12] from the Master's Programme in Data Science from Uppsala University, under the supervision of Raazesh Sainudiin from Uppsala University's math department, as well as Harry Biddle and Vivian Ribeiro from the Trase initiative [13].

2 Trase: trade intelligence for sustainable trade

2.1 General description of Trase

Trase.earth [9] is a leading supply chain data transparency initiative spearheaded by the Stockholm Environment Institute and Global Canopy. It maps the international trade and financing of agricultural commodities that contribute to tropical deforestation. Trase enables the identification of various actors and connections within a supply chain, ranging from producers in specific municipalities to exporting and importing companies, as well as the investors behind them. Traditionally, data about supply chains and their associated environmental impacts are aggregated (e.g., total exports of a commodity or overall deforestation at the country level), obscuring the specific exposure of investors, traders, and producers to particular deforestation events. Trase addresses this gap by integrating disparate data sources, allowing for the mapping of flows between subnational production regions and these actors, as depicted in figure 2.1 for the case of beef production.
To build each supply chain, the data requirements include:

- Per-shipment trade: this refers to detailed data for each individual shipment that moves through the supply chain, including information such as the quantity of goods, the date of shipment, origin, destination, and the parties involved.
- In-country asset information: this encompasses all data related to assets within a given country that are part of the supply chain, such as warehouses, manufacturing plants, distribution centers, and their respective capacities and locations.
- Commodity production: this involves data on the production volumes, methods, and locations of the commodities in question. It may also include information on labor practices, environmental impacts, and compliance with standards or regulations.
- Additional information: this could be any supplementary data that may help in understanding, analyzing, or optimizing the supply chain, such as seasonal trends, historical data, or economic indicators.

Depending on the country and commodity, dozens or even hundreds of heterogeneous datasets are processed. Based on the amount and quality of the data, different levels of detail of the supply chains are possible, including:

- Level (version) 1: results mostly driven by mathematical modeling given the detailed information isn't available. This level allows for basic locality resolution (e.g. municipality).
- Level (version) 2: mostly data-driven results that still require some mathematical modeling for making up for unavailable information. Also allows basic locality resolution (e.g. municipality).
- Level (version) 3: entirely data-driven results, made possible by available detailed information, which allows for precise sub-locality resolution (e.g. field, property, concession).

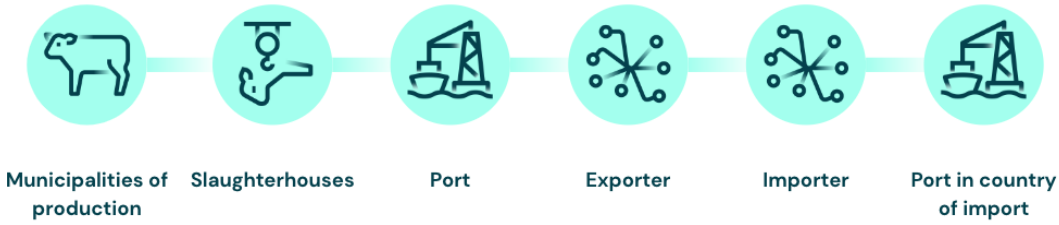

Figure 2.1: Trase actors at different points of a supply chain. Image taken from [14]

When choosing a specific country commodity within Trase.earth data tools, it is possible to see the flow between producing municipality, exporter, importer, and country of import. When selecting a specific regional or actor value, it will show its related flow, as well as related aggregated information at each step (associated hectares of territorial deforestation, trade volume, etc). It is also possible to explore the data through different categories and levels of aggregation, as well as download it. See figure 2.2 for a flow view including a map with a territorial deforestation layer, and figure 2.3 for the links related to a specific exporter.

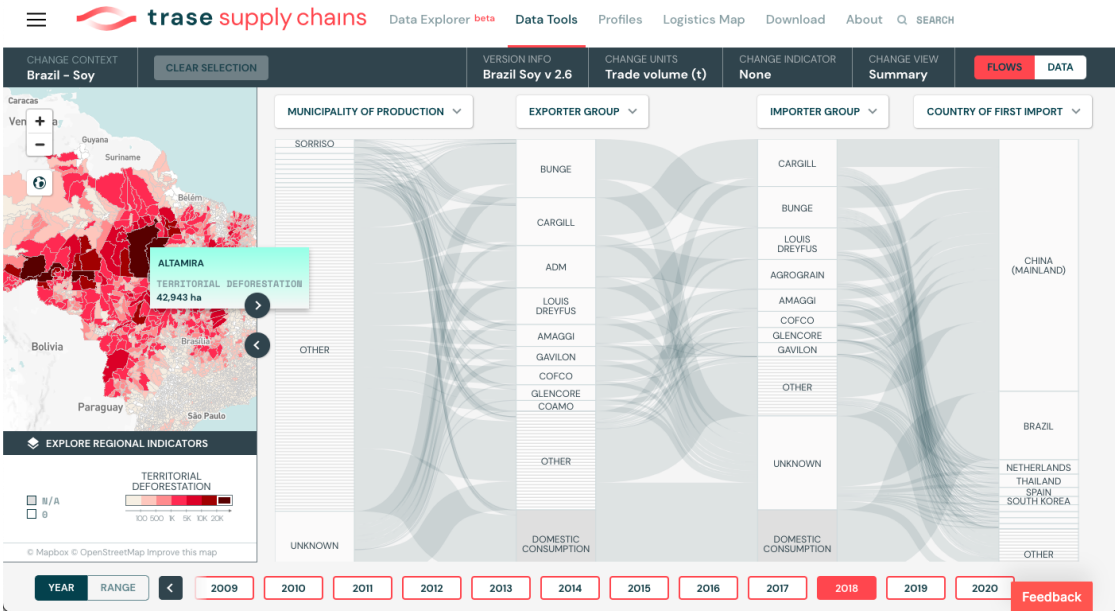

Figure 2.2: Sankey diagram showing the trade flows throughout actors in the Brazil - Soy supply chain for a given year. Allows viewing and selecting municipalities based on their territorial deforestation. Image taken from [9]

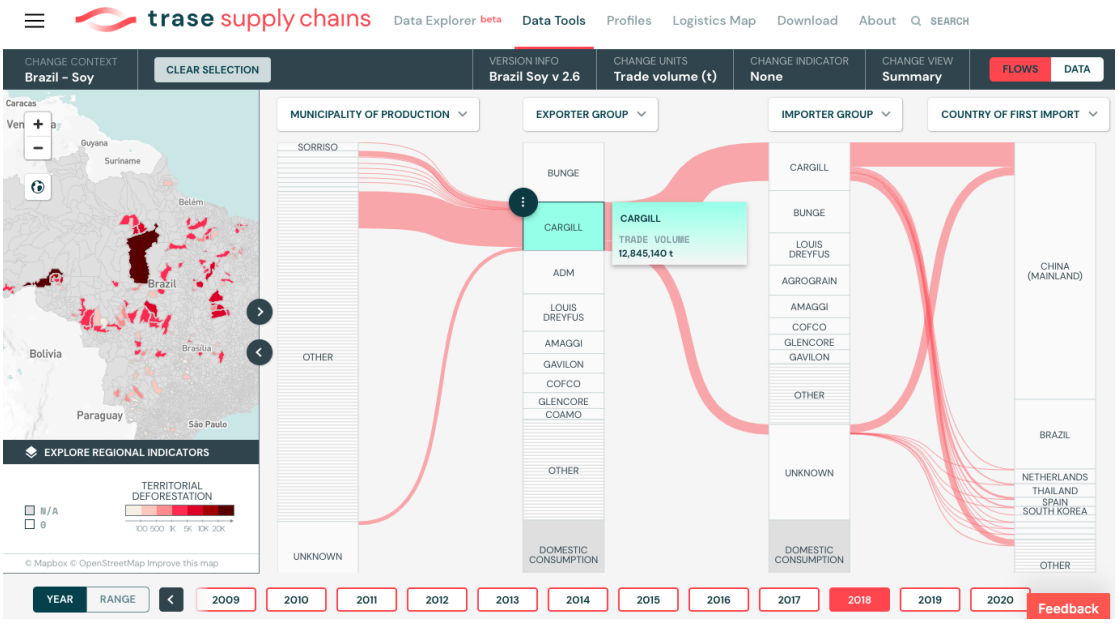

Figure 2.3: Sankey diagram highlighting the trade flow associated to a specific exporter. Image taken from [9]

Additional to the supply chain data, it is also possible to explore the exposure to deforestation from investors financing companies operating in deforestation hotspots [15]. This includes detailed information on associated asset classes, and capital flows, as well as from each financial actor. Each actor has their profile including their exposure to different commodities, financing & owners, legal hierarchy, and corporate network.

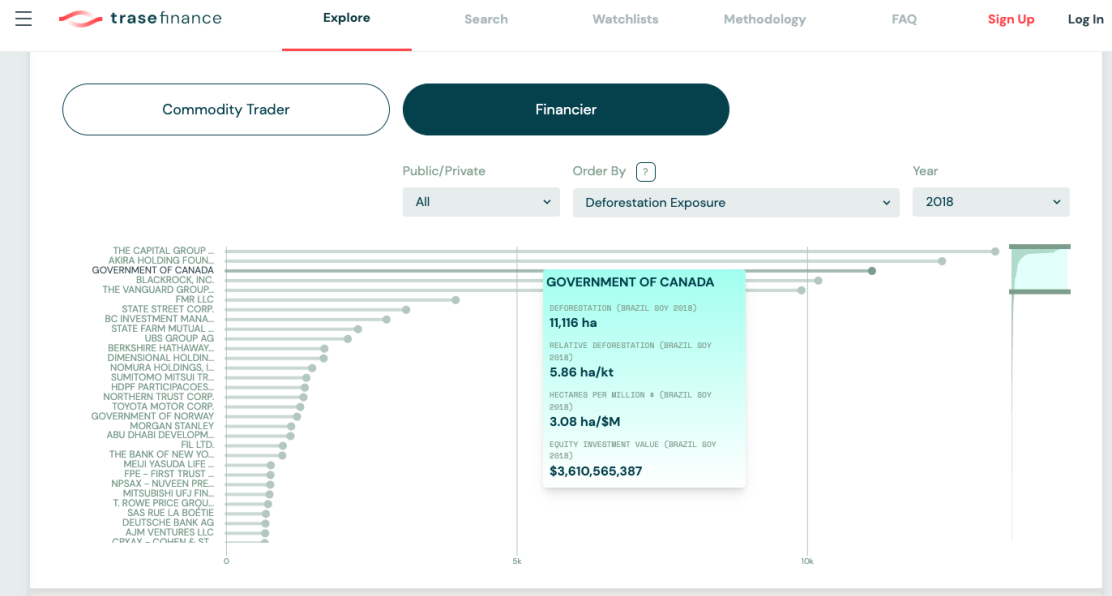

Figure 2.4: Financing actors ranked by deforestation exposure. Image taken from [15]

There are several ways of seeing aggregations of this data (see figure 2.4 for an example of ranking investors with the biggest exposure to deforestation from Brazil Soy in 2018). Currently Trase maps nearly 70% of global trade in forest risk commodities driving deforestation including soy, palm oil, beef, shrimp, cocoa, coffee, corn, wood pulp, palm kernel, chicken, cotton, sugarcane, and pork in 10 source countries including Argentina, Bolivia, Brazil, Colombia, Cote D'Ivoire, Ecuador, Ghana, Indonesia, Paraguay, and Peru (see table 2.1). Information can also be explored based on importing countries. The information generated by Trase and the methods developed has enabled multiple results in research, policy, and several specific actions, including:

- Showcase how a small percentage (usually less than 10%) of the production and producers within each commodity usually make up for most of its deforestation exposure [16]. This helps to prioritize action and provides data for actors within the supply chain to minimize their exposure to deforestation.
- Helping make the case for a new regulatory framework in Europe, including the recently approved EU Deforestation-Free Regulation (EUDR) and its requirement for companies to demonstrate that their imports are deforestation-free [17].
- Allows targeted action on illegal deforestation by public prosecutors [18].
- Providing some governments access to in-depth information to monitor and reduce their imported deforestation.

- Developing multiple insights providing a detailed examination of specific supply chains, as well as overall analysis on related topics, including trends, advances or draw-backs, and evidence-based policy recommendations, among others.
- Producing research papers and sharing detailed methods for each country's commodity, advancing a scientific and data-driven approach to inform deforestation efforts.

Country	**Commodities**
Argentina	Corn, cotton, soy, wood pulp
Bolivia	Soy
Brazil	Beef, chicken, cocoa, coffee, corn, cotton, palm kernel, palm oil, pork, soy, sugar cane, wood pulp
Colombia	Beef, cocoa, coffee, palm kernel, palm oil, shrimp, wood pulp
Cote d'Ivoire	Cocoa
Ecuador	Shrimp
Ghana	Cocoa
Indonesia	Palm oil, shrimp, wood pulp
Paraguay	Beef, corn, soy
Peru	Cocoa, coffee, shrimp

Table 2.1: Commodities currently mapped by Trase throughout different producing countries.

Information such as deforestation exposure, emissions exposure, financial flows, land use, and volumes depend on the country and commodity. The level of detail and the need to rely on mathematical modeling depends on data availability. Specific methods for country-commodities can be found on Trase.earth and Trase.finance pages and related scientific papers [19].

2.2 Data requirements and challenges

Trase methods, commodities mapped and results produced have evolved since it started in 2015, largely building on the SEI-PCS model (Spatially Explicit Information on Production to Consumption Systems) [20] and a growing team of partners, researchers, and data availability. While there is increasing awareness and regulation for reducing deforestation, transparency throughout commodity supply chains remains a challenge, despite improvements in standards and practice. The lack of visibility within supply chains makes the monitoring, reporting, and verification (MRV) of the commitments and regulations difficult to assess. Developing end-to-end traceability systems within supply

chains can be technically, financially, and legally challenging, and can take several years to implement. In this context, Trase's approach is to use existing available information (trade, in-country asset information, commodity production, among others) and apply a data-driven scientific approach to provide more transparency.
Though Trase uses a standardized approach to account for the various country-commodity supply chains mapped, they still require expertise and specialized data for each commodity and sourcing country. For example, the data for the palm oil supply chain in Indonesia shown in figure 2.5 relies on mathematical modeling and linear optimization (see the final flow allocation in figure 2.6) as there is no detailed information at each point.

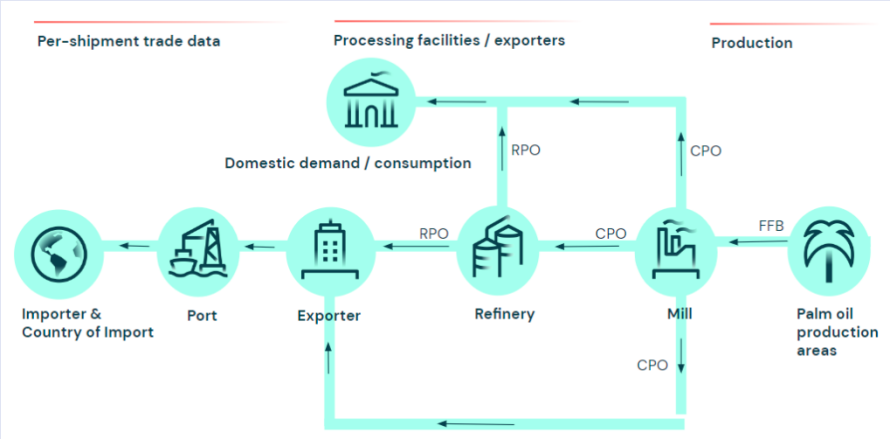

Figure 2.5: Outline of supply chain modelled by SEI-PCS Indonesian Palm Oil v1.2 [21]

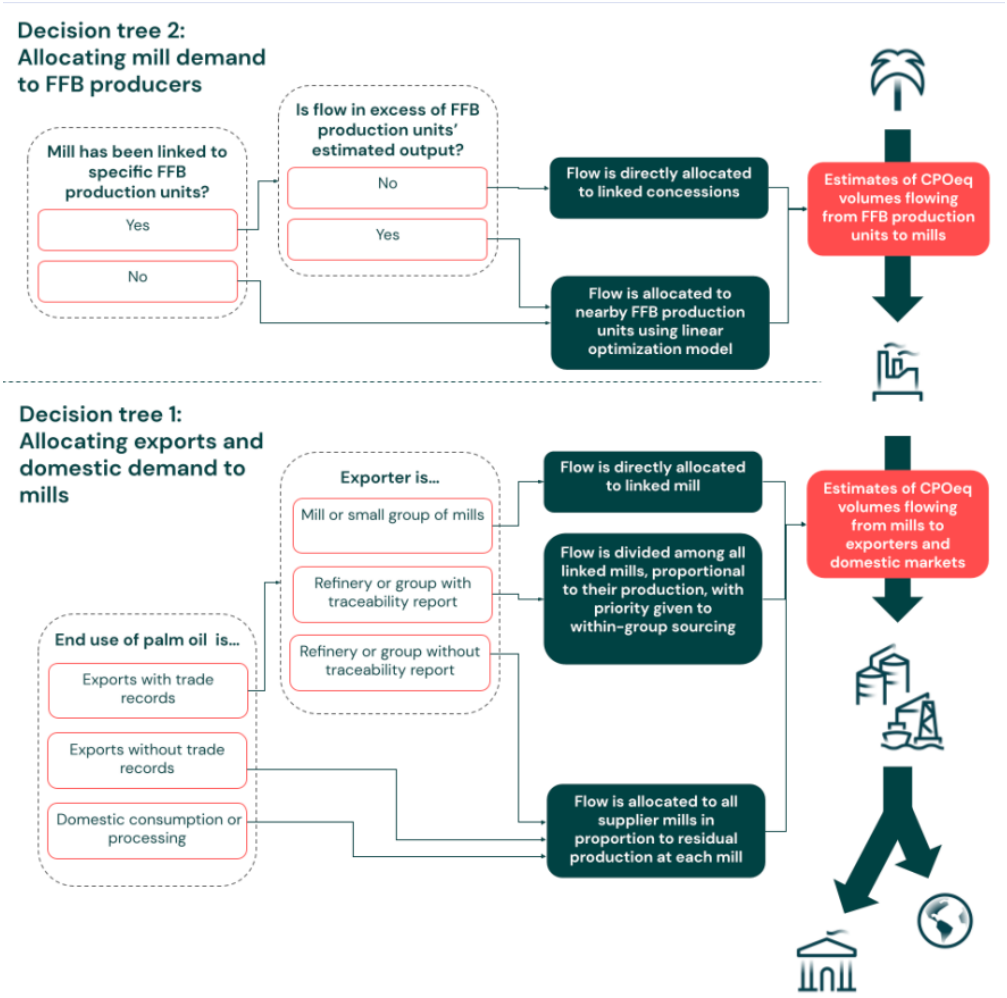

Figure 2.6: Indonesian palm oil decision tree [21]

As mentioned earlier, this complexity can imply that for certain country-commodity, dozens or even hundreds of data sources need to be processed. The process of cleaning, pre-processing, and applying basic transformations to the data to analyze and be able to use it, can take the bulk of the researcher's time. Though several data engineering and software engineering practices have been implemented in the last couple of years to manage this complexity, additional potential improvements in the 'data engineering lifecycle' are explored in this thesis. Currently, each country-commodity can take a couple of years to be developed, and up to a year or more to make general updates based on newly available data or models. Additionally, some researchers might be involved for short periods (months instead of multi-year involvement) and focus on specific contributions. They might also use different practices and conventions for managing data and code (mainly, through python, R, and SQL). Though standards, good practices, and documentation has been developed, requiring long onboarding processes to ensure proper usage of the high-level functions in Trase's codebase can be challenging given the high turnover and can limit the possibility of integrating open contributions from a wider community.

For a general overview of Trase's current situation and areas of improvement, a 'data engineering lifecycle' (as explained in the next section) is considered. The focus is placed in simplifying the data architecture, while at the same time enabling it to respond to current and projected needs. A balance between simplicity, flexibility, robustness, and scalability is sought, while responding to the following non-functional requirements:

- Compatibility with existing codebase and data: allows integrating the currently existing codebase and external data sources (collected in AWS S3 buckets), in a way that implies minimal disruption to Trase's team, while allowing the transition to new platforms and architecture if needed.
- Compatible with the current workflow: it should allow most of the current workflow to be compatible with the improvements showcased in the solutions and prototypes.
- Reduce the risk of vendor lock-in: the solution should use open standards where possible, as well as allows mechanisms for changing or opting out of specific vendors if needed.
- Flexible and interconnectable: the solutions should favor flexible and loosely-coupled components with native capabilities to connect to popular external systems.
- Keep down technologically related costs: the prototype should offer measurable cost/benefit improvements. Integrating the solutions proposed by the prototype should generate savings from some of the current technology services costs.
- Robust: allows efficient scaling up or down of data and computation requirements.

These non-functional requirements directly address the research questions posed in the introduction: how does a lakehouse architecture simplify complex data pipelines for analytical workloads, and is it possible to transition to this architecture in a way that is backward-compatible, relies on open standards, and reduces current technology costs?

3 Data engineering and pipelines exploration: lakehouse architecture

3.1 Data engineering lifecycle framework

3.1.1 An overview and historical perspective of data engineering

The 'data engineering' label has evolved dynamically over the past few decades. One of the first international events coining the term was the *IEEE First International Conference on Data Engineering* in 1984, which had the purpose of discussing "automated data and knowledge management from an engineering point of view" including topics such as "database design; data management methodologies; computer architectures for knowledge bases; technology implementation and operation for data management" [22]. As shown in figure 3.1, a fluctuating interest on data engineering can be seen in the trends of Google searches over the past 20 years, showing a 'high' interest for data engineering in the mid-2000s, followed by a steady decline up to 2017, after which it has been rising again.

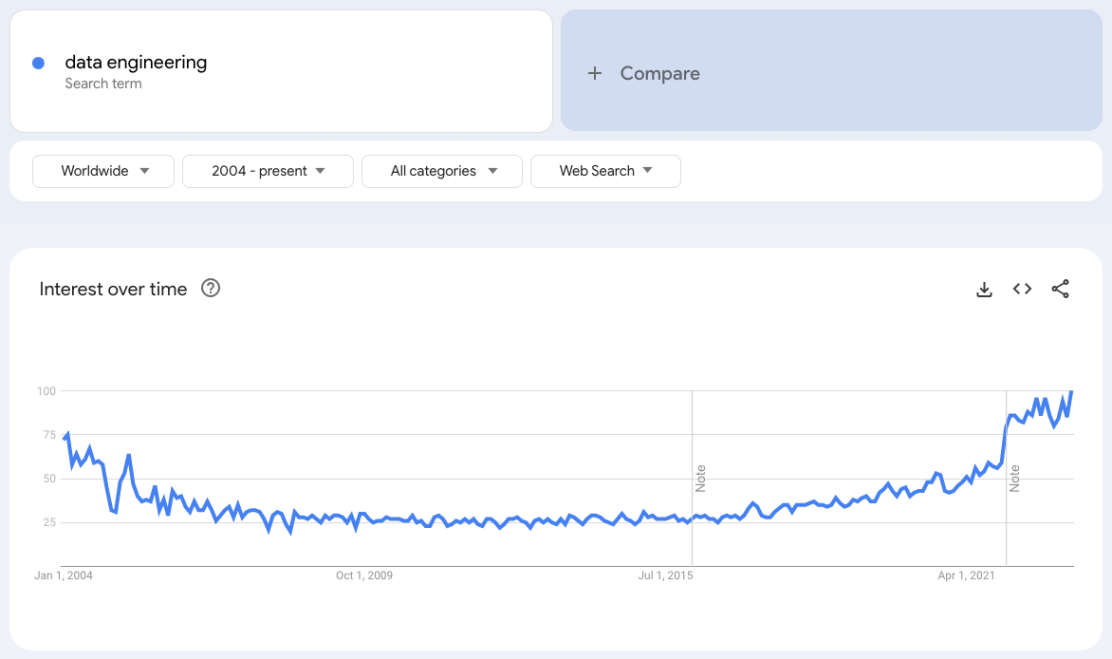

Figure 3.1: Search trend for 'data engineering' from Jan 1 , 2004 to April 2023, based on Google Trends [23]

Currently, it is common for data engineering to be considered directly related with data science. For instance, at the beginning of the O'Reilly article "Data engineering: A quick and simple definition", data engineering is defined using data science as a point of reference.

> *"As the data space has matured, data engineering has emerged as a separate and related role that works in concert with data scientists.*
>
> *Ian Buss, principal solutions architect at Cloudera, notes that data scientists focus on finding new insights from a data set, while data engineers are concerned with the production readiness of that data and all that comes with it: formats, scaling, resilience, security, and more."* [24]

Comparing the trends in interest between data science and data engineering, the view of data engineering at the service of data science is understandable, as seen in figures 3.1 and 3.2.

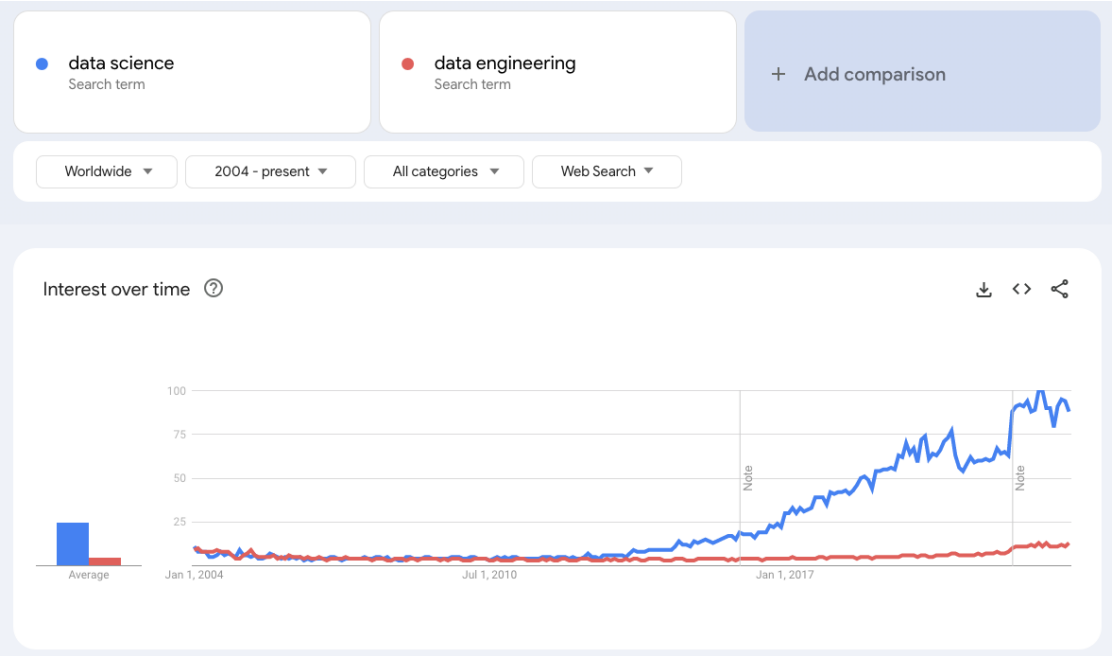

Figure 3.2: Last 20 years search trend for 'data science' and 'data engineering' [23]

According to Anaconda's *State of Data Science*, data preparation and cleansing take the biggest percentage of a data scientists' time (45% on the 2020's survey, 38% on 2022's survey) [1] [25]. Furthermore, in 2022's survey, when respondents were asked what are the most important skills and expertise missing in the data science/ML areas of their organizations, 38% marked 'engineering skills'.
In an effort to provide a comprehensive framework for the evolving field of data engineering, especially in light of the constant technological changes affecting its practice, Joe Reis and Matt Housley offer the following definition in their O'Reilly book, 'Fundamentals of Data Engineering':

> *"Data engineering is the development, implementation, and maintenance of systems and processes that take in raw data and produce high-quality, consistent information that supports downstream use cases, such as analysis and machine learning. Data engineering is the intersection of security, data management, dataOps, data architecture, orchestration, and software engineering. A data engineer manages the data engineering lifecycle, beginning with getting data from source systems and ending with serving data for use cases, such as analysis or machine learning."* [11, p. 23]

In discussing the interplay between data engineering and data science, the authors emphasized that the role of data engineering lies in managing the 'upstream' data processes. This involves providing data scientists with pre-processed and easily accessible data, thereby facilitating their work.

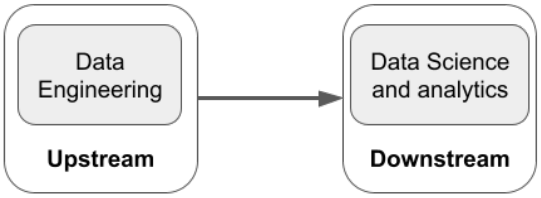

Figure 3.3: Perspective of data engineering of being responsible of providing data in a way that is readily available for data science and analytics workloads. Figure recreated from [11, p. 33]

In another reference to this idea, Monica Rogati [26] draws on a loose analogy of Maslow's "hierarchy of needs" [27] (see figure 3.4), where most of data scientist's time is be spent on the bottom three levels of the hierarchy (collecting, storing, cleaning). Consequently, Rogati calls for the importance of building a solid data foundation so that data scientists can focus on the top three levels. Anaconda's *State of Data Science* survey, both in terms of most time spent and biggest skill gap confirms the importance of data engineering as the most relevant enabler for data science.

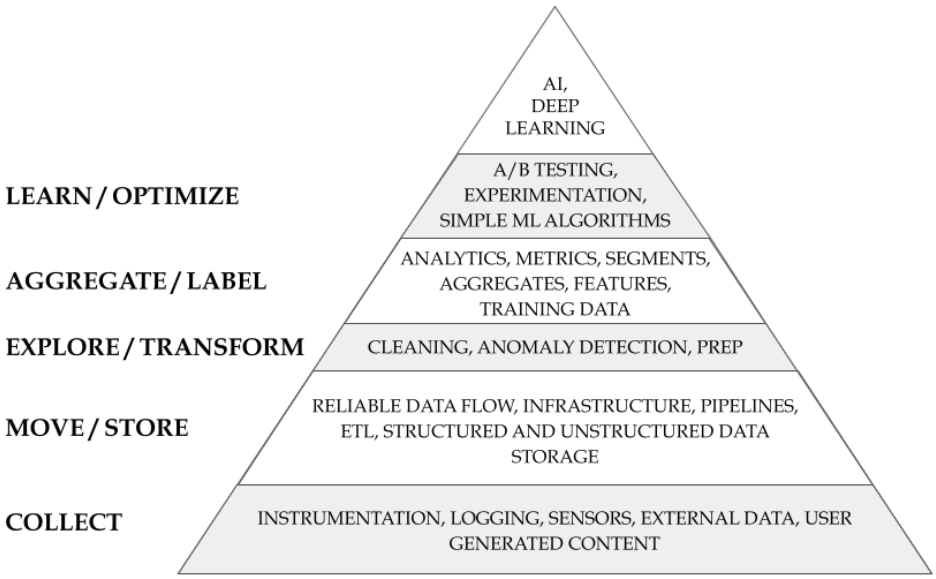

Figure 3.4: The data science hierarchy of needs. Figure recreated from [26]

3.1.2 The data engineering lifecycle

Data technologies and tools constantly improve ease of use and abstraction while allowing for flexible scalability. Some currently popular tools for managing complex data that do not fit within a single computing node have emerged within the last 10 to 15 years, such as Apache Spark [28], Apache Kafka [29], and various NoSQL databases, among others. Discussing data engineering within a framework can help analyze such trends without focusing too much on specific tools that might be replaced within a decade or two.
The data engineering lifecycle proposed by Reis and Housley [11, p. 71] is one of such frameworks and is based on the different stages data goes through from its generation until it's served to a process that generates value from it (such as data analytics, machine learning, or feeding again one of the source systems). This framework considers the progression through the following data stages:

- Generation: where the data originates from (i.e. source system), such as a device, an application, or a database.
- Storage: storage systems, usually supporting all of the lifecycle stages.
- Ingestion: gathering of the data from the source systems, be it in batches, streaming, pulling it from the source systems, or having it pushed from them.
- Transformation: cleaning, normalizing, and aggregating data.
- Serving: providing access of transformed data to create value from it.

In addition to the main stages of data generation, storage, ingestion, transformation, and serving, Reis and Housley introduce the concept of 'undercurrents'. These undercurrents act as foundational elements that support all stages of the data engineering lifecycle, serving as a bedrock upon which the other stages rely for their functionality and efficiency. Below, each undercurrent is elaborated upon for better understanding, as seen in figure 3.6:

- Security: data and access security, according to defined policies, privileges, and practices.
- Data management: policies and best practices to manage the data, including [11, p.99 - 100]:
  - Data governance, including discoverability and accountability.
  - Data modeling and design.
  - Data lineage.
  - Storage and operations.

  - Data integration and interoperability.
  - Data lifecycle management.

- DataOps: often considered as the adaptation of devOps principles to data engineering and analytics, dataOps aims to streamline the design, deployment, and maintenance of data architectures. DevOps, short for Development (dev) and IT Operations (Ops), is a set of practices that automates the processes between software development and IT teams to build, test, and release software more quickly and reliably. According to Reis and Housley [11, p. 114], dataOps is more than just a set of technologies or methodologies; it is fundamentally a set of cultural practices that foster collaboration between data engineers, data scientists, and business stakeholders. Some of the key principles of dataOps are:
  - Collaboration: dataOps seeks to foster seamless interaction between different roles involved in the data lifecycle. By bringing data engineers, data scientists, and business analysts into a unified framework, dataOps ensures that data strategies are aligned with business goals.
  - Automation: dataOps advocates for the automation of repetitive tasks within the data pipeline, from data ingestion and transformation to analytics and visualization. Automation not only reduces manual errors but also frees up valuable time for focusing on more complex, value-added tasks.
  - Continuous improvement: borrowing from Agile methodologies, dataOps encourages iterative development and continuous feedback loops. This approach allows teams to quickly adapt to changes, whether they are in data sources, business requirements, or technology stacks.
  - Monitoring and governance: effective dataOps involves a robust set of monitoring tools and practices that ensure data quality and performance metrics are consistently met. Governance policies are put in place to manage data access, security, and compliance.

- Data architecture: essentially serving as the blueprint for managing an organization's data, data architecture outlines the structure and interaction of various types of data assets within an organization. According to TOGAF - The Open Group Architecture Framework [30], it encompasses logical data assets, physical data assets, and data management resources.
  - Logical data assets: these refer to the organization and structure of data as it appears to end-users or applications, regardless of how or where the data is physically stored. Logical data assets enable a high level of abstraction and are usually represented in data models.

- Physical data assets: unlike logical assets, these involve the actual storage mechanisms like databases, data lakes, or cloud storage solutions.
- Data management resources: these are the tools, technologies, and practices employed to manage data throughout its lifecycle. This includes data governance, data quality frameworks, and metadata management, among others.

- Orchestration: coordination of jobs, including scheduling, monitoring, and their general management.
- Software engineering: including around code for core data processing, job and infrastructure orchestration, code-testing methodologies, and various frameworks and languages.

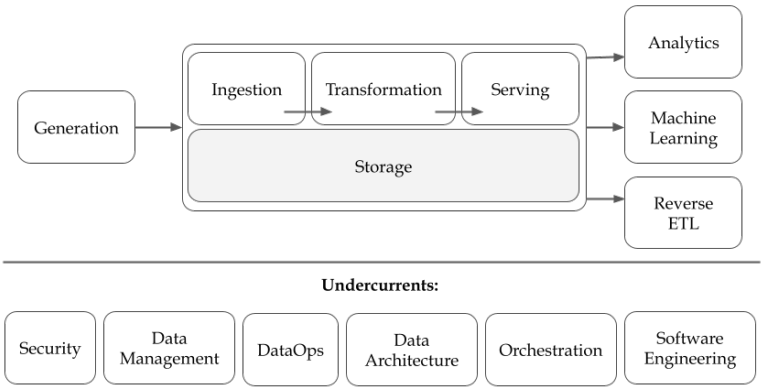

Figure 3.5: Data engineering lifecycle. Figure recreated from [11, p. 265]

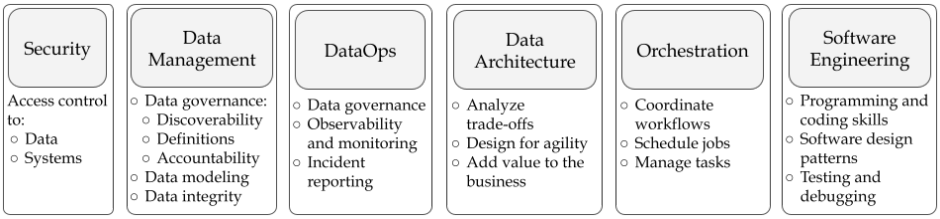

Figure 3.6: Data engineering undercurrents. Figure recreated from [11, p.97]

3.1.3 Data Engineering stages and undercurrents explored with Trase

Each stage and undercurrent within the data engineering lifecycle possesses its own associated complexity, set of practices, and relevant tools. While many of these were briefly discussed with the Trase team overseeing this thesis, certain areas emerged as offering greater value for short- and medium-term improvement. Specifically, ongoing efforts of the highest priority are focused on the following:

- Simplification of the data architecture.

- Creation of a data catalog including:
  - Data discoverability.
  - Metadata management.
  - Data lineage.
  - Quality assurance information.
- Collaborative environment for data analysis and processing.
- Quality assurance (QA) procedures.
- External sharing.

From the perspective of the data engineering lifecycle, these priorities closely relate to the data architecture and data management undercurrents, as well as to the ingestion and serving stages. Concerning data architecture, and building on the work accomplished in the preceding project course [13], a key component is a lakehouse architecture, as described in [2] and implemented via the Databricks Lakehouse platform. This architecture, further detailed in the following section, significantly influences the solutions this thesis explores.
Explorations take place to review solutions that align with Trase's priorities from a data engineering perspective. Specifically, this report discusses several working prototypes developed in relation to these priorities, as seen in figure 3.7. Each prototype employs sample data and procedures currently in use at Trase, facilitating an analysis of the suitability, performance, costs, and overall implications of potential integration into Trase's workflow.

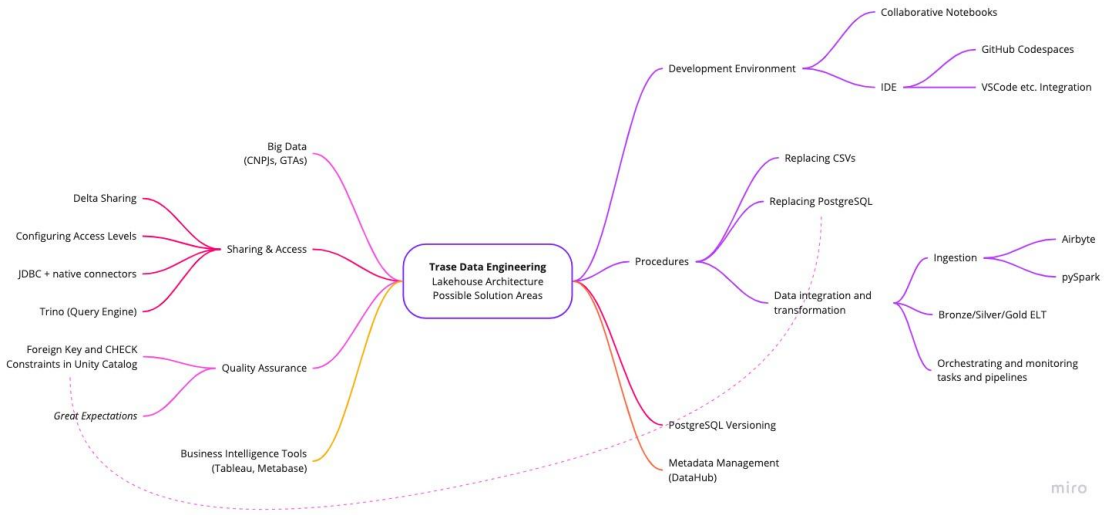

Figure 3.7: Mind map developed by Trase's team summarizing the different explorations conducted in relation to the lakehouse architecture proposed in this thesis

3.2 Data architectures and the lakehouse

According to The Open Group Architecture Framework (TOGAF) [30], a data architecture describes the structure and interaction of an organization's main types and sources of data, logical data assets, physical data assets, and data management resources. In this section, the focus is on the technical aspects of data architecture, rather than operational considerations such as process design and team organization. The technical side outlines the components of how data is ingested, stored, transformed, and served along the data engineering lifecycle.
Data architectures serve as a foundation for data management and analytics. The possibilities, technologies, and workflows within a data pipeline are strongly dependent on the underlying data architecture. Since the 2010s, a prevalent architecture for data analytics applications that manage and analyze large volumes of data — ranging from thousands to millions of datasets or spanning terabytes to petabytes in size — and draw from heterogeneous data sources, is the combination of a data lake and a data warehouse. We will explore this architecture, as well as the emerging lakehouse architecture. The examination will extend to Trase's current setup and delve into the potential benefits and considerations involved in transitioning to a lakehouse architecture. Such an infrastructure will not only enable but also lay the groundwork for the review and development of working prototypes pertinent to the overall data pipelines of Trase.

3.2.1 Data lake

A data lake is a centralized repository designed to store a vast array of raw, unprocessed data from various sources. This allows for a flexible environment for data exploration and analysis. In contrast to traditional data storage methods, data lakes are built to handle diverse data types — ranging from structured to semi-structured and unstructured data. One of the defining features of a data lake is that it allows for a 'schema-on-read' approach, where the data schema is not defined until the data is read. This contrasts with 'schema-on-write' systems, where data must conform to a predefined schema before being stored. Additionally, data lakes facilitate the ingestion of data in its native format, providing high levels of scalability. This makes it easier to accommodate large volumes of data, whether it's streaming in real-time or being batch-loaded from multiple sources, all stored in their natural format. With schema-on-read, there's no requirement to predefine data structures, schemas, or transformation processes, allowing for greater agility in managing and utilizing data [31].
The 2010s witnessed the emergence and rapid adoption of cloud data lakes leveraging the capabilities of cloud computing platforms, such as Amazon Web Services (AWS) S3, Google Cloud Storage, and Microsoft Azure Blob Storage, to provide scalable, cost-effective, and highly available storage for vast amounts of data. Unlike traditional on-premises data lakes, cloud data lakes offer several advantages, including elastic storage capacity, on-demand computing resources, and seamless integration with other cloud

services. This shift to cloud-based data lakes has enabled organizations to overcome the limitations of on-premises infrastructure, accelerate data processing and analysis, and leverage capabilities offered by cloud platforms. Organizations can then pay only for what they use, and avoid the costs of setting up and maintaining the infrastructure.

3.2.2 Data warehouse

The data warehouse concept has been around since the end of the 1980s and refers to a structured repository that integrates and organizes data from disparate sources into a unified, consistent format, serving as the single source of truth. It follows a schema-on-write approach, transforming and aggregating data to provide a structured view suitable for business intelligence, analytics, and data science. Data warehouses typically support complex analytical queries and ensure data consistency and quality. Bill Inmon, commonly credited for coining the term, defines it in the following way:
"*A data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions. The data warehouse contains granular corporate data. Data in the data warehouse is able to be used for many different purposes, including sitting and waiting for future requirements which are unknown today*" [3].
Inmon defines four characteristics a data warehouse must have:

- Subject-oriented: the data warehouse is structured according to specific subject areas, such as sales or marketing.
- Integrated: data from disparate sources is consolidated and normalized.
- Nonvolatile: data remains unchanged after data is stored in a data warehouse.
- Time-variant: varying time ranges can be queried.

After the emergence of cloud data lakes, cloud-based data warehouses designed to easily connect to them as data sources appeared in the early 2010's. The current leading cloud data warehouses [4], such as Amazon Redshift [6], Google BigQuery [5], Microsoft Azure Synapse Analytics [8], IBM Db2 Warehouse on Cloud [32], and Snowflake [7], have become popular due to their ease of use and cloud-native capabilities. These cloud services eliminate the need for complex on-premise setups and maintenance, and they offer flexible scalability and a usage-based cost structure. An advantage of these cloud data warehouses is their inherent ability to integrate with a multitude of data sources. They are not only capable of connecting with data lakes, but they can also interface with various databases, APIs, and external services. This interconnectivity is facilitated by a suite of tools and client libraries provided by these platforms, which supports a variety of programming languages and data formats.
Furthermore, these warehouses provide a rich set of services and tools that assist in the data ingestion and normalization processes. They offer built-in functionalities for data

transformation, allowing organizations to handle diverse datasets and prepare them for analysis. The availability of pre-built connectors, data adapters, and ETL (extract, transform, load) services simplifies the process of bringing in data from different origins and formats, ensuring that data analysts and scientists can focus on deriving insights rather than managing data logistics. The flexibility to ingest data in real-time or in batches complements the varied analytical needs of businesses, making these cloud data warehouses a central component in modern data strategies.

3.2.3 Data lakehouse

A data lakehouse merges the best of both worlds: it retains the expansive storage capabilities of a data lake while introducing the structured querying features of a data warehouse, as illustrated in figure 3.8 [2]. This architecture provides flexibility to access and analyze unrefined data without complex schemas or preliminary transformations, implementing schema-on-read to tailor data structures at the time of analysis.
Furthermore, the lakehouse adds capabilities for doing on-the-fly schema definition, maintaining data integrity through transactions with ACID (atomicity, consistency, isolation, and durability) transaction capabilities [33]. This enhances security, and enables efficient updates and deletions. It is also optimized for SQL queries and business intelligence tools [13], facilitating a seamless workflow from exploratory analysis to systematic reporting and simplifying operations by giving direct access to both raw data and ACID-compliant tables. This supports analytical tasks and common machine learning operations without the need for separate data transfer and preprocessing.
Leveraging scalable storage, lakehouses enable parallel data processing and provide decoupled compute resources, which can be scaled on-demand for cost-effective and agile operations.
In contrast to traditional cloud warehouses that rely on ETL processes to import data from lakes, as shown in section b of figure 3.8, lakehouses maintain a metadata layer on top of the data storage. This layer, often composed of Parquet or Avro files, adds to them a transaction log. It also employs a Hive metastore [34], which is a centralized repository for storing the structure and metadata of database tables (such as columns, types, and table location), to manage the metadata of each data asset. The use of a Hive metastore, which will be better described in the next chapter, enables efficient data management and supports the schema-on-read capability that allows for direct querying and analysis of data in situ. This integration combines the performance and governance of conventional warehouses without redundant ETL steps, as shown in section c of the figure.

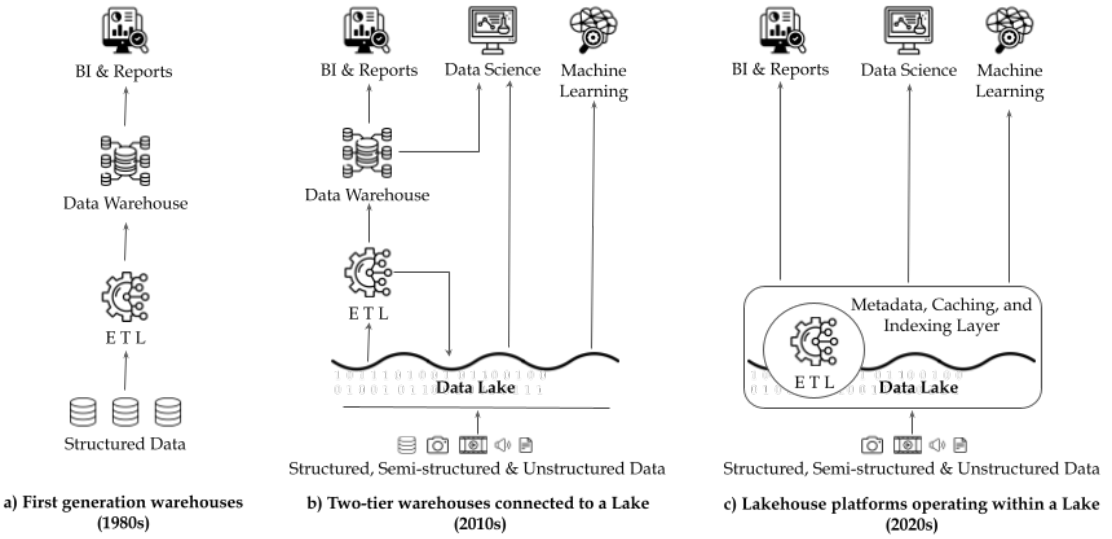

Figure 3.8: Progression from warehouse to lakehouse architectures. Figure recreated from [2, Figure 1] with supporting images from Flaticon.com

3.2.4 Trase's current architecture

While Trase does not currently rely on a data warehouse, it has other loosely coupled mechanisms (see figure 3.9) to support some of its features, including:

- Subject-oriented and integrated single source of truth: a PostgreSQL database is used for providing a centralized, normalized source of truth, especially for:
  - Trader hierarchies (label, trader, group).
  - Region names, levels, hierarchies, codes, geometry.
  - Commodity equivalence factors.
  - Logistics maps.
  - Model results
  - Spatial metrics in tabular form.
- Nonvolatile and time-variant:
  - File data sources in S3 have versioning enabled to keep historical versions of datasets.
  - Access and exploration of current and historical versions, through a third party service called Splitgraph [35]. This service main purpose is to enable wider public access to some of Trase's datasets, and as an endpoint to Trase's web applications. It does so by replicating a selection of data from the PostgreSQL database.

Though the current configuration allows for flexibility and relies on tools that most researchers use confidently, it also makes some processes intricately complex and non-standard, sometimes jumping back and forth between S3, pre-processing and modeling locally and/or in JupyterHub, and PostgreSQL. Though coding standards, high-level functions, documentation, and tutorials have evolved to try to manage this complexity, this still remains challenging.
Although a cloud data warehouse could manage these complexities more effectively and introduce additional services, it would also require the complexity of a two-tier architecture. Additionally, while a cloud data warehouse might add value, it does not fully align with several of the non-functional requirements established by Trase, which are pertinent to the main research question. These include compatibility with the existing codebase and data, congruence with current workflows, and minimizing vendor lock-in, all of which are crucial considerations for the proposed lakehouse architecture in simplifying complex data pipelines for analytical workloads. The question remains: can a transition to this architecture be implemented in a backward-compatible manner, while also adhering to open standards and curtailing current technological expenditures?

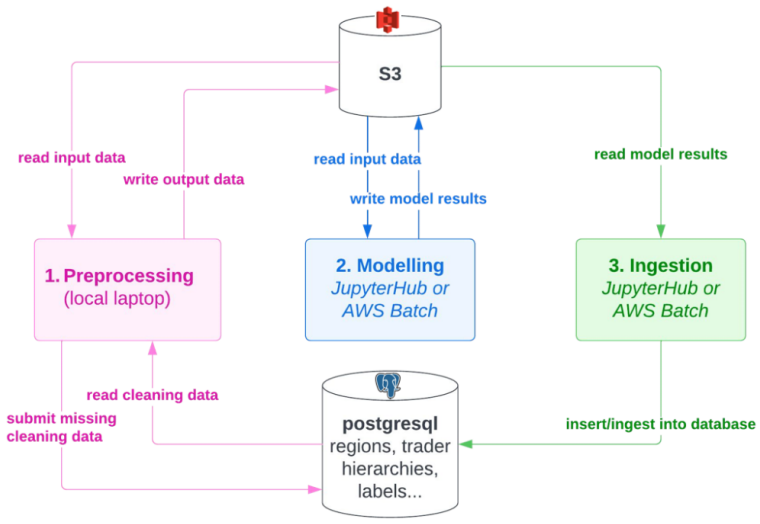

Figure 3.9: Current Trase's main flows of data

3.2.5 Lakehouse architecture implementation

To account for the different Trase requirements, several lakehouse architecture configurations are explored. The explorations are based on the Databricks platform, the leading lakehouse cloud service, which to a large extent have coined the term. Other non-vendor open-source solutions are also considered, and although only some of their components are tested, no full working prototypes are developed. They will be reviewed in the closing sections. Also, some tests are made where Databricks is replaced by open-source ones, especially through a Trino [36] query engine. Trino is an open-source distributed SQL query engine designed for running interactive analytic queries against data sources

Figure 3.10: Delta Lake main features. Figure recreated from [40] with supporting images from Flaticon.com

of all sizes ranging from gigabytes to petabytes. It was formerly known as PrestoSQL [37] and enables querying data where it lives, including structured and semi-structured data across multiple data stores.

Databricks Lakehouse

Databricks, established in 2013 by the creators of Apache Spark from UC Berkeley's AMPLab [38], initially gained recognition for its hosted Apache Spark service. Apache Spark [28] itself is an open-source, distributed computing system that offers an interface for programming entire clusters with implicit data parallelism and fault tolerance. Databricks leveraged this technology to provide a notebook-oriented environment in the cloud, enabling data scientists to utilize spark's powerful data processing capabilities without the complexity of configuring and managing computing clusters.
Continuing with the idea of building technologies that enable the separation of computing and storage for big data processing, Databricks developed Delta Tables (see figure 3.10), an open-source storage layer that can run on an existing data lake (e.g. S3) while guaranteeing ACID transactions. Delta Tables, which store data based on a Parquet file format, further include transaction logs, allowing versioning and concurrency. They also integrate other features such as automatic data layout optimization, upserts, caching, audit logs, and the possibility of access by multiple compute engines such as spark, Hive, Presto/Trino, and Redshift, among others [39] . Databricks has named this framework 'Delta Lake' and has open-sourced its core components [40].
The components of Delta Lake enable the lakehouse architectural pattern, offering comparable functionality to that of a data warehouse while also allowing for direct access to various data formats —whether structured, semi-structured, or unstructured— thereby providing inherent support for machine learning and data science tasks [2]. Databricks Lakehouse has also demonstrated leading performance on industry-standard benchmarks, such as the TPC-DS power score [41]. This specific measure assesses the processing power of a system by evaluating its capability to efficiently execute a

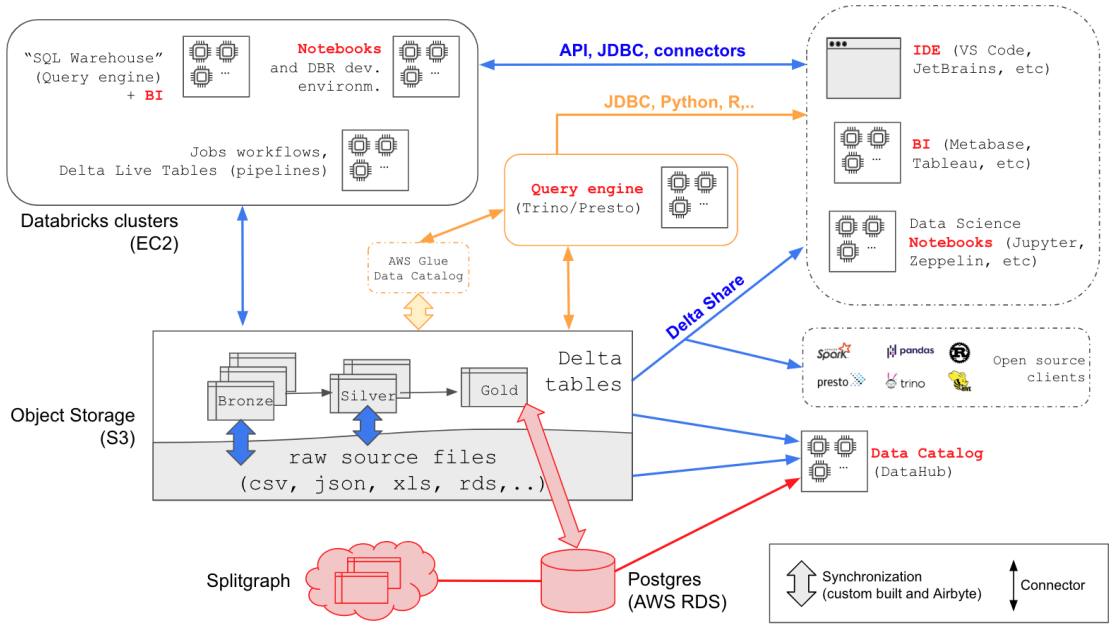

Figure 3.11: Sketch of a lakehouse implementation in Trase allowing for backwards compatibility

wide array of complex analytical queries, reflective of real-world data management and analysis workloads. Additional to performance, Databricks Lakehouse allows for a lower total cost of ownership, benefiting from a more pronounced separation of computing and storage resources. The integration of data lake and Delta Lake concepts, along with a metastore that ensures proper governance of data assets (which will be further elaborated in a subsequent section), forms the essence of Databricks' 'Lakehouse Platform' offering [42].

Working prototype: Databricks Lakehouse implementation

The different working prototypes rely on a Databricks Lakehouse while simultaneously allowing for backward compatibility of current Trase systems, as seen in figure 3.11. Though eventually some of the systems Trase currently uses could be deprecated (for example PostgreSQL and Splitgraph, marked red in the image), the architecture should allow for an incremental and flexible transition.

In the setting tested, Delta Tables replicate the data contained in the CSV source files in S3 object storage, as well as from a PostgreSQL database (lower left). Notebooks running in a Databricks-managed cluster (deployed with AWS EC2 nodes, top left) handle these replications, using PySpark to do the ingestion. For CSVs it automatically corrects common errors in the headers, and stores aside malformed records for further review. Methods have been developed for flexible ways to handle the replication such as replicating all existing CSVs in S3. Also, a Notebook is created to replicate specified schemas and tables from PostgreSQL and identify only the changes between replication runs, so to allow for data versioning even in the absence of a Primary Key or Cursor

commonly required to make logical replications. PySpark is also used to read and process large tables efficiently (5GB tables with millions of rows took between 1 and 2 minutes when processed with 1 or 2 computing nodes). The notebooks allow ingesting the whole S3 or PostgreSQL database, or only a subset of countries, commodities, or databases, while automatically mapping the datasets to a `<catalog>.<database>.<table_name>` three-level namespace (e.g. '`brazil.soy.bronze_deforestation_co2`'). The notebooks can be configured to run as job workflows to be run manually when triggered by an event (for example if detecting a change within an S3 bucket), or scheduled. Resulting Delta Tables can be further explored, visualized, and transformed within other notebooks, or through integrated SQL-based query and dashboard BI tools.
Additionally, end-to-end pipelines can be defined and managed using job workflows and Delta Live Tables, including the possibility of combining batch and streaming processes, defining relational data quality expectations, and checking real-time for data health quality.
Delta Tables can also be accessed or written through external tools (top right of the image). Though popular tools have native connectors to Databricks, there is also a REST API, JDBC drivers, a CLI (command line interface) tool, a python API, and python and Go SDKs. For accessing Databricks through a Databricks managed computing cluster, for example for connecting through JDBC or to the Databricks Runtime environment (DBR) from an IDE, an access token associated with a Databricks user or a service principal has to be generated. Another option is accessing via Delta Sharing, which enables direct read access to the tables in S3 without requiring a Databricks cluster. As Databricks and its underlying storage and computing resources (AWS, GCP, or Azure) charge on a pay-as-you-go basis, it only requires the data-egress fees of reading from the object storage (e.g. S3) when the data is read from outside the associated cloud service. Delta sharing is also useful for sharing with outside organizations and requires a credential file instead of a string token.
Lastly, the open-source Acryl DataHub Data Catalog [43] is implemented (depicted in the bottom right of the figure), facilitating data observability and discovery alongside comprehensive metadata management. This platform not only supports the addition of custom metadata and descriptions but also enables schema inference upon reading source files in S3. It offers data profiling to summarize statistics at the column level, provides visibility into data lineage to trace data dependencies, ensures quality with assertion tests, and features tools for organizing data assets into categories and tags as well as overseeing their governance.
To circumvent a potential lock-in with Databricks, or to integrate another open-source solution, a setup utilizing the Trino query engine [36] is explored (indicated in yellow within the image). In such a setup, Trino is tasked with querying Delta Tables, while the Databricks platform manages governance tasks and the infrequent write and delete operations.

For more decentralized and nimble configurations, tools such as Delta-rs [44], PyArrow [45], and DuckDB [46] are viable. Delta-rs offers rust [47] and python bindings for Delta Lake, providing a lightweight alternative to manage Delta Lake files without spark. PyArrow is a cross-language development platform for in-memory data, offering columnar storage optimized for analytical operations, which translates to rapid data access speeds. DuckDB, on the other hand, is an in-process SQL database focused on analytical workloads (OLAP) and designed to run efficiently.
Additionally, these technologies can work in tandem with open-source object storage solutions such as Minio [48], which delivers an S3 compatible API for object storage, and Swift [49], which is part of the OpenStack [50] suite for cloud storage. These integrations allow for flexible deployment models, including on-premise and hybrid cloud/on-premise lakehouse architectures, thereby broadening the scope for organizations to build a system that aligns with their specific needs and minimizes vendor dependency.
Although the solution including a Trino query engine is developed and tested, it is not described in detail, as for Trase's setting in particular it doesn't prove to be a cost-effective solution: reading Delta Tables without requiring a Databricks computing cluster can also be achieved through Delta Share. Also, currently, Trino does not connect to Databricks Unity Catalog [51] metastore (description in a later section), and cannot query specific history versions of a dataset. It also requires setting up a Hive metastore compatible service (AWS Glue Data Catalog [52] is used for this purpose), and registering the Delta Tables with their corresponding S3 locations to it, which requires additional synchronization efforts (and thus, more complexity).

3.3 Data pipelines

Data pipelines refer to the end-to-end systems that enable data flow from various sources to final target destinations. It involves designing, orchestrating, and maintaining the systems that allow the movement, transformation, and integration of data, ensuring its quality, reliability, and timeliness.
Data pipelines are viewed within the context of a lakehouse architecture, which has implications on how the metadata of the data assets is managed, how data is ingested, transformed, and consequently how pipeline management, in general, is carried on.

3.3.1 Data metastores: Hive metastore and Databricks Unity Catalog

Hive metastore

A lakehouse architecture requires a metadata management system, called a metastore. A metastore is a centralized repository that stores metadata information about data stored in a data lake or other data storage systems. It provides a catalog of data objects, such as tables, columns, partitions, and schemas, and their relationships, making it easier to understand and work with the data in the lake. It also allows referencing data assets based on different formats such as Parquet, ORC, AVRO, CSV, or JSON.
One of the first and most popular metastores, built initially for Hive [34] and allowing for usage with object stores such as S3, Google Cloud Storage, Azure Blob Storage, is the Hive metastore. It was created to act as a catalog for Hive's underlying data storage, enabling the separation of metadata and data storage, and providing a logical view of the data stored in a Hadoop Distributed File System (HDFS). With time, the Hive metastore started to be used as a metadata management system not necessarily linked to the Hive runtime (allowing for Hive Query Language - HQL) or to Hive's storage component (HDFS). For example, it is used in this way by spark, Trino [53], Drill [54], Impala [55], as well in services such as AWS Glue Data Catalog [52]. Some of these services extend on the original functionalities of the Hive metastore to provide a wider range of services. For example, the AWS Glue Data Catalog, based on the Apache Hive metastore, also allows referencing data sources in RDS [56], Redshift [6], and other JDBC/ODBC-compatible data stores.

Databricks Unity Catalog

While Databricks initially relied on a Hive metastore for its metadata management system, and still offers support for it as a 'legacy option', it started supporting its own metastore, called Unity Catalog in 2022 [51]. It offers some enhancements of the Hive metastore to make it easier to have a unified governance of data assets, not only of tabular data and files, but also of dashboards, machine learning models, and unstructured data like video and images, as well as streaming data sources. The Unity Catalog is hosted by Databricks centrally instead of deployed alongside each Databricks

account workspace as with its legacy hive metastore, it allows for more flexibility around organizing, governing, as well as sharing the data.
In terms of organization, Databricks Unity Catalog uses a three-level namespace to organize data. Within each Unity Catalog metastore (to which different Databricks workspaces can have access), these levels are:

- Catalogs: the higher level, which contains a collection of schemas (also referred to as databases).
- Schemas (databases): a collection of tables.
- Tables: tabular data.

In terms of governance, catalogs, schemas, tables, and other data assets (e.g. files, dashboards, ML models), permissions for managing them (create, delete, alter), or read/write access can be defined using SQL-like `GRANT` declarations. For tabular data, these permissions can be specified at the column level (just allowing access to certain fields) or the row level (allowing access to records that match certain `WHERE` conditions).
This flexibility allows for easy data sharing of organized data assets with granular access permissions, be it internally or to external users. A working prototype showcasing this is described later in this chapter.

3.3.2 Ingestion, and ETL vs ELT

Data ingestion is the process of importing data from various sources into a data storage or processing system. It is a crucial step in the data engineering pipeline, as it sets the foundation for all downstream data processing tasks. Proper data ingestion ensures that data is accurate, complete, and consistent.

ETL (extract, transform, load) and ELT (extract, load, transform)

ETL (extract, transform, load) and ELT (extract, load, transform) are two common approaches to data processing. ETL involves extracting data from various sources, transforming it into a common format, and loading it into a data warehouse or data lake. ELT, on the other hand, involves extracting data from various sources, loading it into a data storage system, and then transforming it as needed.
The ETL approach has been widely used in traditional data warehousing, where data is processed in batches, and after being transformed it is stored in a centralized repository. This can help make the warehouse or central repository have better-structured information. However, ETL can be time-consuming, complex, and inflexible when dealing with large amounts of unstructured data or where there are schema changes. For instance, if eventually there is a change in a transformation or if an additional source data wants to be considered or traced back to it for analysis or quality assurance. Also, it can make it slower to do exploratory data analysis, as first it might be expected that the data should

be cleaned and an initial 'correct' schema defined (schema-on-write). Usually, relational databases such as MySQL [57], Oracle [58], SQL Server [59], and PostgreSQL [60] require this approach.
ELT, on the other hand, is a more flexible approach that facilitates real-time processing of unstructured data. It allows for storing data in its original format, without the need for transformations before loading it into a central repository. ELT is particularly useful when dealing with large amounts of data from different sources, as it allows for greater scalability and agility in data processing. Usually, it allows for schema-on-read where there is no need to do data modeling or schema design upfront, but having the system do this automatically as the data is being processed or retrieved. This approach is commonly used in big data processing systems like Hadoop [61], spark, and NoSQL databases like MongoDB [62].

Spark and Databricks Lakehouse for ELT

Additionally to scalability and performance to process large amounts of data in parallel (which though are some of their key features, are not some of the current priorities in Trase), spark and Databricks allow for flexibility and a unified data platform that is simple and cost-effective to manage.
In terms of flexibility, spark and Databricks Delta Tables are equipped to manage both structured and unstructured data across various file formats. They provide robust support for semi-structured data, including CSV [63] and JSON [64] — the formats most commonly utilized by Trase. These systems offer the capability to infer schemas and provide a range of options for dealing with badly formatted records—such as setting them aside, dropping them, or failing the entire process. Additionally, they handle null and NaN values, encoding discrepancies, and include features specifically designed for navigating JSON structures.
Databricks in particular also has an Auto Loader function [65] that allows to incrementally processes new data files as they arrive in cloud storage, including schema inference and schema evolution (allowing for updating the schema in case new columns start to appear in source files).
For the current thesis, a working prototype for ingesting and managing intricate CSV sources is developed.

Medallion data pattern: bronze, silver, and gold

Medallion architecture is a data engineering pattern that involves organizing data processing workflows into three distinct stages: bronze, silver, and gold. As data gets processed in these stages, it should be organized or have a prefix name included that allows it to identify which stage it corresponds to easily.
The bronze stage involves collecting raw data from various sources and storing it in a data lake or data warehouse. The data is usually unprocessed and may contain

inconsistencies, errors, or duplicates. Here, the schema is usually inferred and might change over time. When there are badly formed records, they could be dropped or set aside in a column or log to be analyzed if needed, or potentially corrected and integrated. The silver stage involves cleaning and transforming the data, removing any inconsistencies, and enriching it with additional information. The gold stage involves aggregating and analyzing the cleaned and transformed data to generate insights and inform business decisions. This stage typically involves data modeling or machine learning to identify patterns, trends, and make predictions.
The medallion architecture provides a framework for managing the complexity of data processing by breaking it down into smaller, more manageable stages. By separating the data into distinct stages, it becomes easier to manage, analyze, and secure. It also allows for better collaboration between teams, as each team can focus on their specific area of expertise. This provides a structured approach to data processing that facilitates data quality, consistency, and accuracy, while also enabling greater agility and flexibility.

Ingestion and management of intricate CSV sources

Most of the data sources the Trase team currently works with (trading data, tax information, business registrations, and sanitary inspections, among others), are saved in a CSV format. When those CSV data sources are pre-processed for basic cleaning, normalizing, and transformations, the results are also saved as CSV files. Of the approximately current 10,000 CSV files in Trase's S3 main storage, 90% of them are below 5 megabytes of size, and 99% below 100 megabytes.
There have been considerable improvements in Trase in the last two years. Common practices have been established, and helper functions have been developed within Trase's codebase to manage the CSVs in a more standardized way, including having a preferred delimiter format, and following basic conventions in where to save them within an S3 bucket (following a `'country/commodity/topic/'` path prefix convention). However, there are still several challenging issues, including:

- The S3 folder structure is complex and with heterogeneous conventions, as seen in figure 3.12.
- The more than 10,000 CSV files are disseminated throughout this varying structure.
- There are different conventions for separators, quotes, and null values between CSVs.
- Some CSVs don't follow ANSI-SQL standards for table column names. For instance, some column names include special characters such as dashes, spaces, carriage returns, among others, as seen in figure 3.13.
- Several CSVs have malformed rows, including varying number of fields between records, or a bad use of escaping characters.

- Some multi-gigabyte CSVs require special processing each time they need to be used, or have aggregation and joins based on them, especially when used from single-node environments such as with regular python pandas.
- Many CSVs include information for only a specific year, which usually implies loading several files and joining them to make multi-year aggregations.
- Several data sources containing nested data are saved as separate CSVs, which then need to be joined when processing them, adding complexity.

```
ecuador
|-- logistics
|   `-- shrimp_processing_facilities
|       |-- acuaculturaypesca
|       |   `-- 3-cleaned
|       |       |-- ACTUALIZA_PROCESA_ACUACULTURA_12OCTUBRE2017.csv
|       |       |-- ACTUALIZA_PROCESA_PESCA_ACUA_22DICIEMBRE2017.csv
|       |       |-- tabula-ACTUALIZA_ACUACULTURA_31AGOSTO2018.csv
|       |       `-- tabula-ACTUALIZA_PESCA_ACUA_13AGOSTO2018.csv
|       |       `-- tabula-ACTUALIZA_PESCA_ACUA_24OCTUBRE2017.csv
|       `-- out
|           |-- china
|           |   |-- CHINA_matches.csv
|           |   `-- CHINA_non_matches_cleaned.csv
|           |-- ec_processing_facilities.csv
|           `-- eu
|               `-- originals
|                   `-- FFP_EC_es(Establecimientos_EC).csv
`-- production
    `-- crop_maps
        `-- production
            `-- shrimp_pond_maps
                `-- out
                    |-- ec_production_per_parish.csv
                    |-- ec_shrimp_area_per_parish.csv
```

Figure 3.12: Example of file organization in S3. Deeply nested files with different file name and substructure conventions

```
NUMBER;CÓDIGO;NOMBRE;"DIRECCIÓN^MESTABLECIMIENTO";TELEFONO;CONTACTO
```

Figure 3.13: Example of a malformed CSV: the header has a carriage return (^M) in the middle, has accents, and uses quotes in only one field

While spark has native functionalities to infer the schema of CSVs, they do not work for non-compliant ANSI-SQL column names that include special characters such as carriage returns, it requires specifying the default column delimiter in case it's not separated by commas, as well as any special convention used for specifying null values and for escaping character sequences.
To account for all the previous challenges while integrating an ELT and medallion data pattern, a couple of working prototypes have been developed to ingest existing heterogenous CSVs and organized in Databricks' three-namespace hierarchy: `<catalog>.<database>.<table>`, as seen in figure 3.14.

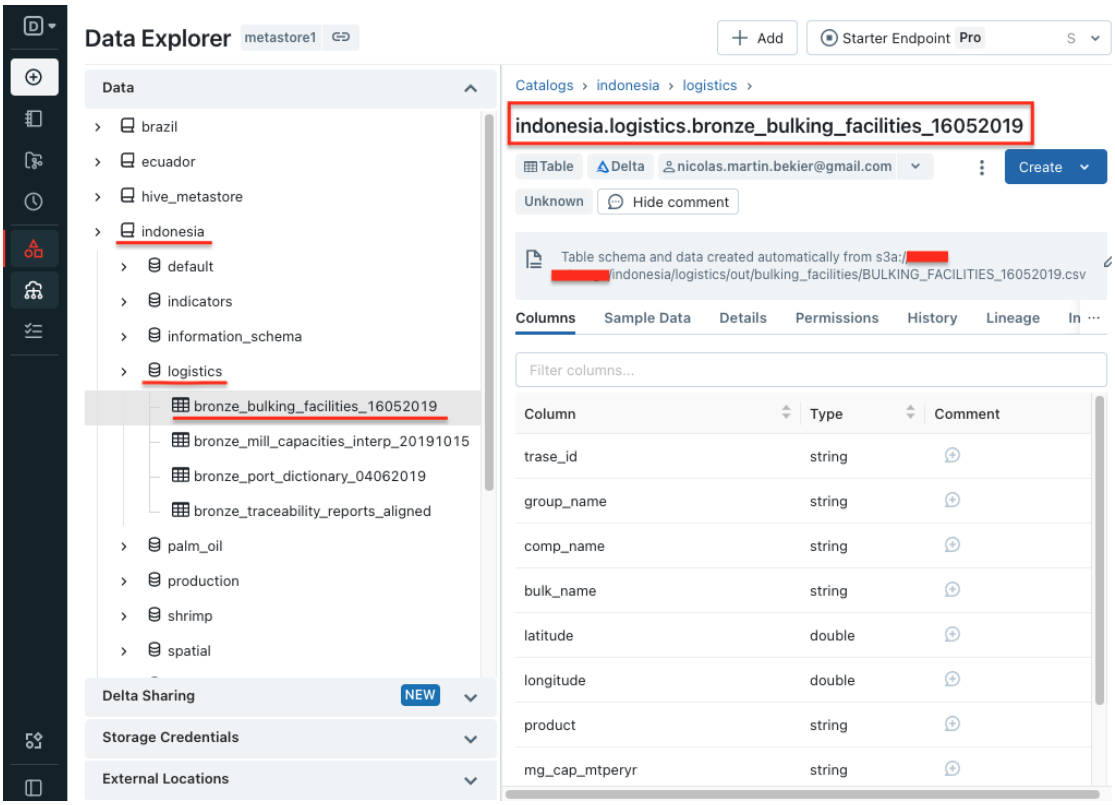

Figure 3.14: Databricks Data Explorer showing the table located at `indonesia.logistics.bronze_bulking_facilities_16052019` generated from a corresponding CSV

Some of the features of the working prototype include:

- Process recursively an S3 path for all their containing CSVs, and creating a delta table from it organized within the three level namespace `<catalog>.<database>.<table>` as seen highlighted in figure 3.14
- Alternatively, process a Trase yml catalog file called `requirements.yml` (see figure 3.15) which includes metadata of some of the data sources (description, labels, owner, source references, as well as to which country, commodity, and related topic it belongs to). When processing with this alternative, it allows to process only the countries or commodities specified in `requirements.yml`.
- Potentially, include some of the metadata specified in the YML file within the table properties of the created delta tables. However, this was not further developed as the `requirements.yml` will potentially become obsolete once the Metadata starts to be managed mainly with the data catalog.
- Pre-processing the CSVs with spark before loading them, including the ability to:
  - Identify the record delimiter.
  - Identify if the header has non-compliant ANSI-SQL column names and if it does, rename them so as to allow processing the file.
  - Include several of the pre-processing functions of spark, such as inferring schema, ignoring leading and trailing white spaces, allowing for multi-line records, among others.
- Save a log including information on each malformed record in a specified badRecords path, which includes its original data and the reason it got dropped.

For pre-processing the CSV, the first bytes of the CSV are loaded using a truncate read, which will be enough to identify the delimiter and if the header is well formed. Depending on this review, one of the following happens:

- If the header is well formed, the CSV is read and loaded normally to a corresponding spark dataframe and saved as a Delta table.
- If the header is badly formed, but the dataset is small (this threshold has been set to 10 megabytes), the CSV will be loaded as a text file in memory, the header replaced within it with a valid format, and then it will be loaded as a spark dataframe and saved as a Delta table.
- If the dataset is bigger than 10 megabytes and the header is badly formed, the first part of the file (65 kilobytes) will be loaded into memory, the

header replaced within it with a valid format, and the schema inferred. An empty table based on this schema will be created, which will later be populated with the whole CSV file, excluding its header.

Beyond the work done in the current thesis, there are several relevant improvements that can be made in the future. Among them:

- Create a table registering the details of each CSV, including:
  - A timestamp with the start of the process.
  - The filepath of the CSV source.
  - The filename of the CSV source.
  - The last modified time of the CSV source.
  - The file size of the CSV source.
  - The name of the table, database and catalog (in different fields).
  - If the transaction has been successful (bool).
  - The number of records written.
  - The number of bad records.
  - The `badRecordPath`.
- If a CSV has already been processed, only process it again if it is a different version from the previous one. Also, account for schema changes with schema evolution, or if it needs a complete overwrite in case it does not have a compatible schema.
- Define the mechanism to identify CSVs to process and CSV changes. Some options for doing this include:
  - Traversing the whole S3 bucket with the CSVs in scheduled intervals.
  - Using an AWS generated file catalog, which includes file paths, names, size, modified times, and checksum, to identify which files must be processed.
  - Using AWS CloudWatch, so that new or modified CSV files get immediately created as Delta Tables, or within some defined time interval (e.g. hourly, daily).
- Account for deleted files, by marking the corresponding Delta Table as deprecated.
- Account for moved files, and if it needs to be moved within the Unity Catalog three namespace hierarchy. If it needs to be moved, report this within a table property.

As a side note, although Airbyte has also been tested for ingesting CSV files, it did not have several of the features required for the working prototype.

```
- namespace: brazil
  repository: cocoa-agricultural-agricultural-land-used-for-cocoa
  metadata:
    readme:
      file: brazil-cocoa-land-used-for-cocoa.998d.md
    description: Brazil Cocoa Agricultural land used for cocoa
    topics:
    - agricultural-land-used-for-cocoa
    - brazil
    - cocoa
    - indicators
    sources:
    - anchor: Brazilian Institute of Geography and Statistics (IBGE)
      href: https://sidra.ibge.gov.br/pesquisa/lspa/tabelas
      isCreator: null
      isSameAs: null
    license: Do not share externally
    extra_metadata:
      trase:
        Back end name: COCOA_AREA_PERC
        GEE script url: ''
        Method: Check the data source for more details on the method.
        Quarterly release: Q4-2019
        Tooltip: Percentage of agricultural land that is cocoa.
        Trase earth link: 'Trase.earth: ********'
        Type: ind
        auto_generated_indicator: yup
        years: 2015-2017
```

Figure 3.15: requirements.yml file compiling current metadata, including its description, labels, owner, source references, as well as to which country, commodity, and related topic it belongs to

Working prototype: ingestion of heterogeneous CSV sources into delta tables in Unity Catalog

The following table lists and links to different python notebooks containing the code of how the CSV loading is done.

Notebook	Description
load_csvs_in_unity_catalog.ipynb	Create Delta Tables for all CSVs within a S3 file path, recursively exploring directories
load_repositories_yml_in_table.ipynb	Load the metadata within 'repositories.yml' in a Delta Table, flattening some of the nested fields in the destination table
create_tables_from_repositories_catalog.ipynb	Create Delta Tables based on specified country (and optionally a commodity), using as reference the 'repositories.yml' file

Replication and potential replacement of a PostgreSQL database

As proposed in figure 3.11 when discussing a potential transition architecture into a lakehouse, one of its components is keeping the PostgreSQL database synchronized to corresponding Delta Tables. This dual-setup ensures continuity; users can keep working with the PostgreSQL database during the transition or they can opt to use the Delta Tables, confident that both sources hold identical data. This is advantageous for processes like quality assurance, which may be complex to replicate on new systems. Meanwhile, activities like analytical queries or data sharing can be shifted to the more efficient Delta Tables.

The PostgreSQL database currently plays a role akin to a data warehouse, acting as the single source of truth for diverse data types such as: trader hierarchies, geographic information, commodity equivalence factors, details of logistics hubs, the output from our subnational supply chain models, and spatial metrics. It also serves as the backbone for quality assurance, analytical queries, and preparing data for public distribution via a relational database and a web application.

Yet, PostgreSQL lacks some intrinsic data warehouse features that would be beneficial for Trase. For instance, the database is not inherently non-volatile or time-variant. Although regular backups are made, retrieving a specific data version or historical point is cumbersome. Schema or data changes in the database could disrupt queries or processes that rely on certain configurations or values. In contrast, a lakehouse model, such as that based on Databricks, allows querying specific versions easily and offers the possibility to create logical table copies. These copies can undergo changes without impacting the source or duplicating data, with the option to merge alterations back into the original table if necessary.

Moreover, data warehouses are optimized for Online Analytical Processing (OLAP), which is advantageous for handling aggregations and analytical queries over the Online Transaction Processing (OLTP) systems that relational databases like PostgreSQL are typically geared towards. Although PostgreSQL can be configured for columnar storage, it is principally designed for high-concurrency transactions rather than analytical workloads.

Quality assurance in Trase's current setup relies heavily on PostgreSQL's constraints, but a more adaptable approach might involve additional tools or services. Tools like DBT tests [66] are already in use for setting data expectations, though to a lesser extent than PostgreSQL's native constraints. Other tools like Deequ [67] or Great Expectations [68] could offer further flexibility, allowing for computed metrics storage, data profiling, and anomaly detection based on these metrics.
Lastly, considering performance, resource utilization, and cost, the columnar storage of data warehouses significantly enhances the efficiency of organizing, compressing, and aggregating data.
Two main mechanisms have been explored to achieve the replication of current PostgreSQL tables in a way that allows for a transition and eventual replacement:

- An incremental replication using spark, which compares table data to identify and reflect only the changes required between versions.
- Usage of change data capture (CDC) in the source tables and replication based on it through Airbyte.

As will be discussed shortly, the spark replication is preferable in terms of simplicity, performance, and adjustment possibilities. The source currently needs several adjustments to be able to use CDC, and Airbyte current synchronization options with Databricks require considerable additional work, with lower performance and flexibility.

Working prototype: incremental replication of PostgreSQL tables using spark

To illustrate the advantages of using Databricks Delta Tables for enhancing Trase's PostgreSQL data with non-volatility and time-variant capabilities, the corresponding prototype embodies these concepts. This prototype, upon receiving a set of PostgreSQL tables designated for replication, performs several functions to maintain and transform the data within the Databricks Unity Catalog.
The process begins with the replication the specified tables into the Unity Catalog. Should a table not be present at the target location, the system is designed to create it. The prototype also recognizes any alterations that have occurred in the source table since the last sync. It pinpoints these modifications, which could be new records (requiring INSERTS) or removed ones (requiring DELETES), and updates the destination table accordingly. This method is particularly efficient for large tables as it prevents unnecessary growth of the destination table by avoiding complete overwrites during each synchronization.
Furthermore, any changes to the source table's schema are detected and similarly updated in the destination table, ensuring structural consistency. The system is also designed to handle refresh timing efficiently. If a refresh time is specified for a source table, it will use this to determine the availability of a new version. In the case of a required update, the prototype will completely load the source tables, compute a unique

hash value for each record based on the concatenated column values, and then execute the comparison against the records in the destination table.
At the destination, an extra 'dbr_hash' field is added to facilitate the identification and management of record changes. This field helps in determining which records are to be inserted or deleted based on the source table's updates. Meanwhile, the 'last_refresh_time' field from the source table is not directly replicated; instead, it's stored as a table property named 'source_timestamp'. This strategy is chosen to prevent unnecessary storage consumption with each version update, which is especially beneficial for large tables.
For processing large tables (> 100MB) in parallel instead of a single thread, a column partition can be specified as long as it has an integer value. Ideally, it is preferable to choose a column with high cardinality and evenly distributed. Based on this field, the script identifies the lower and upper bounds required by spark's PostgreSQL driver to read in parallel partitions. In the tests conducted using partitions, a 5 to 10GB PostgreSQL table is loaded and processed in 2 minutes using a single spark node with 8 cores; using two nodes (16 cores) it processes in 1 minute. When not using partitions, the same table takes 18 minutes to process. As mentioned previously, processing includes loading the table, adding a hash value to each record based on all the fields, comparing the values with the destination table, and based on this, running the corresponding `INSERT` and/or `DELETE` statements. If there is a change in the source schema or if the destination table doesn't exist or is empty, all the source table is loaded in the destination Delta table.
Beyond the work done in the current thesis, there are several relevant improvements that can be made in the future. Among them:

- Include a primary key in the source tables, making it possible to use change data capture (CDC) using a write ahead log (WAL).
- Make the changes on the source tables incremental, as this is required for CDC to work properly.
- Include a cursor field in source tables, so it's only needed to load the records above the cursor field value (usually a timestamp).

In implementing the working prototype some of the most used tables and views are selected (9 tables and 5 views). These tables range from a few megabytes up to several gigabytes, don't have primary keys, and are constantly being re-created from other source tables. Each time they are re-created, a `'last_refresh_time'` column within each table is set based on the creation time, though might not include different data as from the last refresh.
The following script link includes the main code related to the PostgreSQL replication using spark.

Notebook	**Description**

`postgres_to_databricks_replication.ipynb`	Implementation of the replication as described

Usage of change data capture (CDC) using write ahead log (WAL)

Change data capture (CDC) allows capturing changes made to a table in real-time, by identifying and extracting only the modified data from it and propagating those changes to other systems or data repositories. Using CDC makes it easier to keep track of a source system (including for replicating it, which would be the interest in our case), and avoid unnecessarily sending duplicated data. Also, it lowers the burden on the source system, as it does not have to find and report changes based on queries conditioned to a datetime field. A special kind of CDC called 'Logical CDC' and which is tested in the current thesis, keeps track and allows propagating the specific INSERT/DELETE/UPDATE statements, instead of the data itself. In order for CDC to work, it relies on a write ahead log (WAL).
The write ahead log is a mechanism used to ensure data durability and recoverability. It is a transaction log that records changes made to the database before they are written to disk. This provides a reliable way to recover the database in case of system failures or crashes. change data capture (CDC), uses the information stored in the WAL to capture and propagate database changes in real-time or near-real-time. CDC reads the changes recorded in the WAL and transforms them into a consumable format that can be processed by other systems or applications.
By capturing changes at the row level, CDC enables extracting only the modified data, reducing the amount of data transferred and improving efficiency. As it does not require querying the table for retrieving data, but only giving access to the WAL, the replication process does not drain the database of its resources while running.
CDC is tested with a couple of Trase's PostgreSQL tables, using Airbyte for ingestion, after creating a primary key (PK) for those tables (as required for CDC to work properly). The advantages and drawbacks of the different options are reviewed after showcasing the working prototypes.

Using Airbyte for ingestion

Airbyte [69] is an open-source platform designed for data integration and ingestion. It simplifies the process of collecting and moving data from various sources to destinations including data warehouses and lakes. For this, it offers pre-built connectors that allow reading from hundreds of popular data sources such as databases, APIs, or cloud services and connecting them to dozens of popular data destinations. If a connector for a specific source or destination does not exist, it provides tools to develop it. Also, it supports both incremental and full data replication. Some of the most popular connectors include:

- Databases: PostgreSQL, BigQuery, MySQL.

- Marketing: Google Ads, Facebook Marketing.
- Analytics: Google Analytics, Amplitude.
- Project Management: Asana, Jira.
- Customer Relations: HubSpot, Intercom.
- Version Control: GitHub, GitLab.
- Cloud Storage: AWS S3, Google Drive.
- Communication: Slack, Zoom.

Given the current adoption of Airbyte open source core (currently more than 10k github stars), the number of connectors and the options for developing new ones, it can simplify the process of ingesting from various sources. In this thesis, Airbyte is tested mainly as an alternative for ingesting the content of PostgreSQL tables into Databricks Delta Tables in Unity Catalog. As mentioned in a previous section, although Airbyte is used to test ingestion from CSVs, it has several drawbacks that make it cumbersome to use (a connection configuration is required to be set up per each CSV to be processed), as well as requiring changes in its connector source-code to allow for different delimiters, as well as for fixing non-standard headers.
For ingesting from PostgreSQL, Airbyte currently allows the following source-destination options with Databricks:

- Full Refresh - Overwrite
- Full Refresh - Append
- Incremental Sync - Append (requires a cursor, or if using CDC, a PK)
- Incremental Sync - Deduped History (not currently available with Databricks as a destination)

For doing incremental syncs using CDC, which is the test case made for Trase, for the moment it can only append records to Airbyte generated tables, which includes the data of each source record within a JSON blob made of '`field_name`': '`value`' pairs. Also, it does not currently support Deduped History incremental sync, nor does it create the actual replica of the source database (though provides information that can be used to achieve this).
While this procedure is tested using two of Trase's PostgreSQL tables, it requires the creation of the actual destination tables based on Airbyte's intermediary append-only tables. Additionally, the source tables to replicate do not have primary keys or a cursor and are fully refreshed instead of being incrementally created, and CDC does not work on those conditions.

For these reasons, the spark-based PostgreSQL replication from the previous working prototype is deemed preferable to use, even if it requires computing the difference of the whole datasets each time it looks for updates. It is simpler, more versatile, and takes considerable less time and computing resources to run.

3.3.3 Pipeline management in Trase

In Trase's data ecosystem, some of the main challenges of managing pipelines stems from the diverse and descentralized development of models for each commodity and country. Despite the documentation and public sharing of these models, standardization of code and data practices has only recently been emphasized through:

- Development of a unified codebase on GitHub.
- Introduction of helper functions to promote modularity.

Trase standard operations in pipeline management

Core operations integral to Trase's pipeline management include:

- Defining and overseeing the supply chain models within Trase.
- Simplifying pre-processing tasks through advanced wrapper functions.
- Efficient data loading, predominantly from structured CSV files in S3 storage.
- Implementing rigorous quality assurance protocols for input data.
- Processing and uploading datasets to S3, formatted for compatibility with Trase models.
- Executing models and disseminating the resultant data.

These practices help improving access and coordination across the data and codebase, while being flexible enough to be used by researchers with different language expertise and preference, especially in R and python.

Challenges in heterogeneous development environments

Despite these standardizations, challenges related the heterogeneity in coding practices among researchers persist. This diversity, while useful for accommodating individual expertise, complicates the management and synchronization of the pipeline across commodities and countries. Specific challenges include:

- Local data processing on researchers' personal computers, which risks data version inconsistencies with the centralized S3 storage.

- Variability in CSV formatting, including separators, null value representations, and header conventions, which increases the potential for data inconsistencies.
- Navigating version histories in S3 is non-trivial, exacerbated by limited client application support, making it difficult to differentiate between stable and development data versions.
- The absence of a standardized file naming and path convention in S3 complicates data retrieval and management.
- The necessity for researchers to configure development environments and access to AWS and database resources individually.

Orchestration and dependency challenges

The loose coupling of pipeline components poses significant challenges for executing end-to-end processes. These include:

- Difficulty in tracing dependencies between datasets and processing scripts.
- Challenges in reviewing processed data metrics at each pipeline stage.
- Complexities in defining and modifying data processing expectations, such as value ranges and aggregated statistics.
- The need for a shared environment that facilitates pipeline component execution without individual environment configuration.
- Keeping track of pipeline runs, such as: who ran it, when, which source dataset and script versions are used, did the whole process was completed, failed, or issued warnings?

To address these challenges within the prototypes developed in the thesis, the Databricks platform is used in a way that allows for unified management of the pipelines, including version control, access permissions, infrastructure deployment, and task orchestration. The two explored orchestration mechanisms are Databricks Jobs, and Delta Live Tables.

Working prototype: pipeline orchestration with Databricks Jobs

Databricks workflows orchestration services allow managing multi-task workflows with complex dependencies [70], including cluster management, monitoring, and error reporting. Databricks jobs workflows can be configured through its User Interface (UI), Command Line Interface (CLI), or API, or Apache Airflow [71]. Databricks workflows allows configuring one or several dependencies of tasks specifying, per task :

- Type of the Task: Databricks notebook, python script, SQL, Delta Live Tables, DBT, Jar, Spark Submit.

- Source: Databricks workspace or git source.
- Trigger: manual, scheduled, on file arrival detected (including external S3 location), continuous.
- Cluster: computing infrastructure selection, including auto-scaling capabilities.
- Library dependencies, in case they are not available through the Databricks runtime used to execute the task, or configured to be installed within the notebook definition.
- Other: notification configurations (email, slack, webhook), retries, timeouts.

During this thesis, simple tasks such as the execution of the notebooks replicating some of Trase's PostgreSQL tables have been tested, as can be seen in figure 3.16.

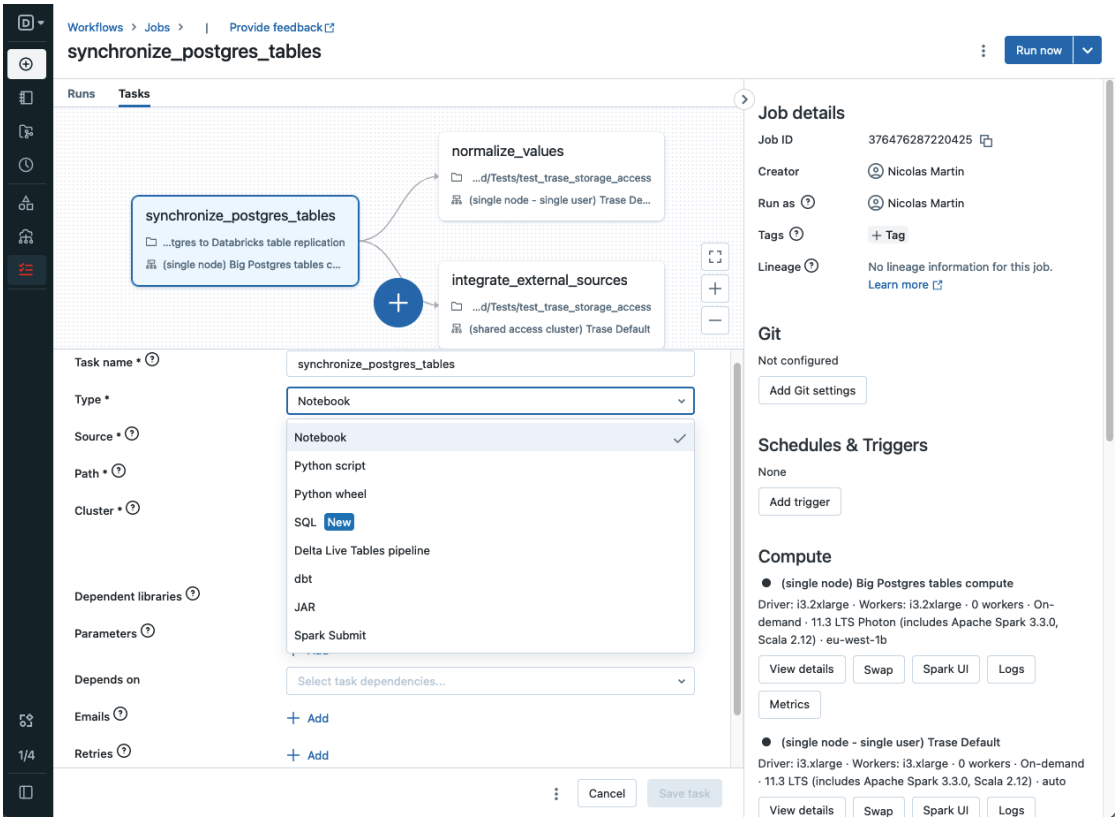

Figure 3.16: Databricks jobs for synchronizing tables from a PostgreSQL database

In the previous example, after running the job of synchronizing the PostgreSQL tables in Delta Tables, two additional test jobs are executed: one to normalize some of the values, and another one to further integrate external resources. Each job can have its own configuration.

Working prototype: pipeline orchestration with Delta Live Tables (DLT)

Delta Live Tables (DLT) enhance the management capabilities of Databricks Jobs by allowing for the declarative definition of data processing pipelines. These pipelines can also integrate the definition of data quality expectations and remedial actions for records that fail to meet these standards [72]. DLT pipelines, which can be defined using either python or SQL, support a range of operations including data ingestion, transformation, and the storage of the resulting datasets. They are also compatible with streaming tables, views, and materialized views. Streaming tables are particularly beneficial for incremental data processing, enabling pipelines to handle data continuously. This means they can apply transformations and compute results as new data flows in, without the need to reprocess the entire dataset from scratch.
DLT provides real-time monitoring and management of pipeline executions. This includes insights into the records processed and any that have been discarded due to predefined expectations. These logs offer additional information about each pipeline stage (as illustrated in figure 3.17). In instances where a pipeline's execution fails or deviates from expected data quality metrics, the UI makes it possible to pinpoint the problematic dataset and the associated source Notebooks within the pipeline.
Pipelines can operate in either 'Development' or 'Production' mode. Each mode permits different settings and allocates varying levels of computing resources to suit the needs of the task at hand.
The following notebook links shows working prototype links of using DLT for Trase.

Notebook	Description
`ec_shrimp_area_production_per_parish_dlt_version.ipynb`	DLT rewrite of Trase script pre-processing Ecuador custom declarations and shrimp ponds on a local level (parish)
`shrimp_delta_poc_dlt_version.ipynb`	DLT rewrite of Trase script compiling different sources containing shrimp processing facilities

These notebooks were written mainly by Oskar Åsbrink in 1DL507 Project in Data Science course (1DL507) [12], with the collaboration of Nicolás Martín.
They are based on a Trase pre-processing script for Ecuador shrimp, that takes customs declarations from 2013 to 2019, and information about shrimp ponds. Based on them, it generates production information per parish -Ecuador's third-level administrative unit- as well as the existing shrimp area in hectares.
The notebooks take the original Trase script and wraps the creation of CSVs in it with the creation of Delta Live Tables in a way that allows for running and visualizing the process as seen in figure 3.17. Additionally, for creating the Delta Live Tables it uses spark dataframes as this is a requirement of DLT. Otherwise, it uses pandas for general logic and transformations.

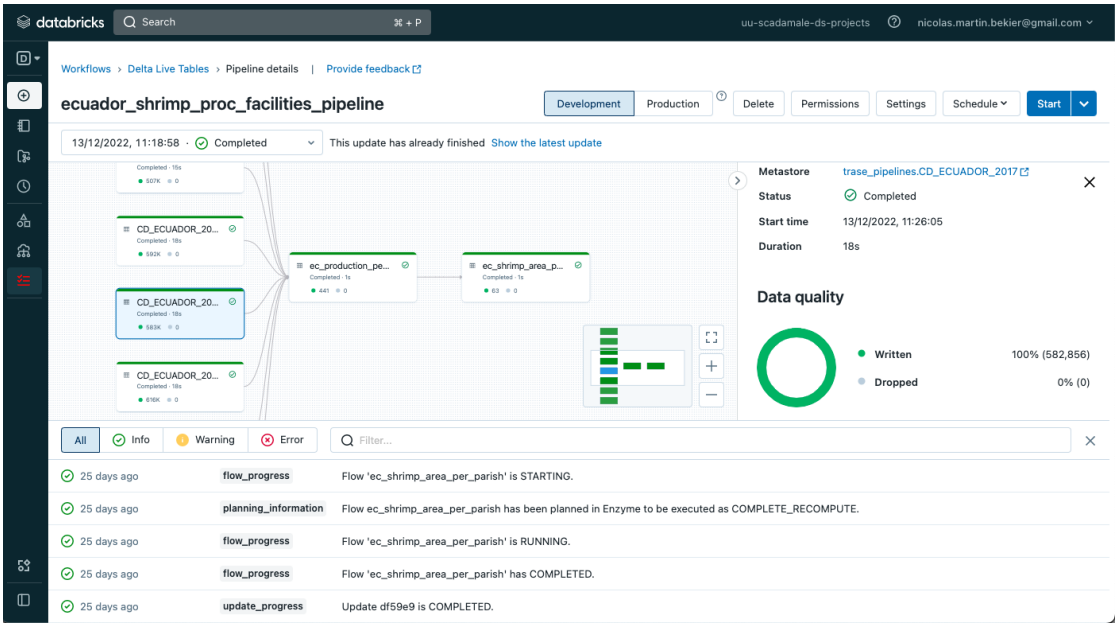

Figure 3.17: Databricks pipeline with Delta Live Tables

In the '`shrimp_delta_poc_dlt_version.ipynb`' notebook, the inclusion of Trase's codebase is tested being included as a pip installation of a python wheel containing it. Though an easier and recommended approach is to define and run the pipeline as a script (or scripts) within a git repo and access the related codebase of it when needed, this feature was not enabled when the working prototype was developed. Note that the code and additional imported libraries should be compatible with the packages available in the Databricks runtime (DBR) version used to run the pipeline [73].

Future improvements to the working prototype

Beyond the work done in the current thesis, several further adjustments can build upon and improve the work done, such as the following:

- Transfer part of the new logic of the DLT into the existing helper functions libraries. Specifically, on helper functions that load a CSV from S3 into a dataframe, or that write a CSV in S3 based on a dataframe. Additionally of doing this, the helper functions could also create the corresponding Delta Live Tables.
- Load Delta Tables instead of CSVs, once there is a replication process of them. In this way, when a helper function to read or write from/to a CSV is called, it loads the corresponding Delta Table. This would further solve the problems related to CSV formats mentioned earlier.
- Eventually, the reliance on CSVs as the main storage format could be deprecated.

Also, when the code is being updated -which is usually done when there is a model or data update-, additional improvements could be integrated, such as:

- Transforming the logic to use mainly SQL instead of python and R. While python and R can still be used to allow for flexibility on managing multiple datasets and complex transformations, most of the transformations could be expressed in an easier and more legible way using SQL, including the use of DLT. This could also work as a transition to use other transformation frameworks such as DBT, that allow for managing more modular and versioned data transformations.
- Integrate yearly datasets into compiled tables which include information on all available years. This can make it easier to load and manage multi-year datasets, including:
  - Not requiring to have one year per dataset in case the source data comes multi-year.
  - Making exploratory analysis of this multi-year information, including filtering the dates of interest instead of adding tables together.
  - Use Auto Loader or an automated script that integrates all available source datasets within a folder when the script is run or detects an event (for example including a new file in a source folder).
  - Use change data capture (CDC) to update resulting tables based on changes in source data. DLT allows for using CDC retaining a history of records if needed.

Management of pipeline data quality

Within Trase's codebase there are high-level functions that seek to standardize and simplify the management of datasets related to a supply chain. Also, there are helper functions to verify that the data complies with predefined standards as well as log the corresponding warnings or errors. As mentioned before, scripts also tend to have specific logic for cleaning and formatting the source datasets per-source. The way the codebase is used can vary depending on the researcher processing the data and writing the transformation logic, which allows for flexibility, though can also lead to duplication of code logic, as well as fuzzy separation between data cleaning pre-processing and domain-specific transformations and aggregations, even though there is easily accessible documentation and tutorials to help following standards.
Additional to these data quality efforts for pre-processing and initial transformations, the main centralized place used for the quality assurance between datasets is in the PostgreSQL database.
By using a lakehouse architecture and integrating a medallion (bronze, silver, gold) data pattern, this process can be simplified:

- The general data flow would be: Ingestion (ELT based) -> Delta tables (within a three level namespace and a medallion pattern) -> serving for analytics / BI / potentially ML.
- Processing scripts would not need to interact directly with CSVs -which are error-prone, lack a schema and a consistent format standard-, or PostgreSQL.

This simplification allows for easier governance, deployment of infrastructure, and version control, making it easier to define and review data quality. The main relevant considerations for this lakehouse architecture using a Databricks implementation are:

- In Databricks Delta Tables, similar to other warehouse platforms (e.g. RedShift, Snowflake), primary and foreign keys are not natively supported, so referential integrity requires CHECK constraints or other methods [74].
- To allow for more complex constraints, such as defining them based on conditions calculated using several tables, DLT expectations can be used. DLT expectations are boolean statements that for each passing record, can generate a warning, drop record, or fail action. Fail assertions can further quarantine the corresponding records as well as send them through a different downstream path.

While Delta Live Tables includes several features to facilitate pipeline management using declarative statements in python or SQL, a downside is that it is not currently an open source standard. While it is planned to be open-sourced in the future, similar to other Databricks components related to Delta Tables [75], there is no current timeline for this. However, as DLT components are defined declaratively using python or SQL, and based on the spark and the Delta Tables API, it is possible to migrate the transformations and expectations logic of the DLT pipelines outside Databricks (however, this could require considerable effort, and would lose the integrated management functionalities).
Another solution for defining and managing data quality expectations is the Great Expectations [68] open source initiative, which allows for defining and automating pipeline tests and applying them to the data instead of the code. Great Expectations is currently compatible with more than 20 popular data platforms and tools, including Databricks, BigQuery, RedShift, Snowflake, PostgreSQL, Trino, Airflow, and spark, among others. Also, it includes integration with the DataHub data catalog working prototype showcased later on.

3.3.4 Data discovery, metadata management, and data sharing

Data discovery refer to the process of locating, identifying, and understanding the available data resources within an organization. It involves discovering the various

data sources, whether structured or unstructured, across different systems, databases, and applications. Data discovery allows for gaining visibility into the existing data landscape, enabling them to ascertain data quality, relevance, and potential usability for analytical or operational purposes.
Metadata management is the practice of capturing, organizing, and maintaining metadata, which encompasses information about the data itself. Metadata provides essential context and understanding of the data, including its origin, structure, format, relationships, and business meaning. Metadata management techniques allow for establishing data catalogs, and documenting the characteristics and attributes of datasets. By effectively managing metadata, organizations can ensure data accuracy, enhance data governance, and facilitate collaboration and sharing among internal and external data users, analysts, and other stakeholders.

Creation of Trase's data catalog

Currently, Trase has loosely-coupled data assets distributed among S3 buckets, the PostgreSQL database, and Google Earth Engine. Description of the datasets can be found in multiple locations:

- Within the scripts where the data is processed or generated.
- Within a consolidated YAML file which contains description, source information, and tags, among others.
- Within YAML files that show the lineage of some datasets. These files are kept alongside the S3 folder where the related datasets are.
- Within Readme.md files in related github folders.
- Within google docs that describe the methods used around a specific country commodity, including its key datasets.

Although the variety of options allows for flexibility, it also creates several challenges:

- The variety of storage locations makes it hard to have a unified view of all metadata, search it, browse it by categories (tags, type of dataset, domain-specific taxonomies, ownership) or view its lineage.
- It is hard to connect metadata with other systems, such as quality expectation tests, automated warnings, among others.

Acryl DataHub data catalog

After a review of open-source systems and components, a working prototype using Acryl DataHub [43] has been implemented. DataHub is a metadata platform open-sourced by LinkedIn in 2019, integrating loosely coupled components. Some of the most relevant components that it can offer to Trase are:

- Search and discovery: it can unify metadata from different data sources and trace lineage information. These functionalities can be accessed through a UI, a REST API, or custom connectors.
- Automated notifications and actions: notifications can be set based on custom events, such as when a tag is added, dataset metadata changes, or a dependency is possibly impacted. Notifications can be emailed or connected to other systems, such as Slack or Teams. Also, the events can trigger actions within DataHub or through connected systems.

Installation can be done through a Docker compose -though this is not recommended for production-, or preferably using Kubernetes through a provided Helm chart.
The philosophy of DataHub, as described by one of its creators [76], is characterized by a model-first approach and a loosely coupled architecture. At its heart lies a metadata model constructed around entities and their interconnections. Entities possess attributes or sets of attributes known as aspects.
To illustrate, consider the representation of the ownership of a specific dataset. For this, consider two entities: datasets and users, where the dataset is owned by a user. Each entity may be associated with further aspects; for example, a dataset might include a 'schema' aspect, and a user might have a 'profile' aspect. This graph-based model provides flexibility, expressive strength, and the facility to query or search through complex structures of entities and their relationships with ease.
In terms of architecture, metadata can come in through a stream or batch source, or directly through an API (for example REST), connecting to a metadata service that stores it in a database (MySQL or PostgreSQL), as well as sending it to a change log connected to a search service, a graph database, and potentially other services.
This architecture allows for flexible connection to other systems, be it for ingesting metadata based on a wide range of sources that can push it to DataHub, or have it pulled manually, scheduled, or based on a trigger. Once processed by the metadata service, the information in the data catalog can be viewed, queried, or modified through a frontend, by API integrations, or by stream integrations using Kafka.

Working prototype: DataHub implementation with Trase's data

For testing the integration of DataHub with Trase's data assets, a working prototype was developed including:

- Basic deployment in a 4 CPU, 16 GB compute node in OpenStack running Ubuntu 22.04.
- Ingestion of:
  - S3 files: CSV and JSON files have their schema (column and column type) inferred. Other files (e.g., XLS, DOC, PDF, etc) are cataloged though without schema.

- Tables and views in PostgreSQL.
- Metabase related data in PostgreSQL.
- Data assets within Databricks Unity Catalog. For the moment, this requires custom adjustments to the DataHub - Databricks plugin source code.

- Test of data profiling of CSVs, using pySpark, and the pyDeequ library.

As an example of DataHub's usage, a simple search with autocomplete can be seen in figure 3.18. Also, the description of a dataset can be seen in figure 3.19, and the visualization of a data lineage in figure 3.20.

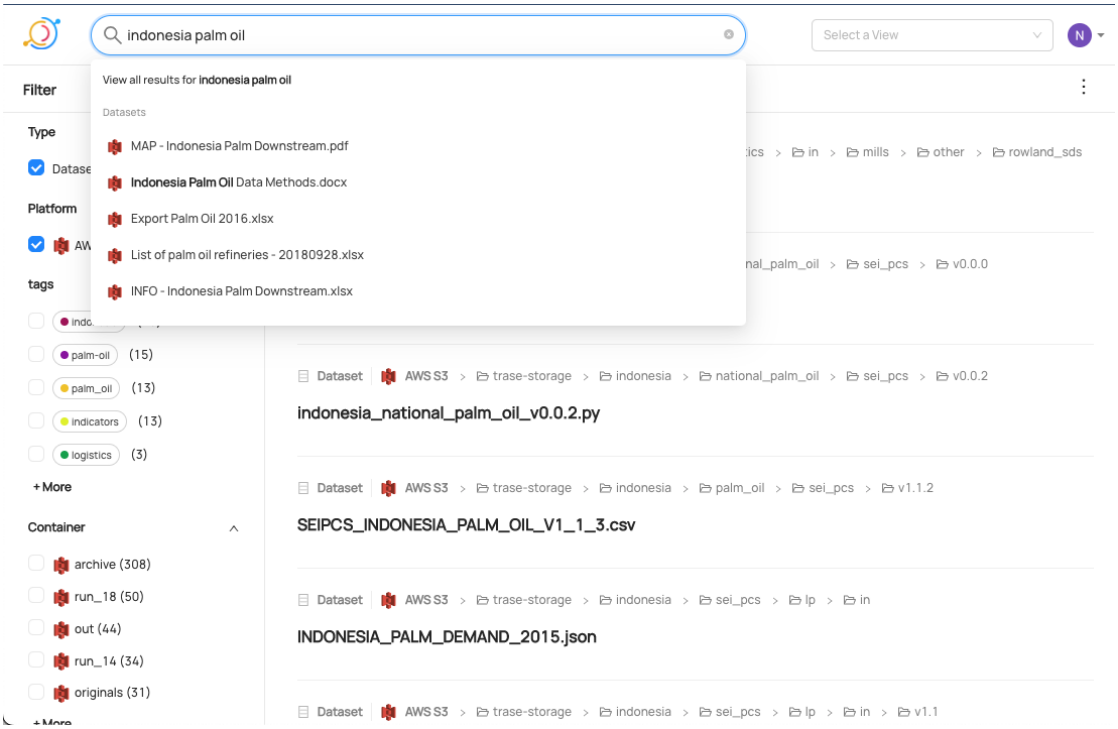

Figure 3.18: Showing assets related with 'Indonesian Palm Oil'

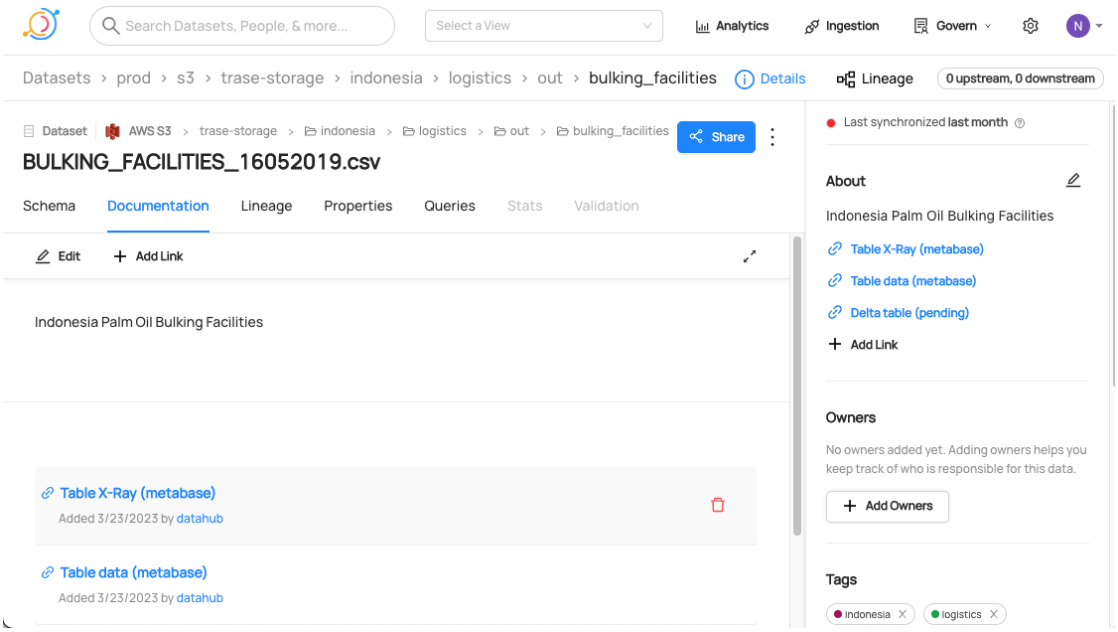

Figure 3.19: Dataset description with links and context

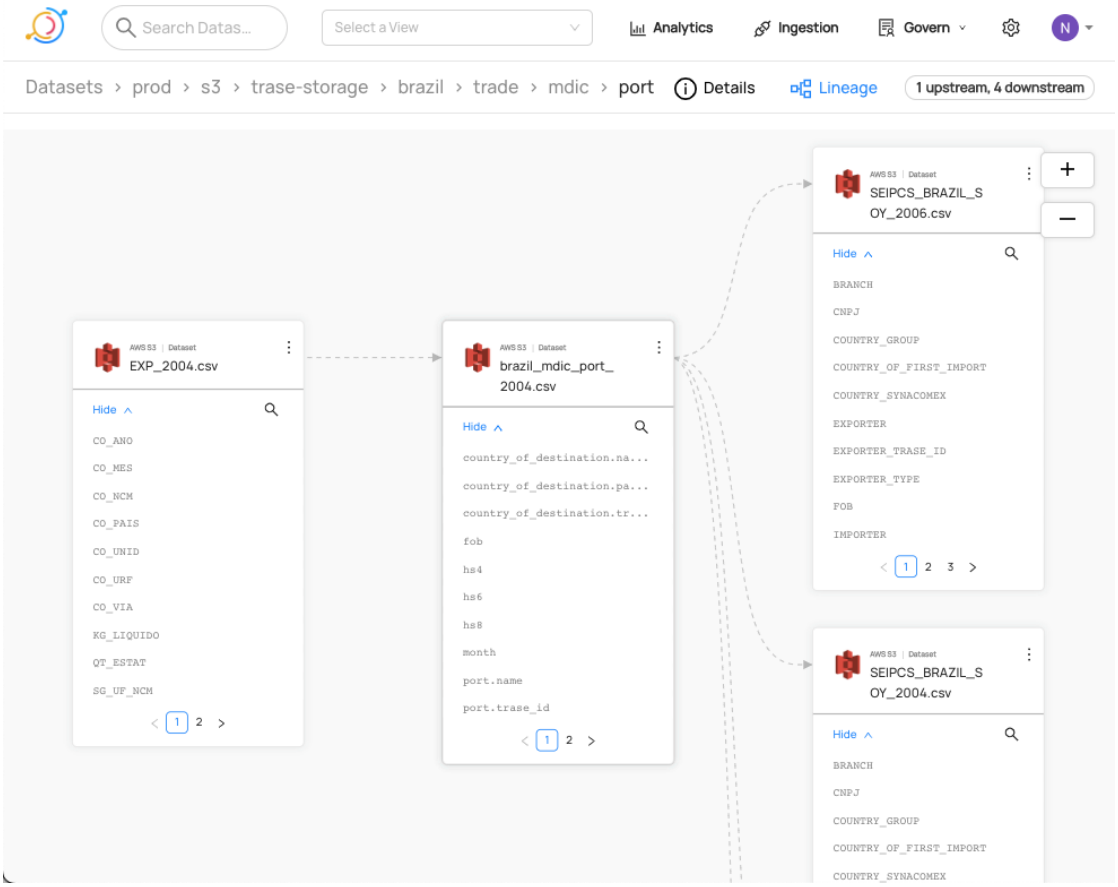

Figure 3.20: Dataset lineage showing one dataset upstream and four downstream

For the initial ingestion, YAML 'recipe' files are created for each source type. As the file and folder structure within Trase's S3 does not follow a stable structure, the path specs

for including source files are not defined through include/exclude general patterns mentioned in the S3 Data Lake source connector [77]. Instead, each file to be cataloged is included explicitly. For inferring the column types of the schemas of CSV and JSON files, DataHub's default configuration uses the first 100 records.
For ingesting data assets from Databricks Unity Catalog, the source code of the plugin has been modified. In particular, the requirement from the `UnityCatalogApiProxy` [78] for listing Unity Catalog metastores has been disabled, as it requires a Databricks token from the Databricks account owner, including its administrative rights. This change was made for lack of this kind of access, as well as for security reasons. The adjustments are made based on Datahub guidelines for 'Developing on Metadata Ingestion' [79].
Based on the development of the working prototype, Trase's data team has extended upon it, including:

- Including all available files in S3, by developing scripts to create recipes that included all files and file types within S3. In contrast, the working prototype only included a sample of CSV and JSON files.
- Integrating Trase's metadata compilation on '`repositories.yml`' file to corresponding locations within the data catalog.
- Integrating lineage information available in the corresponding YAML files.
- Connecting the system with Trase authentication service for their user's access.

For some of these integrations, they have used DataHub's GraphQL API [80], as well as relied on DataHub's included browser-based GraphQL Explorer Tool (GraphiQL).
Currently, more than 50,000 datasets have been included in the DataHub catalog. Also, wider user tests to define the usage, priorities, additional connections, and adjustments have been developed by Trase's data team.

Delta Sharing and connection to other data sources and services

Accessing data stored in the lakehouse through a simple read-only protocol can be useful for various reasons:

- It allows for reading data through a native connector (e.g. python, R), without requiring a particular engine (e.g. spark or Databricks runtime environment).
- Some external applications and services can provide native support for this (e.g. PowerBI, Tableu, Trino).
- It can be used to share live data to external collaborators, while restricting some fields, records, and auditing access.

Delta Sharing is an open standard for securely sharing read-only data access. As Delta Tables protocol specification [81] is based on Parquet files (with several additions) and a transaction log, it is possible to manage access to the underlying data in a fast, scalable, and secure way. If using Databricks, Unity Catalog provides the information to map the data to its underlying information, as well as the capability to manage, govern, audit, and track usage of the shared data. If not using Databricks, a Delta Sharing Reference Server has to be deployed, which can be set up manually, or through additional vendors who offer it as a managed service.
This setting allows for accessing the Delta tables from various clients and platforms as seen in figure 3.21, which can read current data. This access does not require managing copies -and its associated staleness-, and can have custom permissions set for the recipient user. For each table, access can be restricted to specific columns or rows conditions through SQL WHERE statements, in the same way as when using Unity Catalog. Other native Delta Tables capabilities are available, such as accessing the history of a dataset, and access to the change data feed (CDF) of the table in case it has it.

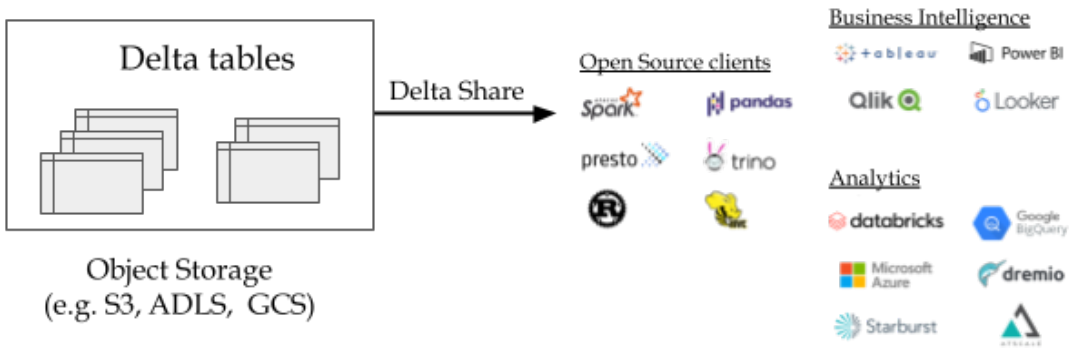

Figure 3.21: Delta share and some of the recipients it can connect to

Working prototype: Delta Sharing of Trase tables

For the working prototype the sharing of different tables is tested. These tables are then read from different external clients. When sharing a table to a non-databricks account, an activation URL is sent to the recipient, which allows a one-time download of a credential file. As shown in figure 3.21, there are multiple clients that can read the delta share. In figure 3.22 a simple example of a python connector reading a single table and loading it in a python pandas dataframe is shown.
This setting allows also for accessing Delta Tables without requiring to start a Databricks compute cluster running Databricks runtime (DBR) or an underlying spark engine.

```
[1]: import delta_sharing

     profile_path = f"/home/ubuntu/trase_uu/share/config.share"
     share_name = f"candyland"
     schema_name = f"candyland"
     table_name = f"cd_2017"

     client = delta_sharing.SharingClient(profile_path)

     client.list_all_tables()

[1]: [Table(name='cd_2017', share='candyland', schema='candyland')]

[2]: candy_df = delta_sharing.load_as_pandas(f"{profile_path}#{share_name}.{schema_name}.{table_name}")

[3]: candy_df
```

[3]:

	ncm	exporter	municipality	volume	fob	country	port	importer
0	0	123001	101	35	175	GERMANY	UNKNOWN PORT INDONESIA	UNKNOWN CUSTOMER
1	0	125001	101	20	100	GERMANY	UNKNOWN PORT INDONESIA	UNKNOWN CUSTOMER
2	0	125001	101	20	100	POLAND	UNKNOWN PORT INDONESIA	UNKNOWN CUSTOMER
3	0	166001	102	100	500	FRANCE	UNKNOWN PORT INDONESIA	UNKNOWN CUSTOMER

Figure 3.22: Example reading a delta table from python pandas through Delta Sharing

4 Graph data mining for the traceability of the beef supply chain in Brazil

This chapter addresses a parallel exploration to that of the lakehouse architecture, delving into the potential of graph and network analysis to derive insights from Brazil's beef supply chain. Graph-based data can make it easier to gain insights into not only the attributes of individual entities, but also the networks of relationships that interlink them. This form of data representation is inherently intuitive and offers a versatile framework for modeling complex systems. In the context of supply chain management, graph modeling and graph databases have become influential in informing decision-making and operations [82].
The explorations conducted in this chapter are intended to provide contributions that future research can build upon. By capturing the network of cattle movements between farms and slaughterhouses within a graph database, we facilitate the construction of detailed representations of each slaughterhouse's potential indirect suppliers. Among others, this enables us to identify actors that may be at risk of indirectly sourcing from regions or entities associated with deforestation or human rights abuses.

4.1 Graphs, graph modeling, and graph data mining

A graph comprises nodes which embody entities or objects, and edges (also known as links or connections) that represent the relationships between the nodes. Graph modeling is the act of mapping a dataset into a graph format, particularly for utilization within a graph database. Graph data mining, viewed through the lens of data mining and knowledge discovery, focuses on analyzing graphs by extracting implicit, potentially valuable information from their data [83].
Graph modeling and mining allow capturing complex relationships and dependencies between entities more intuitively and flexibly than traditional data structures. It enables the analysis of network properties, patterns, and structures, as well as the exploration of various algorithms and techniques specific to graphs. By modeling a system or dataset as a graph, we can leverage graph algorithms and techniques to gain insights, detect patterns, predict behavior, and make informed decisions based on the relationships and connections within the graph.

4.2 Graph databases and Neo4j

Native graph databases are specifically designed to store and query graph data, allowing efficient storage and retrieval of complex graph structures [84]. They usually store pointers to relationships alongside the entities' information. This makes it both easy to write relational information, as well as retrieving it, as it only needs to chase pointers to find associated relationships and entities. Native graph databases also use specialized indexing and traversal algorithms optimized for graph operations. By contrast, representing and exploring relationships in a traditional relational database management system (RDMS), requires managing explicit keys, foreign keys, and based on this joins and self-joins in a way that is harder to represent, and that is computationally more expensive. Additionally, this does not allow for easily capturing the structure of a graph and consequently running queries that explicitly refer to these structures. According to db-engine.com, graph databases have been the fastest-growing database category in the last 10 years [85].
Graph databases also provide a query syntax that allows the expression of complex graph patterns and relationships concisely and intuitively. This makes it easier to explore and query the graph, navigate through nodes and relationships, and perform advanced graph analytics.
Currently, Neo4j is one of the most popular native graph databases, claiming to be the biggest in terms of community, deployments, and the largest catalog of graph algorithms, among others [86]. In particular, Neo4j is a 'property graph', where both entities and relationships can have one or more labels attached to them, as well as an arbitrary number of associated properties, which don't need to be defined as a schema common to all elements, as seen in figure 4.1.

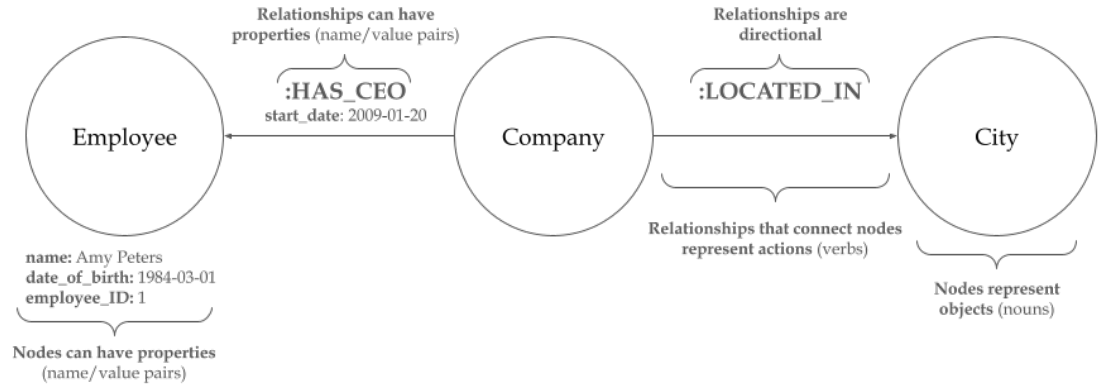

Figure 4.1: Neo4j property graph. Figure recreated from [87, slide 16].

Additional to being a graph database that can run transactional operations (OLTP), it can also project part of the graph (or a subgraph resulting from a query) into memory to run analytical queries (OLAP) and memory-intensive data science algorithms for path

finding, community detection, centrality, similarity, and many others, including Pregel API support for efficient custom-built algorithms [88].

4.3 Usage of graphs and detailed operational information in Trase

Graphs have been used for different types of analysis within Trase. For instance, it is used to analyze the financial flows an ownership of commodity supply chains in general, as well as for different analyses within Brazil's beef supply chain in particular. These two uses are briefly mentioned here, emphasizing the current work around the beef supply chain, while making further explorations on it.
Trase Finance [15] shows direct and indirect exposure of the finance sector to deforestation. This service has been built in partnership between Trase and Neural Alpha [89], who have expertise in using graphs for modeling the complexity of financial and ownership relationships of different actors throughout a supply chain, as seen in figure 4.2.

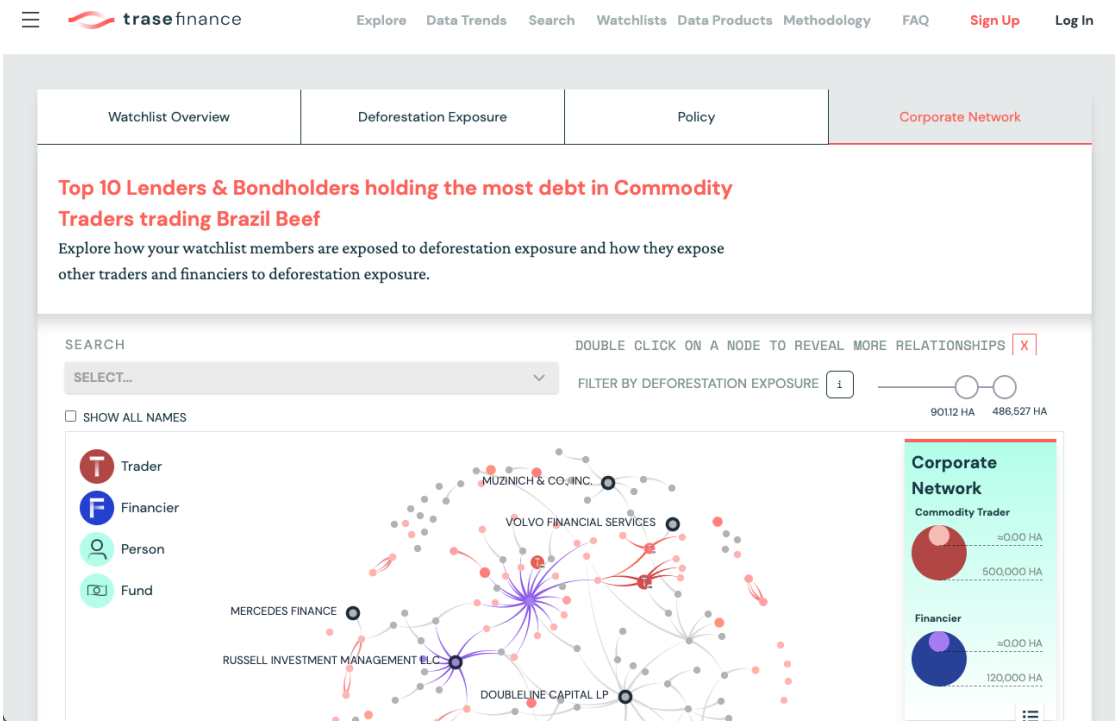

Figure 4.2: Filtering financial actors and their relationships by deforestation exposure

Other example -which is the focus of this section- is in analyzing the traceability of the beef supply chain in Brazil using public records of animal movements (called GTAs - Animal Transit Guide - for its Portuguese acronym). The Animal Transit Guides (GTAs) are the official documents for animal transport in Brazil and contain essential information on beef traceability, such as origin, destination, purpose, species, vaccinations, among others [90]. While GTAs are used and required mainly for sanitary reasons, they offer

a provisional source of information for allowing supply chain transparency. Using the GTA information can be useful to increase the transparency of indirect supply networks and identify the deforestation exposure. Deforestation exposure quantifies the degree to which supply chain entities —such as companies, countries, and investors— are at risk of being associated with commodity-driven deforestation due to their sourcing practices, measured in terms of the deforested area (in hectares) they are connected to [91].
For instance, Trase has conducted research integrating this information and showcasing how less than 2% of rural properties in the Amazon and Cerrado are responsible for 62% of all potentially illegal deforestation [16]. This helps to make the case for investors and companies of the viability to remove direct or indirect supply from deforestation areas without disrupting their supply needs while avoiding reputational and regulatory risks. Trase has also included this data to model the flow of cattle between regions in Brazil, and export destinations, and estimated the "deforestation risk" (in hectares/year) of each supply chain actor over time [92].

4.4 Modeling of the beef supply network in the state of Pará, Brazil

GTAs have detailed information on the animals being transported, which include relevant data to map the different entities and relationships between them, including:

- GTA ID: composed by the state, an alphabetic letter designating the series and six numeric digits
- Source and destination: farm/slaughterhouse name, tax number, city, state
- Animals: How many, of which species, which sex, and age range (in months)
- Transportation date
- Other: destination purpose, transport mode

Information on GTAs within Pará state was previously available, including records stored and cleaned within JSON files including the relevant information (see figure 4.1). In particular, 1,969,303 of such records, ranging from 2013 to 2019 have been used for the modeling and analysis.

```
1 [{
2     "ID": "PA|K|102009",
3     "INFO_STATUS": "EM TRANSITO",
4     "ORIGIN_TAX_NUMBER": "*********",
5     "ORIGIN_NAME": "FAZ SAO MATEUS II",
```

```
    "ORIGIN_FARMER": "CAIO GERONIMO DA SILVA",
    "ORIGIN_CITY": "SAO FELIX DO XINGU",
    "ORIGIN_STATE": "PA",
    "DESTINATION_TAX_NUMBER": "********",
    "DESTINATION_NAME": "*********",
    "DESTINATION_FARMER": "********",
    "DESTINATION_CITY": "SAO FELIX DO XINGU",
    "DESTINATION_STATE": "PA",
    "TRANSPORT": "A PE",
    "SPECIES_PURPOSE": "ENGORDA",
    "DESTINATION_CODE": "15073006602",
    "DESTINATION_GEOCODE": "1507300",
    "ORIGIN_CODE": "15073004972",
    "ORIGIN_GEOCODE": "1507300",
    "MISSING_INFO": "PRESENT",
    "SPECIES": "BOV",
    "TRANSPORT_DATE": "2020-01-07",
    "SLAUGHTER_GTA": false,
    "ANIMALS": [
      {
        "AMOUNT_SENT": 5,
        "DESCRIPTION": "BOVINO,MACHO,13 A 24 MESES"
      },
      {
        "AMOUNT_SENT": 80,
        "DESCRIPTION": "BOVINO,FEMEA,13 A 24 MESES"
      }
    ]
  }]
```


Listing 4.1: Example JSON record of an animal transport record (GTA)

4.4.1 Working prototype: modeling and ingestion of ~2M JSON records in Neo4j

For the prototypes, a Neo4j database running Community Edition Version 5.7.0 has been installed in an Ubuntu Linux 22.04 on a 4 CPU - 16 GB RAM computing node deployed in an Open Stack infrastructure. For the graph data science algorithms, Neo4j graph data science library 2.3.3 was used.

For creating a model based on the GTA information on the JSONs, a Cypher script was created using the APOC library [93], where each record value was saved in a corresponding entity or relationship, with some particular modeling decisions:

- Values from each JSON record is mapped as a property within a corresponding entity
- Entities are created with a MERGE statement, so they are only created once (for example, if the value of a city, state, etc, appears several times, this doesn't create several entities sharing the same value).
- Farms and slaughterhouses are chosen to be uniquely identified by their 'name' and 'tax_number' combination. Future implementations will probably add the city as a third uniqueness criterion to identify more easily some possible anomalies in the original GTA records (for instance, registering inconsistently the municipality a farm or slaughterhouse belongs to, especially when located near the border of a municipality).
- Though farms and slaughterhouses carry the same properties, they are labeled correspondingly to make analysis easier. A ('DESTINATION_NAME', 'DESTINATION_TAX_NUMBER') tuple is created as a slaughterhouse if the 'SLAUGHTER_GTA' property is 'true'.
- A '[:SENDS_TO]' relationship is created between origin and destination farms or slaughterhouses, and includes aggregations of all animals sent historically from one to another, as well as aggregations per year

The ingestion of the JSONs is done at an average rate of 100MB per minute, with each JSON file usually includes all the GTAs of a given year, and is usually between 300 - 500 MB, for a total of almost 5 GB.
Based on this initial modeling, additional entities and relationships are created, to support further analysis. For instance, an entity called 'Animal_Group' represent the subsets of animals sent in every animal movement, grouped by sex, lower, and upper age (in months). The example record shown in the listing **??** includes 2 groups of animals, each of them integrated by different quantities (5, 80) and the characteristics of each of them (sex and age range in months).
When an upper age limit is not stated for an animal group, the system defaults to 42 months, reflecting a common upper limit for animals headed to slaughterhouses. However, another common approach used by Trase is to consider an upper age limit of 60 months (5 years) for cattle to go to the slaughterhouse.
Most queries leverage apoc.periodic.iterate() to execute in transaction batches, enhancing efficiency and reducing memory strain during extensive operations like setting, updating, or deleting millions of records. Given that transactions are atomic —either fully executed or not at all— halting a periodic iterate query only ceases the forthcoming batches yet to be committed.
The following table includes links to the actual scripts to create the model and make some sample queries on it.

Script	Description
`indexes.cypher`	Create indexes of the entity properties that will be queried the most
`load_gtas_json.cypher`	Populate the graph based on values from the JSON file. Uses 'apoc.load.json()' to load the JSON file
`add_animal_groups.cypher`	Create 'Animal_Group' entities, based on the information associated in 'Animal_Type' of a GTA. It uses regular expressions, as to convert 'animal_type' string values such as "BOVINO,MACHO,9 A 12 MESES" into: (sex: MACHO, lower_age: 9, upper_age: 12). When the upper age is not specified, it sets it at 42
`indexes.cypher`	General sample queries

Once the JSONs are ingested and the animal groups added based on it, we can check the structure has been correctly represented by calling the query '`CALL db.schema.visualization()`' on the Neo4j browser page (see figure 4.3).

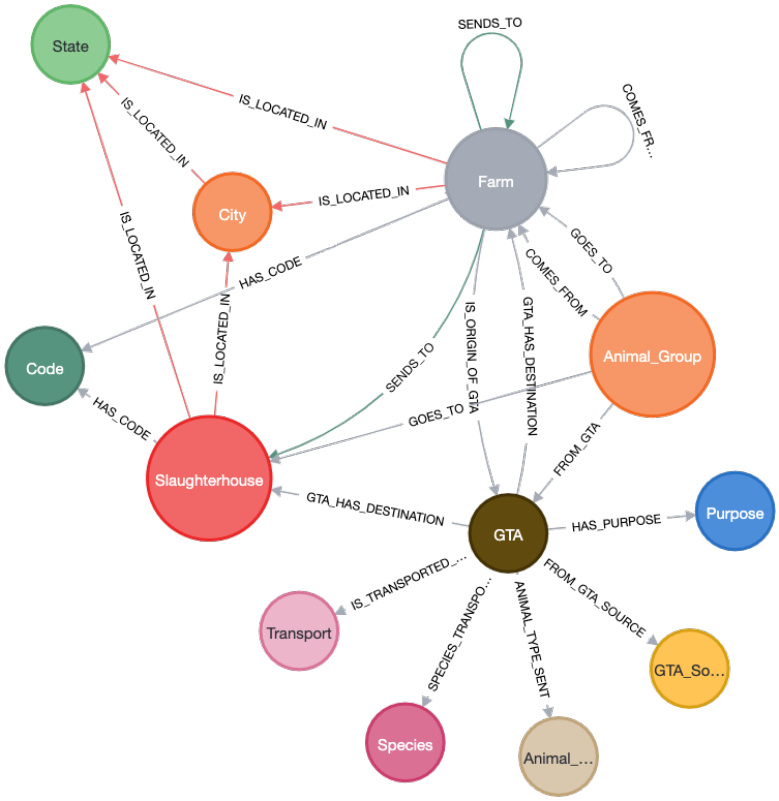

Figure 4.3: Graph model after ingesting the JSON records

Checking the number of values on the main entities, we can see the following total numbers of:

- GTAs: 1,969,303
- Slaughterhouses: 5,262
- Farms: 197,123
- Cities: 1,389
- States: 27 (though the GTAs are issued in Pará, they can have another destination state)
- Animal groups transported: 2,839,437 (animals transported are reported in groups of the same sex and age range)
- Animals transported: 54,809,693 (note that an animal that have been transported multiple times will count that amount of times)

Checking which are the 5 slaughterhouses that have received the most amount of animals, we see that JBS in the city of Maraba, has received a total of 777.036. We will use this JBS - Maraba slaughterhouse as a reference for several explorations.

city	**slaughterhouse**	**animals_received**
MARABA	JBS S/A	777,036
REDENCAO	JBS S/A	684,487
XINGUARA	MERCURIO ALIMENTOS S.A.	661,759
AGUA AZUL DO NORTE	FRIGOL S/A	615,011
CASTANHAL	ATIVO ALIMENTOS EXPORTADORA E IMPORTADORA EIRELI	595,283

For instance, we can quickly find the number of farms connected up to an arbitrary number of degrees of separation to JBS Maraba, by running a query of the form:

```
MATCH (f:Farm)-[:SENDS_TO*1..2]->(s:Slaughterhouse)
WHERE s.tax_number = "#########"
RETURN COUNT(DISTINCT(f))
```

Where a and b in "`[:SENDS_TO*a..b]`" is the range of degrees of separation we are querying for. Running the query for the first degrees of connections to JBS Maraba returns:

- First-degree farms: 1,394
- Farms up to second-degree: 27,501
- Farms up to third-degree: 95,068

- Farms up to fourth-degree: 133,230
- Farms up to fifth-degree: 145,692

Up till the fourth-degree farms from JBS Marabá, already 68% of farms are connected, including 83% of farms within the Marabá municipality.

4.5 Identification of potential indirect supply chain

Information regarding the characteristics of animal groups (including sex and age range) being transported between farms and towards slaughterhouses, as well as the dates of transport, enables the computation of potential connections between farms. Moreover, it allows for the identification of animals traded through indirect connections —those that are two or more degrees of separation away. Specifically, for any given slaughterhouse, we can establish a support group representing the probable number of animals each indirectly connected farm could have supplied to it. This approach extends the findings of the study on the origins of beef [92] (illustrated in figure 4.4), where cattle lacking a direct path to a slaughterhouse (indicated by a dotted purple circle in the figure) or those of differing sex are excluded (shown as gray links in the top-left image of the figure). The analysis in the referenced paper concentrates on inter-municipal transfers and involves calculating aggregate values (translated into animal weight based on the ages of the animals) for the municipalities a slaughterhouse sources from. As detailed in the supplementary section 'Identifying the Source of Cattle Slaughtered per Slaughterhouse' of the cited paper, this process requires increased complexity to accommodate the nuances of the data and the patterns of cattle movement in Brazil. Among other challenges, Guide to Animal Transit documents (GTAs) are recognized as an incomplete source of information —according to the paper, approximately 50% of a municipality's production can be traced through them.

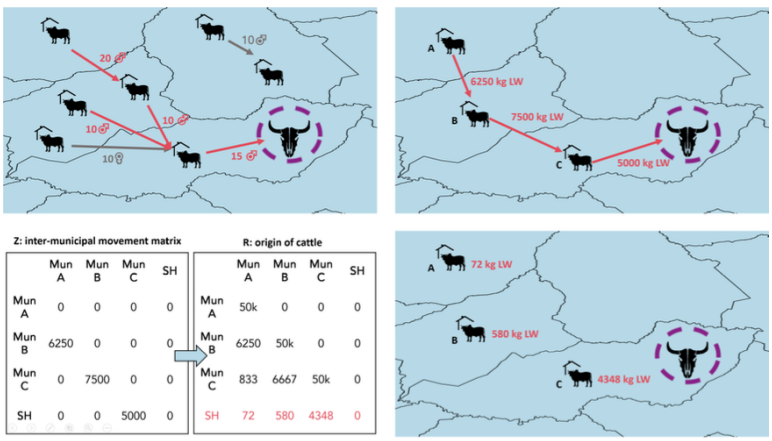

Figure 4.4: Filtering of possible movement paths, and aggregation per municipality. Taken from [92, Supplementary Information]

In this section, the focus is placed on getting detailed information about the possible farm paths, and identifying the specific possible animals that could have gone through those paths, by further filtering based on age ranges and transport dates. While this probably will not be useful for calculating aggregations of municipality production as done previously, it can be the basis to provide additional information, such as:

- Given a slaughterhouse, what is the maximum amount of animals each reachable farm could be sourcing indirectly to it?
- Given a farm sending animals to other farms, which slaughterhouses could these animals have reached and through which paths?
- Which are the most transited paths between indirectly connected farms and slaughterhouses, in terms of animals sent?

Within these questions, certain additional filters can also be useful:

- Can this be easily filtered based on municipality location (from farm and/or slaughterhouses), specific farm or slaughterhouse identifications, and additional characteristics of some farms?
- Can arbitrary time windows be used to filter the queries? For example, a range of dates within which a slaughterhouse received animals or from which animals were transported from a reachable farm?

All of these questions can be better answered if a support set of possible paths of animal movements is created, which integrates some of the constraints already discussed (sex, age range, transport date).
To give an example of a way to build this support set, consider a specific group of 15 male animals of an age range between 25 and 36 months, which were transported from farm 'F_a' to JBS Maraba on 2017-06-22 (see figure 4.5, highlighted node). While 'F_a' has had 127 farms sending 12,866 animals to them (see '`sample_queries.cypher`'), distributed among 305 groups of animals, not all groups of animals reaching 'F_a' could have made part of the specific group of 15 animals highlighted. In the sample image, a sample of three of such animal groups and sending farm possibilities are shown, where the orange circles represent different animal groups and the number of animals within them, and the gray circles represent farms.

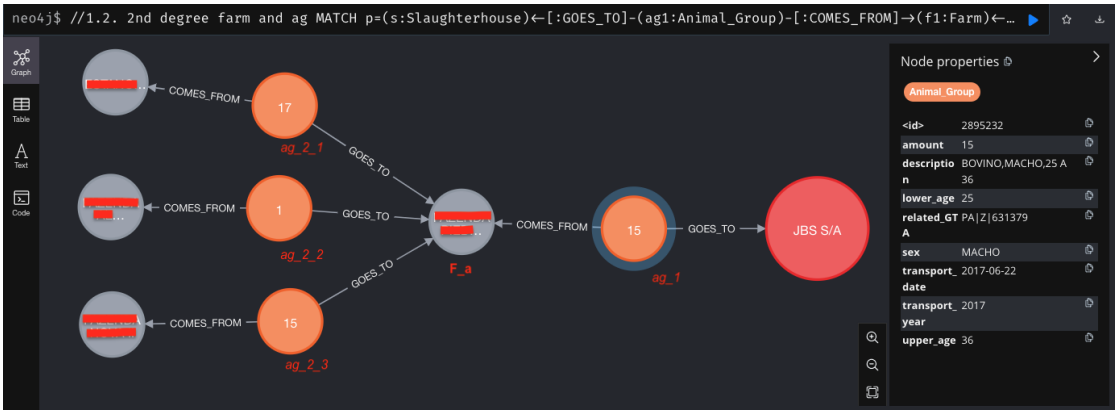

Figure 4.5: Sample paths of indirect animal group movements

4.5.1 Working prototype: constrained support set and identification of possible animals coming from indirect suppliers

In particular, to build the support group based on the conditions of the animal groups as previously mentioned, we create a relationship between each compatible pair of animal groups indirectly connected through a farm. Based on figure 4.5, to know if any of the 'ag_2_i' (animal group connected with a second degree connection) could be a source for ag_1 (animal group directly connected to a slaughterhouse), we check:

- If the sex of ag_2_i is the same as ag_1.
- If the transport date of ag_1 is more recent than ag_2_i.
- If given the age range of ag_2_i and the transport date, the ages are compatible with the age range and transport date of ag_1. For example, if 36 months have passed between the transport date of ag_2_1 and ag_1, and animals in ag_1 are up to 24 months, they wouldn't be compatible.

The 'could_come_from.cypher' script creates such relationships, called '[:COULD_COME_FROM]', which includes the number of possible animals going from one animal group to another, and the maximum possible number of months the animals could have stayed in the transition farm.
Here is the table with the link to related scripts:

Script	Description
could_come_from.cypher	Creates the [:COULD_COME_FROM] relationship (84,897,898 relationships created)

`indirect_suppliers_tables.cypher`	Creates aggregations of indirect suppliers (slaughterhouse centered), showing the possible animals sourced from each reachable farm, per year, and also saves this into a CSV for further analysis.

Based on this `[:COULD_COME_FROM]` relationship, we can now check again which are the farms that are reachable to each slaughterhouse through compatible groups of animal movements (see figure 4.6).

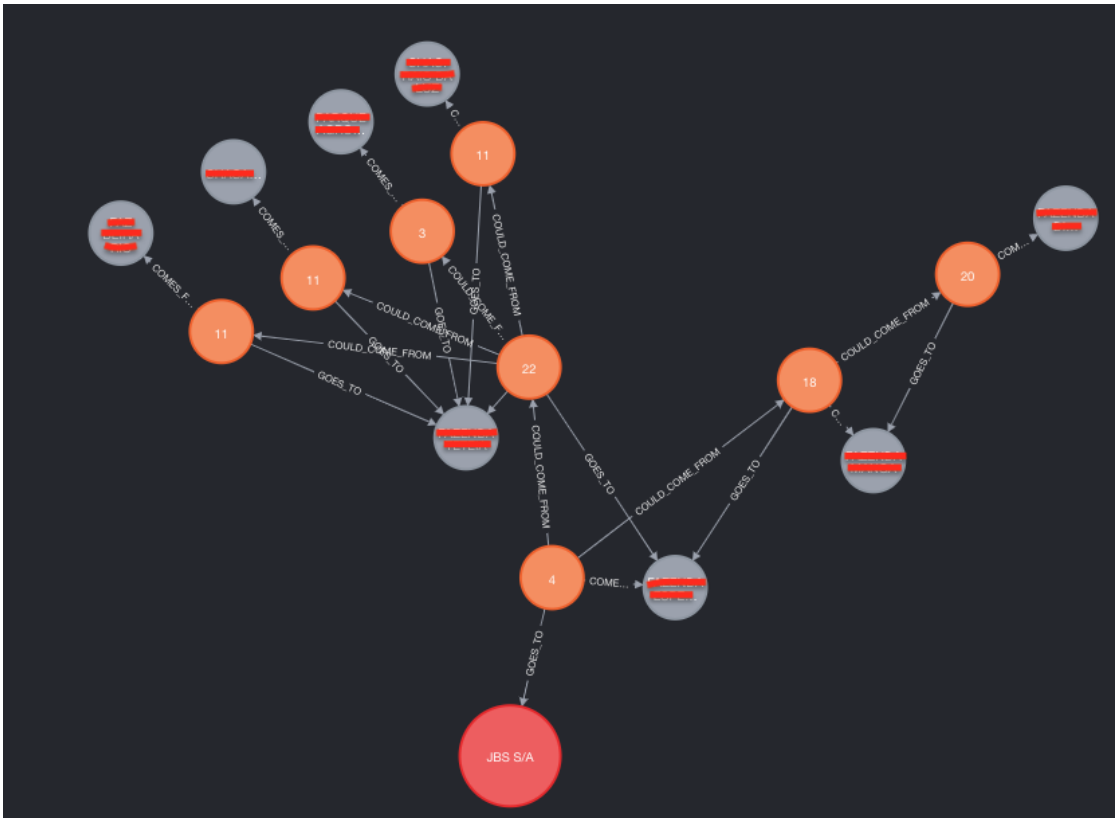

Figure 4.6: Sample paths including a 'COULD_COME_FROM' relationship between animal groups

Taking as example JBS Maraba, we find the following reachable farms for the whole 2013-2020 GTAs dataset:

- First-degree farms: 1,394 (same value, as its a direct connection)
- Farms up to second-degree: 19,483 (70% of originally connected)
- Farms up to third-degree: 52,311 (55% of originally connected)

By only considering farms that are connected only through compatible animal movements, the support set of reachable farms from a slaughterhouse can be more precisely constrained. Also, it allows us to identify the maximum possible number of animals

each farm could have sent to a slaughterhouse through an indirect path, aggregated per degree of separation. This information can be particularly useful, as currently, slaughterhouses only receive direct information from the farms they directly source from. Though some slaughterhouses direct providers must disclose which farms they are sourcing from (especially to comply with export regulations and zero-deforestation agreements), there are at least a couple of problems with this approach:

- Their direct providers can be indirectly sourcing from farms with high exposure to deforestation or human right violations without full awareness of it.
- Direct providers might fail to disclose some of their source farms to the slaughterhouses they send animals to.

Both situations can result in beef exports being reported incorrectly as deforestation and human-rights violations free. By tracing this information, interested slaughterhouses, as well as relevant actors (regulators, researchers, and civil society) can see if a direct provider has high exposure to a risky indirect supply source, or if the direct provider is sourcing from a risky source and seems to be failing to report this. It can help all involved to monitor and address the situation proactively. It can also be useful to drive additional insights, some of which are explored later on.
Figure 4.7 shows a sample of connected animal groups leading to JBS Maraba of up to 3 degrees of separation. Additional to showing possible paths of animal groups, it also allows calculating the maximum number of animals a reachable farm could be sent at a specific degree of separation. In the figure, while for simplicity the farm nodes are not shown, they are marked to belong to farm f1 (direct connection), f2 (2nd degree) or f3 (third degree). For example, f2 could have sent maximum of 40 animals through 'ag1_1' (the sum of its two animal groups, as long as it's lower than 'ag1_1'). Similarly, f3 could have sent, through the 'ag1_1' path, a maximum of animals based on its contribution to 'ag2_1' and 'ag2_2'. The script creating these aggregations is in `'indirect_suppliers_tables.cypher'`.

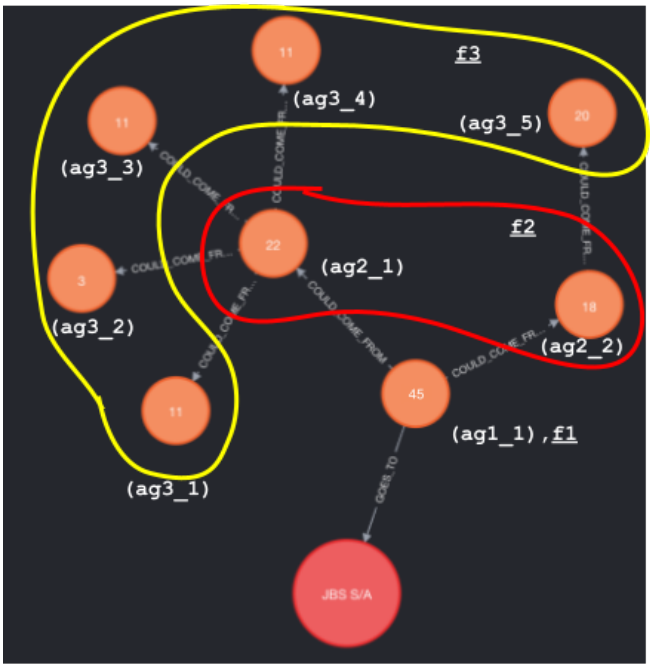

Figure 4.7: Identifying the maximum possible animals a farm indirectly connected could have sent. The red enclosing shows the animal groups related to farm f2, and the yellow enclosing animal groups related to farm f3

For example, table 4.5 is a small sample showing the total possible animals sent to JBS Maraba up to a third degree of connection, per farm, received during the 2019 period and listed in descending order by 3rd_degree animals. As mentioned earlier, this can be useful to corroborate if the 1st-degree providers are including information about 2nd-degree farms, as well as to see if there are some 3rd-degree connections that are worth reviewing. While it would be straightforward to query the possible paths between some of this 3rd-degree farms and the slaughterhouse individually, later we will see a more systematic approach to specifically identify the most transited paths between any farm and all reachable slaughterhouses, using weighted shortest path algorithms.

city	Farm	1st degree	2nd degree	3rd degree
Uruara	**********		5,596	41,659
Placas	**********		8,092	40,117
Medicilandia	**********		10,146	38,625
Altamira	**********		6,661	36,191
Uruara	**********		4,828	35,327

Table 4.5: Top possible indirect animals sent to JBS through different farms, by degree of separation

4.6 Community detection and centrality measurements of farms and slaughterhouses

Identifying the structure of trade relationships between supply chain actors, how they change in time, and possible key drivers of those structures and changes, are relevant to identify and monitor actions and policies for improving sustainability. Trase, as well as other environmental think tanks, have conducted research in this area [94] [95], and refer to it as trade 'stickiness' -defined as the stability of commercial relationships between companies and regions. In [95] the focus is specifically on soy production in Brazil, as the expansion of soy plantations is the second largest direct driver of deforestation [96]. four categories of factors are considered the most influential: economic incentives, institutional factors, social and power dimensions, and biophysical and technological aspects.
Though not part of the scope of the current thesis, a possible area of exploration is identifying relations between graph community and centrality measures and the stickiness measures used in some of Trase's research. In particular,

- Stickiness in trade linkages (unweighted networks), using the temporal correlation coefficient (C) as defined in [97].
- Stickiness in trade flows (weighted networks), using weighted persistence of trade flows (WPi) and the temporal average weighted persistence (TWPi) as defined in [94, Equations 5 and 6].

For example, it could be reviewed if and how the stickiness measurements within graph clusterings (communities) keep similar ranges of values. Also, how centrality and transitive centrality values of influential nodes within these communities relate to the stickiness levels in them. It is worth underlining however that related stickiness research within Trase has been mainly focused on Brazilian soy production (not beef production, which is the emphasis in this section), and that better understanding and discussion would be needed to review the possible value of exploring those relationships.
In the current thesis, basic community and centrality measures are carried on, as will be mentioned later.

4.6.1 Louvain community detection

The Louvain algorithm is a popular method for graph community detection [98], especially used for large networks (thousands or more nodes, where there is bigger potential for identifying meaningful community structures). It aims to identify communities or groups of nodes that are densely connected within themselves while having sparse connections between communities. It aims to optimize a modularity score of each community, which measures the assignment quality of nodes to its community, as defined in [99].

Definition 1 (Modularity Score in a weighted graph)**.** The modularity score captures the extent to which nodes within a community show higher levels of interconnectedness compared to the connectivity expected in a random network. Specifically, the value of modularity is in the range $[-1/2, 1]$, measuring the density of links inside communities compared to links between communities.

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j),$$

where

- A_{ij} represents the edge weight between nodes i and j,
- k_i and k_j are the sum of the weights of the edges attached to nodes i and j respectively,
- m is the sum of all of the edge weights in the graph,
- c_i and c_j are the communities of the nodes,
- $\delta(x, y)$ is 1 if $x = y$, and 0 otherwise.

The Louvain algorithm follows a hierarchical clustering approach, repeatedly merging communities and then applying modularity clustering on the resulting condensed graphs.

4.6.2 PageRank centrality

The PageRank algorithm [100] evaluates the importance of the nodes in a graph by considering the number of incoming connections and the importance of the nodes from which these connections originate. For this, it uses an iterative approach, based on candidate solutions. It assumes that a node's importance is determined by the nodes that link to it, emphasizing that its significance is influenced by the nodes that reference it.
In the original Google paper, PageRank is defined as a function PR as follows.

Definition 2 (PageRank equation)**.**

$$PR(A) = (1 - d) + d \left(\frac{PR(T_1)}{C(T_1)} + \cdots + \frac{PR(T_n)}{C(T_n)} \right)$$

where

- we assume that a node A has nodes T_1 to T_n which point to it.
- d is a damping factor which can be set between 0 (inclusive) and 1 (exclusive). It is usually set to 0.85.
- $C(A)$ is defined as the number of edges going out of node A.

The equation is used to iteratively update a candidate solution and arrive at an approximate solution to the same equation [101].

4.6.3 Working prototype: Louvain community detection and PageRank centrality among farms and slaughterhouses

For using Neo4j's 'graph data science' algorithms [88], a subgraph projection containing the required information to carry out a corresponding algorithm needs to be created [102], and loaded into memory. Each projection created is stored in a 'graph catalog'. This catalog facilitates the loading, reading, and modification of these projections when necessary. When loading these projections, Neo4j adjusts how the entities, relationships, and properties are structured and stored in memory, allowing more efficient analytical processing (OLAP) than the default transactional (OLTP) mode. Additionally, algorithms can be run in stream mode (only producing output), stats mode (generating a summary of statistics useful to calculate required computational resources, tailor performance, identify convergence, and key statistics of the distribution of values generated), mutate mode (modify the projected graph), and write mode (modify underlying data in the graph database).
In particular, for community detection and centrality, we focus on the '`[:SENDS_TO]`' relationship between slaughterhouses and farms and include the 'total_sent' attribute, as well as some year aggregations (2018 to 2020).
Here the table with the link to the cypher script.

Script	**Description**
`community_and_centrality.cypher`	- Creation of the graph projection for running the algorithms - Run Louvain community detection algorithm and writes the community IDs into the nodes - Runs multiple PageRank queries

Some details about the Louvain community detection implementation:

- It is based on [103].
- It uses as weight property the historic '`total_sent`' animals between nodes (farms and/or slaughterhouses). For future analysis, it can be relevant to calculate it also on smaller window frames (yearly for example).
- It uses the default configuration values, including:
  - `maxLevels: 10` (number of levels in which the graph is clustered and then condensed).
  - `maxIterations: 10` (maximum number of iterations that the modularity optimization will run for each level).

- `tolerance: 0.0001` (minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns).
- `includeIntermediateCommunities: false` (currently only returning final community value).

With these configurations, the Louvain algorithm identifies 25.180 communities, from which:

- 85% of farms and slaughterhouses (171,142) are part of the top 20 most populated communities, as seen in figure 4.8.
- 14% of farms and slaughterhouses (29,053) are part of communities with 5 or fewer nodes in it.
- JBS Marabá is part of the 3rd biggest community, composed of 14,537 nodes (14,469 farms and 68 slaughterhouses).

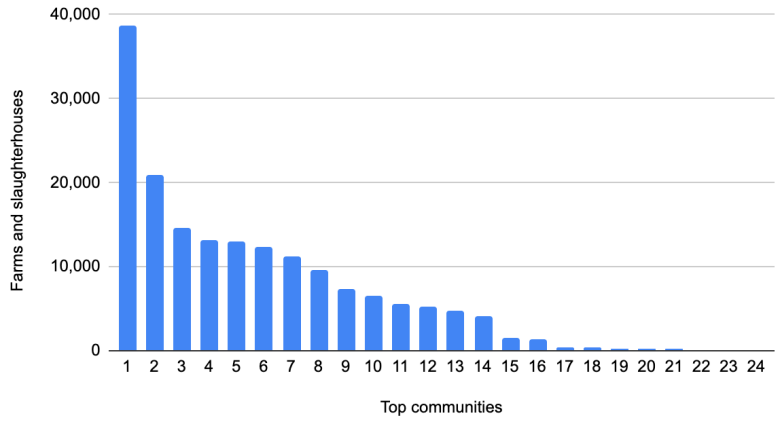

Figure 4.8: Amount of nodes within top ranking communities

Some details about the PageRank implementation:

- It is based on [104].
- It uses as weight property the historic 'total_sent' animals between nodes (farms and/or slaughterhouses). For future analysis, it can be relevant to calculate it also on smaller window frames (yearly for example).
- It uses as a scaler the L1Norm: normalizes each score between 0 and 1, where the sum of all scores is 1. Other options are MinMax, Max, Mean, Log, L2Norm, StdScore, or no scaling.
- It uses the default configuration values, including:

- `dampingFactor: 0.85`. Value between $[0-1)$ that can be interpreted as the probability to jump to a random node, and thus not get stuck in sinks.
- `maxIterations: 20`.
- `tolerance: 0.0000001` (minimum change in scores between iterations before it returns a result).

With these configurations, several exploratory PageRank queries are run (see '`community_and_centrality.cypher`' script), including:

- Explore distribution, as seen in figure 4.9 and 4.10 scatterplots.
- Calculate percentiles.
- Identify farms and slaughterhouses with higher PageRank scores, as seen in tables 4.8 and 4.10.

As can be seen, there is a high level of centrality concentration in a handful of slaughterhouses and farms, including JBS (from Maraba municipality), which has the highest centrality overall.

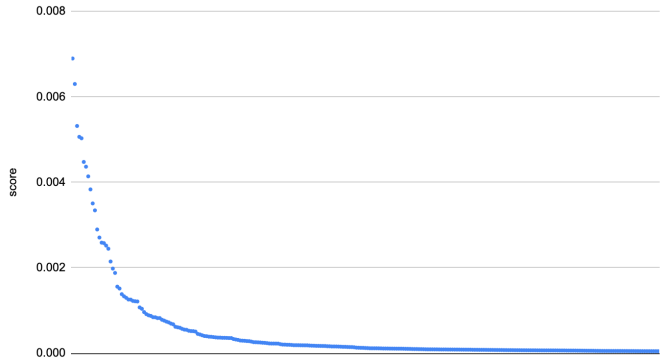

Figure 4.9: Scattered graph sorted in descending order of the L1-normalized PageRank score of top 263 slaughterhouses (top 5%)

slaughterhouse	**score**
JBS S/A	0.006896
ATIVO ALIMENTOS EXPORTADORA E IMPORTADORA EIRELI	0.006300
FRIGORIFICO RIO MARIA LTDA	0.005318
MERCURIO ALIMENTOS S.A. (1)	0.005063
MERCURIO ALIMENTOS S.A. (2)	0.005030

Table 4.8: Top 5 slaughterhouses according to PageRank score

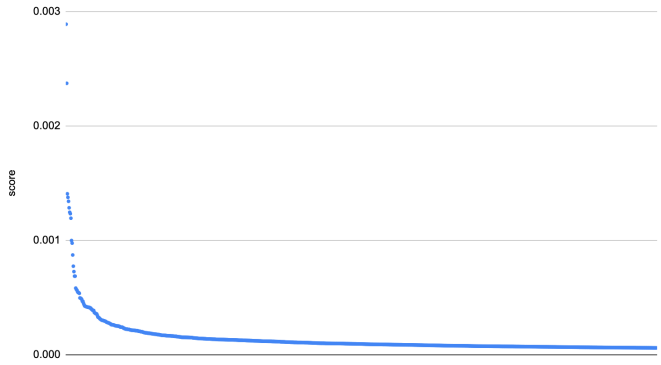

Figure 4.10: Scattered graph sorted in descending order of the L1-normalized PageRank score of the top 985 farms (top 0.05%)

farm rank	score
1	0.002889
2	0.002373
3	0.001407
4	0.001376
5	0.001341

Table 4.10: Top 5 farms according to PageRank score

As mentioned before, the possible use of these explorations and further ones depend on domain-specific research or operational questions. For example:

- Variability of PageRank values throughout the years, including changes of influential nodes and of centrality distribution.
- Relationship between centrality and community identification, such as the relationship between the size of both, and the relationship with stickiness values.
- Patterns and anomaly detection. For instance, examining the relationship between unweighted and weighted PageRank values of nodes (the unweighted taking into account the number of links, though not the number of animals transported). For example, if there is a strong correlation between the percentile ranks of both, but doesn't hold for certain nodes of interest. This could help identify certain qualities of the node (auction node, probable sub-reporting, etc).

4.7 Shortest weighted paths

In the context of graph algorithms, shortest weighted paths algorithms are used to find the most efficient path between two nodes in a graph, where each edge has an associated

weight or cost. These algorithms aim to determine the path with the minimum total weight, considering the weights of the edges traversed.
There are several well-known implementations of shortest-weighted paths algorithms, each with its own characteristics and use cases. Two of the main types of implementations are:

- Dijkstra's Algorithm: used to find the shortest path from a source node to all other nodes, keeping track of the costs of explored nodes, using a priority queue of nodes to explore.
- A* weighted search: integrates a heuristic function for estimating the cost to the goal, and can balance this cost with the cost of an already explored path (using Dijkstra's algorithm).

The specific algorithm used for this thesis is the Delta-Stepping Single-Source Shortest Path [105], which computes all shortest paths between a source node and all reachable nodes in the graph. While it is based on the Dijkstra's algorithm, it differs from its classical implementation in a way that allows for traversing the graph in parallel, making it more efficient when running in multi-threaded environments. These characteristics make it useful for the analysis done, given that several thousand nodes representing farms and slaughterhouses are processed, and that the information from all the shortest paths from each farm to reachable slaughterhouses (up to a limit of degrees) is used.

4.7.1 Working prototype: Identification of most transited paths between slaughterhouses and all farms

As mentioned in a previous subsection, there is a high level of interconnectivity between farms and slaughterhouses. For example, it was mentioned that JBS Marabá can reach 68% (133,230) of all farms on its 4th degree of connection. We have also seen that considering animal group characteristics such as sex, age range, and transport date, the connections and possible animals sent can be reduced.
In Addition to calculating the maximum possible number of animals a farm could have sent to a specific slaughterhouse at any given degree of separation, we can also add relevant information related to the paths between farms and slaughterhouses. In the working prototype, we use the animals transported between any two places to assign weights to a relationship between these places (be them farms or slaughterhouses). First, an intuitive property based on how the weight was calculated will be given, followed by more thorough reasoning based on the work in [106] defining a geometric network model and calculating weighted shortest paths on it.
For each relationship describing animals sent from one place to the other (farm to farm, or farm to slaughterhouse) we assign a weight inversely proportional to the total number of animals sent through it. An intuitive reason for this, especially when using this weight for calculating shortest paths, is that the cost of moving through a relationship

that describes many animals being sent should be lower than that of moving through a relationship that transported few animals. In this way, the shortest weighted path returns the most transited path: the path made of the relationships that included the highest overall number of animals being transported through them. For example, let us have an origin farm called 'O', a destination farm called 'D', and two possible paths to go from 'O' to 'D'. One path goes through farm 'A' and another goes through 'B'. Let us also have that 'O' sent 100 animals to 'A', and 'A' sent 50 animals to 'D', and that 'O' sent 10 animals to 'B', and 'B' sent 1 animal to 'D', as shown in figure 4.11. The shortest path -the path with lower cost- would then be O -> A -> D.

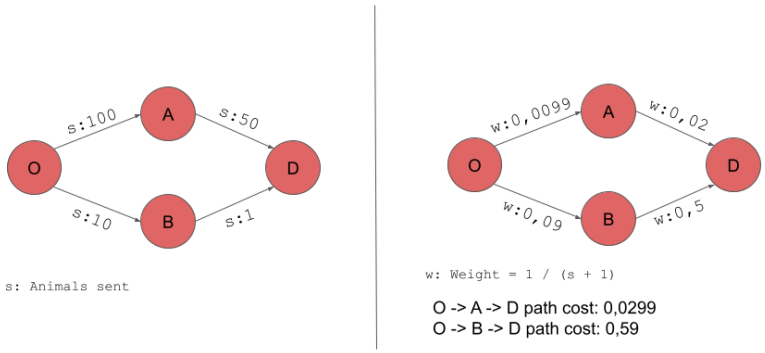

Figure 4.11: Shortest path example, based on animals sent between nodes

The strength of the trade relationship used for the most transited path, is defined by:

- The value $s_{a,b}$, which represent the number of animals that went from node a to node b.
- A directed edge-weight defined by $\hat{p}_{a,b} = 1/(s_{a,b} + 1)$, which specifies the probability of a geometric random variable giving the number of animals sent from node a to node b.

As mentioned, a small weight of $\hat{p}_{a,b}$ corresponds to many animals transported from a to b. And the shortest path between two reachable nodes is the lowest sum of weights within the different possible paths connecting a to b. Thus, the shortest path gives us the path with most animals being transported within it. This can have several potential applications:

- Given a farm, it can show which are the most transited paths to each reachable slaughterhouse, including the cost at each segment of the path. Slaughterhouses with the less short path cost should be 'easier' to reach.
- Given a slaughterhouse, it can show, at each degree of separation, which farms have lower costs to reach.
- Given a segment of a path, it can show which paths are of less cost that go through it. For example, to see which are the shortest paths going

from a slaughterhouse, through a specific farm of interest (for instance, one directly connected) up to indirectly connected farms.

- Given a set of farms of interest (belonging to a specific region, or with other special flag or characteristic), a slaughterhouse can see what are the shortest paths from which they could reach it. Or conversely, starting from these farms, see which slaughterhouses have the lowest cost to reach.

In the following scripts, the supporting weights and relationships are created, as well as the graph projections and specific exploration queries. For the configuration of the Delta-Stepping Single-Source Shortest path, the following values were given:

- `relationshipWeightProperty`: used both the ''total', and 'path_weight' properties of the 'COMES_FROM_2018', and 'COMES_FROM_2018_COMPATIBLE' relationships
- `delta: 2.0` (default). Bucket width for grouping nodes with the same tentative distance to the source node, as described in [107].

Here the table with the links to the cypher scripts.

Script	**Description**
`shortest_paths_creation.cypher`	Shortest paths between each slaughterhouse and all reachable farms, up to 10 degrees of distance - Creates supporting information, specifying as time window animals that could have arrived to a slaughterhouse in 2018 - Creates the graph projection to run analytical queries, and reflects it into the actual graph
`shortest_paths_queries.cypher`	Runs exploration queries based on shortest weighted paths created
`shortest_paths_animal_groups_based.cypher`	Shortest paths between compatible animal groups

As an example, in figure 4.12 a farm that although has 728 reachable slaughterhouses, has no direct connection to one of them. The closest slaughterhouses are 3 degrees of separation away. For each of these slaughterhouses we can see the shortest path to them (made by the id's of the nodes that make this path), the total cost, and the cumulative cost at each step of the path. The specific sample queries to explore this can be found in '`shortest_paths_queries.cypher`'.

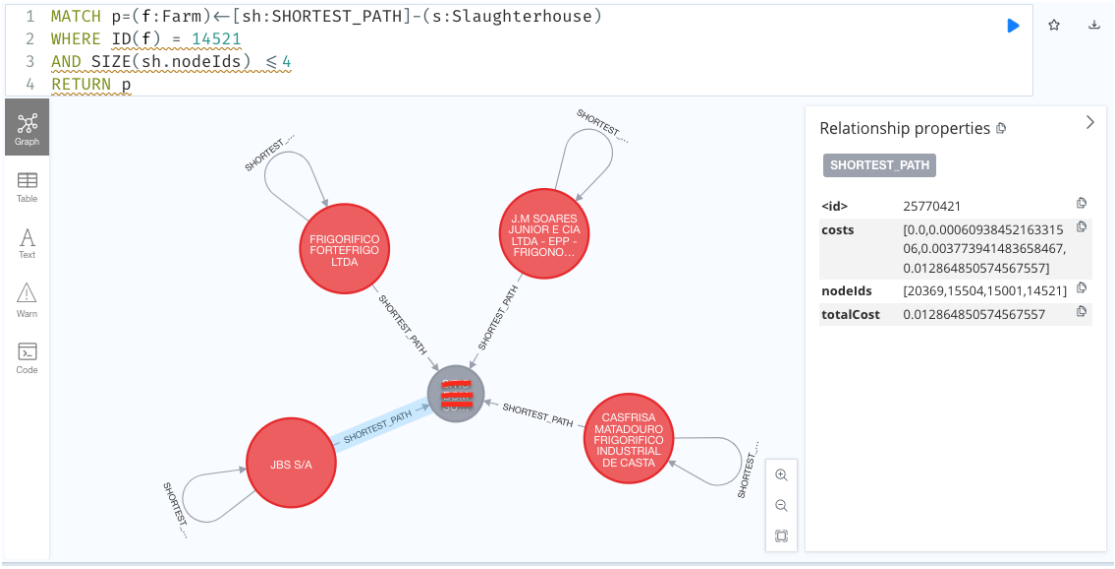

Figure 4.12: Shortest paths from a farm to slaughterhouses maximum 3 degrees away

If instead we sort the shortest paths by total cost without a degrees of separation limitation, the shortest weighted path continues being to JBS, and the next paths in cost are 4 degrees away, as seen in figure 4.13.

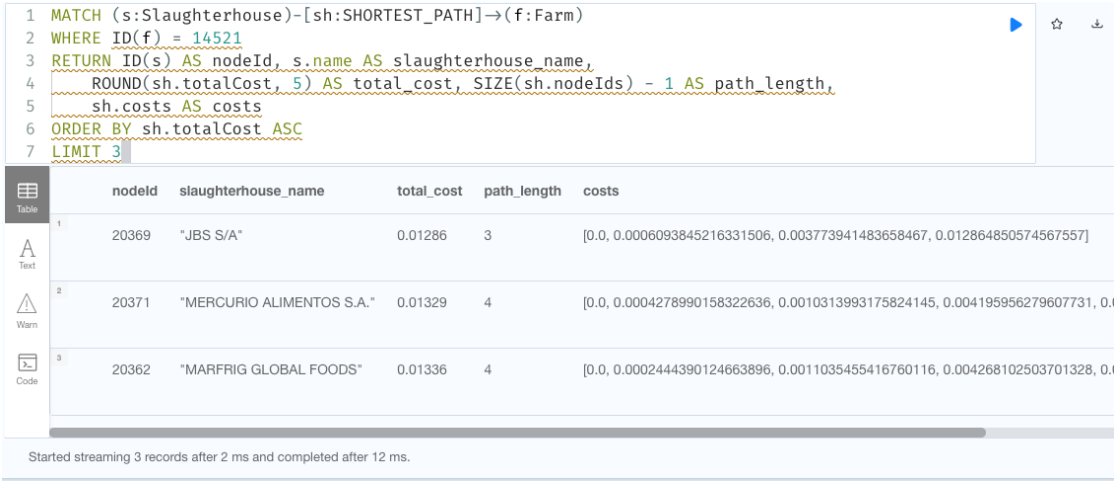

Figure 4.13: Top 3 shortest paths including path length and accumulated cost at each step

Exploring the shortest weighted path with lower cost, as seen in figure 4.14, we can check both the weight and the total number of animals that have traveled through each segment (which is the inverse of its weight). In the example, the first segment has a small weight given the number of animals transported (1,640), in comparison to the second segment (315 animals) and the third (109 animals).

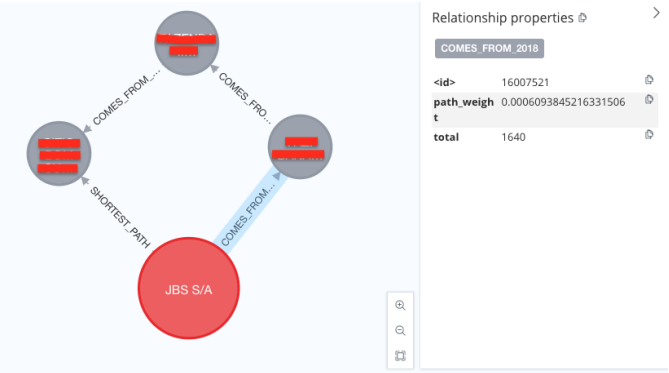

Figure 4.14: Shortest path detail of the first segment, showing the weight cost and the number of animals transported

Other explorations included in '`shortest_paths_animal_groups_based.cypher`', are the creation of shortest paths between animal groups. This allows for more detailed explorations that could be conducted in the future, such as:

- Given a specific animal group that has reached a slaughterhouse, have a way to rank by total cost the paths to other animal groups they could have come from. This gives more precise information of not only which second degree farms could have sourced this animals, but specifically through which group of animals this could have happened.
- Inversely, given a specific animal group transported between two farms, identify and rank by total cost if some of those animals could have reached a specific slaughterhouse (and if it does, identify the specific animal groups reaching the slaughterhouse). When doing this, compare the costs of those animal groups reaching other slaughterhouses.
- Visualize the possible journey of animal groups: dates, purposes (raising, fattening, etc), cities, and GTA observations.

This can allow, for example, for a slaughterhouse to look with more detail at its indirect exposure to certain sourcing farms and animals coming from them. For instance, if some specific animals from farms present in areas linked with deforestation for certain time periods could have reached the slaughterhouse. And if they did, through which specific animal movements throughout the lower cost (most transited) possible paths.

5 Conclusions and future work

5.1 Data engineering and data pipelines

5.1.1 Results

A lakehouse architecture can simplify the creation and management of analytical data pipelines while allowing for more powerful extended uses. While this architecture has seen quick massive adoption within large organizations handling vast amounts of data [42], this thesis explores how it can also be used in other settings in a cost-effective way. In particular, it uses Trase as a case study to review the possibility of transiting to this architecture in a backward-compatible manner, while also relying on open standards and reducing existing technology costs. Some of the main results in relation to these criteria are:

- Compatibility with existing codebase and data, and with Trase's current workflow: an architecture showing how a lakehouse setting can be integrated alongside the current one, as seen in figure 3.11, and allow for a progressive evolution of it.
- Reduce the risk of vendor lock-in: while Databricks is used to implement the lakehouse architecture, the main components used are based on open-source standards, and allow for migrating to other open or vendor-specific solutions if needed, including on-premise configurations.
- Flexible and interconnectable: all main components are connected in a loosely coupled way, which allows for custom configurations, evolution, or partial replacement of them. Also, most components natively allow for connections to popular third-party systems, as well as provide APIs and standard ways to interconnect.
- Keep down technologically related costs: given the simplified architecture and the separation between storage and computing with pay-as-you-go models, several cost efficiencies can be achieved:
  - Some of the current most expensive services can be replaced. For instance, the Amazon RDS for PostgreSQL, or other current services for exploring, accessing, and documenting the datasets.
  - Activate and shut down computing services automatically when using them, instead of paying for idle computing resources.

- As the computing and storage requirements of Trase are currently not so big (total data is under a couple of terabytes, and most computing can work on single node clusters), the pay-as-you-go models would not be so expensive. If more intensive computing and amounts of data are required in the future, costs can scale up and down accordingly.

5.1.2 Cost considerations

Delving more deeply into cost considerations, the two main related costs to the data pipeline implementations are:

- Pay-as-you-go underlying cloud services: mainly, S3 and EC2 instances.
- Pay-as-you-go Databricks usage: Databricks charges a fee (measured in 'Databricks Units' - DBU [108]) based on the computing resources used by the cloud provider. Databricks requires permission to manage these resources from the underlying cloud provider (currently, AWS, GCP, or Azure). Lately, they have also included a 'serverless' option for the SQL Warehouse, where the underlying resources run within Databrick's own pool of running servers. This allows for faster provision of compute instances. For instance, they are operationally available usually within 7 seconds of a request, in contrast to the 3 to 6 minutes that takes to start a new computing cluster. While the hourly cost of the 'serverless' option is higher, the fast availability can also allow for turning off resources on quicker time-outs and resulting in lower total cost.

Components such as DataHub, Airbyte, and Trino require a computing instance to be running. While Airbyte and Trino can be easily started as required, allowing for automatic start and shutdown to reduce costs, DataHub requires a more complex and lengthy startup (approximately 6 to 10 minutes). Also, as it has more points of failure in the starting process, it can require supervision.
Throughout the working prototypes, most of the computing needs (both for Databricks and external components) could be addressed with single nodes of 4CPUs and 16GB of RAM. Although adding an extra node or duplicating a node's capacity cuts by half the processing time of some of the load and pre-processing of bigger datasets (5 - 15 GB) from 2 minutes to 1 minute, this time improvement was not considered sufficiently relevant to explore in more detail.
At the time of writing, the most common EC2 instances used had the following costs, running in AWS eu-west-1 region.

Name and type	**Price per hour**	**Note**

m5d.xlarge (4 CPUs, 16 GB RAM)	US$0.252	Appropriate for most workloads
i3.2xlarge (8 CPUs, 61 GB RAM)	US$0.668	Has the advantage of supporting Delta Caching for quick data re-read, as well as support for Photon acceleration. Useful for more data-intensive pre-processing tasks (for ex., load and manipulation of multi-GB datasets)

5.1.3 Challenges and shortcomings

The main challenges encountered during the thesis are especially related to non-technical and broader engineering considerations than actual implementation limitations. This section broadly presents these challenges, as well as some of the technical ones.

Broader engineering considerations

The adoption of a 'data lakehouse' architecture and its integration into a wider data engineering strategy presents complex challenges. While the proposed solutions tried to be framed within the 'data engineering lifecycle' framework, the breadth of this framework exceeded the constraints of the current thesis, leaving certain aspects unaddressed.
A crucial aspect for correctly addressing Trase's needs and exploring the usefulness of the solutions would be the integration of the data team's participation, which was limited by the scope of a master's thesis. There was a risk of prioritizing technical problems that may not align with the organization's immediate priorities. To mitigate this, a focus on technical explorations was agreed upon, with guidance from Trase supervisors to ensure relevance and applicability.

Technical challenges and risks

The technical challenges primarily revolved around the application of big data technologies for processing small datasets. An initial concern from Trase's team was the use of spark and the orchestration of multi-node clusters for processing small-sized source data, often below 5MB.
Despite these concerns, the exploration of working prototypes and the overall architecture aimed to streamline management and simplify pre-processing tasks. Databricks was chosen for its ability to offer a familiar notebook environment, allowing data scientists to alternate between python, R, SQL, and Scala. This choice aimed to mask the complexities of managing spark across multi-node clusters. However, spark's resource-intensive na-

ture and its dependence on the java virtual machine posed compatibility and efficiency challenges.

Development environment configurations

To address these complexities, various configurations for the development environment were considered and tested, though have not been included in the current thesis report. Options such as native connectors for VS Code or JetBrains, GitHub Codespaces for setting up IDEs, and utilizing python - pandas for reading delta tables via delta sharing were explored. Moreover, the use of a query engine like Trino with a python or R connector was also contemplated.
While these alternatives promise a simplification of the current setup, a comprehensive exploration and understanding of their benefits and necessary adjustments would exceed the scope of this thesis. Further research and testing would be needed to identify the most effective development workflow for the organization.

5.1.4 Future work and alternative solutions

The discussion of future work and alternative solutions highlights areas for further investigation beyond the scope of this thesis.

Pending explorations

While the thesis provided a foundational exploration, several areas remain for future work:

- A comprehensive review of various warehouse and lakehouse implementations can broaden the understanding and possibilities of this architecture.
- An expanded emphasis on metadata usage and management should be pursued, recognizing its critical role in data architecture.

Alternative data architecture implementations

Comparison with other data lakehouse architecture implementations could yield valuable insights. Alternatives to the Databricks Lakehouse that deserve consideration include:

- Cloud warehouse solutions such as Snowflake, which offers cloud-agnostic possibilities, as well as Google's BigQuery and AWS RedShift alongside their comprehensive ecosystems.
- Open-source implementations of the lakehouse architecture, for instance, Apache Iceberg, Apache Hudi, or Delta.io not implemented through Databricks.

- A lean lakehouse architecture that allows for decentralized configurations and more flexibility. This could involve using Delta Tables via Delta-rust and Arrow for memory format framework, and DuckDB for in-process SQL OLAP database operations. The emerging MotherDuck platform [109] could provide a serverless solution, facilitating a blend of centralized and decentralized systems. This approach could allow for local or remote development and execution, fostering greater collaboration and coordination among researchers and external collaborators.

Metadata management in industrial ecology

The role of metadata within the field of industrial ecology requires a bigger domain-specific focus and expertise. While addressing this perspective was an initial consideration, it requires specialized knowledge that was deemed beyond the thesis's scope. Some associated work related with metadata management that would be valuable to explore include:

- The metrics layer, which serves as a structured framework for defining, calculating, and managing business or domain-specific metrics. This layer helps in standardizing how metrics are used across different systems and ensures consistency. In industrial ecology, these metrics could range from environmental impact scores to resource efficiency indicators.
- A data fabric perspective, which provides a flexible and dynamic architecture for integrating various data sources, allowing for real-time access and processing without the need for duplicating data. It enables the aggregation and management of data regardless of where it resides, be it in the cloud, on-premises, or in hybrid environments. This approach does not require the physical movement of data; instead, it creates a virtualized layer that allows users to access and query data without worrying about its original location.
- Semantic graphs, which offer a method for representing information in a graph format. In the context of metadata, semantic graphs can be used to create more intuitive and accessible mappings of data relationships and dependencies. Combined with a data fabric architecture, it can allow for users to explore and query all available and relevant information through it, even if the underlying data is scattered around different systems and formats - hiding this complexity to end users.

Semantic layer and emerging technologies

The semantic layer, along with technologies that facilitate it like knowledge graphs, has seen renewed interest due to advancements in systems' configurability and interconnectivity. These developments allow for the sharing of live data access across organizational

boundaries and underscore the importance of managing metadata and data quality at higher abstraction levels. Tools like Stardog [110] and Cube [111] represent this trend and offer interoperability.
Also, recent advances with Large Language Models (LLM) have seen a growing interest in having them connect to specific datasets as a source of truth on a specifically defined domain. For this, the 'data fabric' pattern, relying on a knowledge graph describing the data assets, can be useful. As the 'data fabric' also provides access to the underlying data, it enables an LLM service to convert a user's natural language request into Sparql, GQL, or Cypher queries [112] and run them. The resulting information can be presented back in various formats such as raw data, a natural language response, or custom charts. Additionally, as an LLM service can further be configured to do training on a specific knowledge base [113], responses can add further context from Trase's insights publications.

5.2 Graph data mining

Most of the explorations and work done in the current thesis is related with basic graph modeling, as well as querying and creating of support sets useful for further analysis. While some graph data science algorithms are used, they are only superficially presented. The main focus is of highlighting some of the possibilities of using a graph database, including the ease of use of a graph query language (such as Neo4j's Cypher), making interactive analysis easy and fast to conduct. For instance, many of the basic analytical queries referenced in this thesis took between milliseconds up to a few seconds to run.
For the modelling, a graph using relevant information available in the GTA records is created, including farms, slaughterhouses, and different levels of aggregation of animal movements between them, as seen in figure 4.3. The current implementation allows for easily ingesting new records (for example from other Brazil regions), as well as additional information from other data sources. Basic statistics of the data are explored, in relation to its connection structure. For example, exploring reachable nodes from a source point, as well as aggregating relationship information -such as animals sent- related to nodes up to a degree of separation, or within a desired structure.
Based on this information, it is possible to easily identify potential indirect supply chains. A first step to do this is applying constraints to possible indirect flows of animals based on sex, age, and transport date. This allows to reduce the real possible reachable farms from each slaughterhouse. Furthermore, it allows identifying the maximum possible amount of animals indirectly sourced to a slaughterhouse from each farm, aggregated by the degree of connection.
Finally, some data mining algorithms are run, including community detection using the Louvain algorithm, centrality measurements using PageRank, and shortest-weighted paths using Delta-Stepping Single-Source Shortest Path. A general description of the algorithms is given, as well as the configurations with which they were run, and ba-

sic analysis based on their results. Some potential uses and further explorations are mentioned.

5.2.1 Cost considerations

A Neo4j database with a community edition is used, which does not have licensing costs. The installation runs in a computing node of 4CPUs and 16GB of RAM, on an Ubuntu 22.04 Linux. For the current explorations done, this setting worked well.
However, it is worth noting some restrictions of the community edition for bigger use cases:

- It can only run on a single machine, so it can't be deployed on a cluster running multiple machines.
- It allows for defining only one user and using one database.
- Allows running maximum 4 threads in parallel. If the machine where it runs has more available CPUs, it won't be able to use them.
- The graph data science machine learning models that can be used are limited to 3, and cannot be persisted for future use.
- It can have a maximum of 34 billion nodes.
- Certain advanced visualization and administration features are not available. For example, it doesn't allow using the Bloom browser [114] for improved graph visualizations, or several management configuration options.

Additionally, Neo4j offers a cloud service called Aura [115], and a self-hosted service that requires a Enterprise License. There are no disclosed specific costs for the Enterprise License: they require contacting sales, which will give a price, also taking in consideration if it will be used by a startup or within education. However, cloud providers allow to buy an annual Enterprise License subscription. For the AWS option, this annual subscription currently costs US$70.080.
For the cloud service, Neo4j offers 3 options:

- AuraDB Free version: can handle up to 200k nodes and 400k relationships
- AuraDB Professional: cost depending on memory (1 - 64GB), CPUs (1 - 12), and storage (2 - 128GB). Hosted in GCP. For a 3 CPU - 16 GB instance, it currently costs US$1.44 per hour ($1.037/month). For a 10 CPU - 48GB instance, it costs US$4.32 per hour.
- AuraDB Enterprise: requires a personalized quote. Allows instances between 4GB - 384GB of RAM, which can be hosted in AWS, Azure, and GCP.

5.2.2 Challenges and shortcomings

This section outlines the challenges encountered and the shortcomings identified during the thesis process, with a focus on the use of graphs within Trase and the implications for data engineering.

Data ingestion and pre-processing challenges

The use of graphs in Trase currently faces challenges primarily in data ingestion and pre-processing. A lakehouse architecture, which orchestrates regular data ingestion, can facilitate the integration and processing of graph query outputs with other datasets and external services. Although not covered in the scope of this thesis, the foundation laid here aims to enable such exploration in the future.

Application of constraints and modeling decisions

Throughout the thesis, several specific shortcomings are identified concerning the application of constraints and modeling decisions:

- The 'Species_Purpose' field in the GTAs could be used to apply additional constraints and analyses. For instance, cattle transported for exportation can be presumed to not move further on to other farms. Similarly, special conditions such as quarantine status or cattle presented at auctions can require additional considerations.
- The management of the upper age limit for cattle could be better managed. While the current age limit is set at 42 months, Trase recognizes an upper limit of 60 months. The modeling could accommodate a flexible age cap for cases where the limit is unspecified, by assigning this condition as a property.
- The identification of unique farms could be improved by incorporating city geocode along with the 'name' and 'tax number' combination. This inclusion would help distinguish farms with identical names and tax numbers but located in different cities, thereby refining municipality aggregations and aiding in the detection of data inconsistencies.
- The use of time windows in queries and support sets was overly restrictive and imprecisely defined. For example, the support set for the shortest paths algorithm, which was designed for animals arriving at a slaughterhouse in 2018, resulted in an overly complex and cluttered database. A more dynamic creation of the support set, allowing for time windows to be defined at query time, would streamline the process. Implementing a native projection that includes 'Animal_Groups' and their 'COULD_COME_FROM' relationships could be more effective.

As stated previously, the focus and results of the graph data mining section are mainly related towards exploring capabilities of the graph engine, the modeling and ingestion of

the data, and testing the use of certain algorithms. Though defining and testing hypothesis regarding the indirect supply chains, and discussing the methods and algorithms used for this could have been very valuable, it hasn't been included and will be referred to within the future work.

5.2.3 Future work

Relevant future work includes defining and testing specific hypothesis related with the indirect supply chain, improving the ingestion and pre-process pipeline of the source data, incorporating a bigger GTA dataset, and extending results and methods to other supply chains. Regarding specific improvements within the algorithms and analysis done in the thesis, and which kind of hypothesis could be relevant to explore, these have been mentioned in the graph data mining section wherever deemed relevant.
Regarding the pipeline and data engineering part, a lakehouse architecture could be useful to set up and orchestrate the ingestion and pre-processing of the source data, which is one of the current bottlenecks. Graph analysis could also be used for data cleaning and quality, such as merging identities of assets that might refer to the same entity even if they have different identification values, or for identifying anomalies. Also, a bigger set of GTAs could be integrated -not only those from Pará state-, as well as datasets containing additional data about the slaughterhouses, farms, and other relevant information.
Information on possible indirect suppliers of a slaughterhouse, and the total possible animals coming from those sources could be integrated into the "Do Pasto ao Prato" app [116], which is a joint project between Stockholm Environment Institute, UCLouvain, and Reporter Brasil.
Finally, regarding the analysis of the indirect supply chains, one of the areas that where most discussed during the development of the thesis, although finally remained out of scope, was the attempt to generate likelihood estimations. In particular, exploring likelihood measurements that certain farms -and specific animal groups- can actually be indirectly connected. This can prove specially challenging as there doesn't seem to be data that can be used as ground truth -for instance, having samples of marked single animals moving along farms throughout their lives-, and as certain dynamics such as auctions, under-reporting, among others, might not be adequately captured through GTAs. However, there is an intuition that a most transited path should be more likely and with greater risk to contain actual indirect connections than a less transited one. One of the discussed ways of exploring this is assigning probabilities that a specific animal group could have come from another specific animal group from a directly connected farm -or from none of them-, based on the number of compatible animals received (matching sex, age, and compatible date) from each of these farms. After this, the graph can be traversed with random walk, and the results of various runs of it reviewed.

References

[1] anaconda.com, *State of Data Science 2020*, 2020. [Online]. Available: https://www.anaconda.com/state-of-data-science-2020.

[2] M. Zaharia, A. G. 0002, R. Xin, and M. Armbrust, "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics," in *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*, www.cidrdb.org, 2021. [Online]. Available: http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf.

[3] W. H. Inmon, *Building the Data Warehouse*. USA: John Wiley and Sons, Inc., 1992.

[4] *Solution Comparison for Cloud Data Warehouse Platforms*, Jul. 2021. [Online]. Available: https://www.gartner.com/en/documents/4003294.

[5] S. Fernandes and J. Bernardino, "What is BigQuery?" *ACM International Conference Proceeding Series*, vol. 0, no. CONFCODENUMBER, pp. 202–203, Jul. 2015. [Online]. Available: https://dl.acm.org/doi/10.1145/2790755.2790797.

[6] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, and Mengchu Cai, "Amazon Redshift re-invented," in *SIGMOD/PODS*, 2022. [Online]. Available: https://www.amazon.science/publications/amazon-redshift-re-invented.

[7] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, "The Snowflake Elastic Data Warehouse Keywords," [Online]. Available: http://dx.doi.org/10.1145/2882903.2903741.

[8] *Azure Synapse Analytics | Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/en-us/products/synapse-analytics.

[9] Stockholm Environment Institute and Global Canopy, *Trase Homepage*, 2023. [Online]. Available: https://www.trase.earth/.

[10] *Databricks Customer Stories | Databricks*. [Online]. Available: https://www.databricks.com/customers.

[11] J. Reis and M. Housley, *Fundamentals of data engineering*. Sebastopol, CA: O'Reilly Media, Jul. 2022.

[12] *Syllabus for Project in Data Science 15 hp.pdf: Project in Data Science 1DL507 11012 HT2022*. [Online]. Available: https://uppsala.instructure.com/courses/65853/files/3823774?module_item_id=611196.

[13] N. Martín and O. Åsbrink, "Data Lakehouse architecture for data pipelines: implementation in supply chains case study," Uppsala Universitet, Uppsala, Tech. Rep., January 2023. [Online]. Available: https://github.com/nmartinbekier/project_course_1DL507/blob/main/Data_Lakehouse_architecture_for_data_pipelines.pdf.

[14] M. Titley, *An introduction to Trase*, Stockholm, March 2023.

[15] Stockholm Environment Institute, Global Canopy, and Neural Alpha, *Trase Finance*. [Online]. Available: https://trase.finance/.

[16] R. Rajão, B. Soares-Filho, F. Nunes, J. Börner, L. Machado, D. Assis, A. Oliveira, L. Pinto, V. Ribeiro, L. Rausch, H. Gibbs, and D. Figueira, "The rotten apples of Brazil's agribusiness," *Science*, vol. 369, no. 6501, pp. 246–248, Jul. 2020. [Online]. Available: https://www.science.org/doi/10.1126/science.aba6646.

[17] *Parliament adopts new law to fight global deforestation | News | European Parliament*. [Online]. Available: https://www.europarl.europa.eu/news/en/press-room/20230414IPR80129/parliament-adopts-new-law-to-fight-global-deforestation.

[18] Stockholm Envirnoment Institute and Global Canopy, *Trase Insights - Supporting France's plan to increase transparency over deforestation*, August 2022. [Online]. Available: https://insights.trase.earth/insights/supporting-france-s-plan-to-increase-transparency-over-deforestation/.

[19] *DataCite Commons: Trase Insights*. [Online]. Available: https://commons.datacite.org/repositories/4ygc95x.

[20] J. Godar, U. M. Persson, E. J. Tizado, and P. Meyfroidt, "Towards more accurate and policy relevant footprint analyses: Tracing fine-scale socio-environmental impacts of production to consumption," *Ecological Economics*, vol. 112, pp. 25–35, April 2015.

[21] "SEI-PCS Indonesia palm oil v1.2 supply chain map: Data sources and methods," 2022. [Online]. Available: https://doi.org/10.48650/ZY8Z-F795.

[22] Various, "Proceedings of 1984 IEEE First International Conference on Data Engineering," in *IEEE First International Conference on Data Engineering*, IEEE, April 1984. [Online]. Available: https://www.computer.org/csdl/proceedings/icde/1984/12OmNynsbDh.

[23] *Google Trends*. [Online]. Available: https://trends.google.com/trends/.

[24] J. Furbush, *Data engineering: A quick and simple definition*, Jul. 2018. [Online]. Available: https://www.oreilly.com/content/data-engineering-a-quick-and-simple-definition/.

[25] anaconda.com, *State of Data Science 2022*, 2022. [Online]. Available: https://www.anaconda.com/state-of-data-science-2022.

[26] M. Rogati, *The AI hierarchy of needs*, Jun. 2017. [Online]. Available: https://hackernoon.com/the-ai-hierarchy-of-needs-18f111fcc007.

[27] A. H. Maslow, "A theory of human motivation.," *Psychological Review*, vol. 50, no. 4, pp. 370–396, Jul. 1943.

[28] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, and J. Gonzalez, "Apache Spark: A Unified Engine for Big Data Processing," *COMMUNICATIONS OF THE ACM*, vol. 59, no. 11, 2016.

[29] M. J. Sax, "Apache Kafka," *Encyclopedia of Big Data Technologies*, pp. 1–8, 2018. [Online]. Available: https://link.springer.com/referenceworkentry/10.1007/978-3-319-63962-8_196-1.

[30] The Open Group, *TOGAF Version 9.1, an Open Group Standard*. [Online]. Available: https://pubs.opengroup.org/architecture/togaf91-doc/arch/index.html.

[31] *What is a data lake?* [Online]. Available: https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/.

[32] *Database - IBM Db2*. [Online]. Available: https://www.ibm.com/products/db2/database.

[33] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, November 1983. [Online]. Available: https://dl.acm.org/doi/10.1145/289.291.

[34] S. Shaw, A. F. Vermeulen, A. Gupta, and D. Kjerrumgaard, "Hive Architecture," *Practical Hive*, pp. 37–48, 2016. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4842-0271-5_3.

[35] *trase - Splitgraph*. [Online]. Available: https://www.splitgraph.com/trase.

[36] *Trino | Distributed SQL query engine for big data*. [Online]. Available: https://trino.io/.

[37] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yigitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: SQL on Everything," [Online]. Available: https://aws.amazon.com/athena.

[38] *Databricks - Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Databricks.

[39] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. Szafrá nski, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, M. Zaharia, and U. Berkeley, "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores," [Online]. Available: https://doi.org/10.14778/3415478.3415560.

[40] *Home | Delta Lake*. [Online]. Available: https://delta.io/.

[41] M. Poess, R. Othayoth Nambiar, and D. Walrath, "Why You Should Run TPC-DS:A Workload Analysis," 2007.

[42] *Data Lakehouse Platform by Databricks*. [Online]. Available: https://www.databricks.com/product/data-lakehouse.

[43] *datahub-project/datahub: The Metadata Platform for the Modern Data Stack*. [Online]. Available: https://github.com/datahub-project/datahub.

[44] *delta-io/delta-rs: A native Rust library for Delta Lake, with bindings into Python*. [Online]. Available: https://github.com/delta-io/delta-rs.

[45] *PyArrow - Apache Arrow Python bindings — Apache Arrow v13.0.0*. [Online]. Available: https://arrow.apache.org/docs/python/index.html.

[46] M. Raasveldt and H. Mühleisen, "DuckDB: An embeddable analytical database," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1981–1984, Jun. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3299869.3320212.

[47] W. Bugden, A. Alahmar, and C. Author, "Rust: The Programming Language for Safety and Performance," *Igscong'22*, no. June, Jun. 2022. [Online]. Available: https://arxiv.org/abs/2206.05503v1.

[48] *MinIO | High Performance, Kubernetes Native Object Storage*. [Online]. Available: https://min.io/.

[49] *Swift - OpenStack*. [Online]. Available: https://wiki.openstack.org/wiki/Swift.

[50] *Open Source Cloud Computing Infrastructure - OpenStack*. [Online]. Available: https://www.openstack.org/.

[51] *Unity Catalog - Databricks*. [Online]. Available: https://www.databricks.com/product/unity-catalog.

[52] *Data Catalog and crawlers in AWS Glue - AWS Glue*. [Online]. Available: https://docs.aws.amazon.com/glue/latest/dg/catalog-and-crawler.html.

[53] *Trino | A gentle introduction to the Hive connector*. [Online]. Available: https://trino.io/blog/2020/10/20/intro-to-hive-connector.html.

[54] M. Hausenblas and J. Nadeau, "Apache Drill: Interactive Ad-Hoc Analysis at Scale," *https://home.liebertpub.com/big*, vol. 1, no. 2, pp. 100–104, Jun. 2013. [Online]. Available: https://www.liebertpub.com/doi/10.1089/big.2013.0011.

[55] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Y. Cloudera, "Impala: A Modern, Open-Source SQL Engine for Hadoop," [Online]. Available: http://impala.io/.

[56] *Fully Managed Relational Database - Amazon RDS - Amazon Web Services*. [Online]. Available: https://aws.amazon.com/rds/.

[57] *MySQL*. [Online]. Available: https://www.mysql.com/.

[58] *Database | Oracle*. [Online]. Available: https://www.oracle.com/database/.

[59] *Microsoft SQL Server*. [Online]. Available: https://www.microsoft.com/en-us/sql-server.

[60] *PostgreSQL: The world's most advanced open source database*. [Online]. Available: https://www.postgresql.org/.

[61] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010*, 2010.

[62] *MongoDB: The Developer Data Platform | MongoDB*. [Online]. Available: https://www.mongodb.com/.

[63] *CSV Files - Spark 3.4.0 Documentation*. [Online]. Available: https://spark.apache.org/docs/latest/sql-data-sources-csv.html.

[64] *JSON Files - Spark 3.4.0 Documentation*. [Online]. Available: https://spark.apache.org/docs/latest/sql-data-sources-json.html.

[65] *What is Auto Loader? | Databricks on AWS*. [Online]. Available: https://docs.databricks.com/ingestion/auto-loader/index.html.

[66] *Add tests to your DAG | dbt Developer Hub*. [Online]. Available: https://docs.getdbt.com/docs/build/tests.

[67] *awslabs/deequ: ibrary built on top of Apache Spark for defining "unit tests for data"*. [Online]. Available: https://github.com/awslabs/deequ.

[68] *great-expectations/great_expectations: Always know what to expect from your data.* [Online]. Available: https://github.com/great-expectations/great_expectations.

[69] *airbytehq/airbyte: Data integration platform for ELT pipelines from APIs, databases & files to warehouses & lakes.* [Online]. Available: https://github.com/airbytehq/airbyte.

[70] *What is Databricks Workflows? | Databricks on AWS*. [Online]. Available: https://docs.databricks.com/workflows/index.html.

[71] *Apache Airflow*. [Online]. Available: https://airflow.apache.org/.

[72] *What is Delta Live Tables? | Databricks on AWS*. [Online]. Available: https://docs.databricks.com/delta-live-tables/index.html.

[73] *Databricks runtime releases | Databricks on AWS*. [Online]. Available: https://docs.databricks.com/release-notes/runtime/releases.html.

[74] *CONSTRAINT clause | Databricks on AWS*. [Online]. Available: https://docs.databricks.com/sql/language-manual/sql-ref-syntax-ddl-create-table-constraint.html.

[75] *Home | Delta Lake*. [Online]. Available: https://delta.io/.

[76] *DataHub: Popular metadata architectures explained | LinkedIn Engineering*. [Online]. Available: https://engineering.linkedin.com/blog/2020/datahub-popular-metadata-architectures-explained.

[77] *S3 Data Lake | DataHub*. [Online]. Available: https://datahubproject.io/docs/generated/ingestion/sources/s3/.

[78] *datahub/proxy.py at master · datahub-project/datahub · GitHub*. [Online]. Available: https://github.com/datahub-project/datahub/blob/master/metadata-ingestion/src/datahub/ingestion/source/unity/proxy.py.

[79] *Developing on Metadata Ingestion | DataHub*. [Online]. Available: https://datahubproject.io/docs/metadata-ingestion/developing/.

[80] *DataHub GraphQL API | DataHub*. [Online]. Available: https://datahubproject.io/docs/api/graphql/overview/.

[81] *delta/PROTOCOL.md at master · delta-io/delta · GitHub*. [Online]. Available: https://github.com/delta-io/delta/blob/master/PROTOCOL.md.

[82] *Supply Chain Graph Database Use Cases: Data Management & Visualization*. [Online]. Available: https://neo4j.com/use-cases/supply-chain-management/.

[83] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus, "Knowledge Discovery in Databases: An Overview," *AI Magazine*, vol. 13, no. 3, pp. 57–57, Sep. 1992.

[Online]. Available:
https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1011.

[84] *Native vs. Non-Native Graph Database Architecture & Technology*. [Online]. Available: https://neo4j.com/blog/native-vs-non-native-graph-technology/.

[85] *DB-Engines Ranking per database model category*. [Online]. Available: https://db-engines.com/en/ranking_categories.

[86] *10 Reasons For Choosing Neo4j*. [Online]. Available: https://neo4j.com/top-ten-reasons/.

[87] *The art of the possible with graph technology_Neo4j GraphSummit Dublin 2023.pptx*. [Online]. Available: https://www.slideshare.net/neo4j/the-art-of-the-possible-with-graph-technologyneo4j-graphsummit-dublin-2023pptx.

[88] *Graph algorithms - Neo4j Graph Data Science*. [Online]. Available: https://neo4j.com/docs/graph-data-science/current/algorithms/.

[89] *Neural Alpha*. [Online]. Available: https://www.neuralalpha.com/.

[90] *Habilitar-se para emissão da Guia de Trânsito Animal*. [Online]. Available: https://www.gov.br/pt-br/servicos/habilitar-se-para-emissao-da-guia-de-transito-animal/#GTAs.

[91] "Commodity deforestation exposure and carbon emissions assessment," 2022. [Online]. Available: https://doi.org/10.48650/GE7V-Q043.

[92] E. K. Zu Ermgassen, J. Godar, M. J. Lathuillière, P. Löfgren, T. Gardner, A. Vasconcelos, and P. Meyfroidt, "The origin, supply chain, and deforestation risk of Brazil's beef exports," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 117, no. 50, pp. 31 770–31 779, December 2020. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.2003270117.

[93] *Awesome Procedures On Cypher (APOC) - Neo4j Labs*. [Online]. Available: https://neo4j.com/labs/apoc/.

[94] T. N. d. Reis, P. Meyfroidt, E. K. zu Ermgassen, C. West, T. Gardner, S. Bager, S. Croft, M. J. Lathuillière, and J. Godar, "Understanding the Stickiness of Commodity Supply Chains Is Key to Improving Their Sustainability," *One Earth*, vol. 3, no. 1, pp. 100–115, Jul. 2020.

[95] T. N. d. Reis, V. Ribeiro, R. D. Garrett, T. Kuemmerle, P. Rufin, V. Guidotti, P. C. Amaral, and P. Meyfroidt, "Explaining the stickiness of supply chain relations in the Brazilian soybean trade," *Global Environmental Change*, vol. 78, p. 102 633, January 2023.

[96] X.-P. Song, M. C. Hansen, P. Potapov, B. Adusei, J. Pickering, M. Adami, A. Lima, V. Zalles, S. V. Stehman, C. M. Di Bella, M. C. Conde, E. J. Copati, L. B. Fernandes, A. Hernandez-Serna, S. M. Jantz, A. H. Pickens, S. Turubanova, and A. Tyukavina, "Massive soybean expansion in South America since 2000 and implications for conservation," *Nature Sustainability*, vol. 4, no. 9, pp. 784–792, Jun. 2021.

[97] V. Nicosia, J. Tang, C. Mascolo, M. Musolesi, G. Russo, and V. Latora, "Graph Metrics for Temporal Networks," Jun. 2013.

[98] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026 113, February 2004.

[99] V. D. Blondel, J. L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, P10008, October 2008. [Online]. Available: https://iopscience.iop.org/article/10.1088/1742-5468/2008/10/P10008%20https://iopscience.iop.org/article/10.1088/1742-5468/2008/10/P10008/meta.

[100] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, April 1998.

[101] *PageRank - Neo4j Graph Data Science*. [Online]. Available: https://neo4j.com/docs/graph-data-science/current/algorithms/page-rank/.

[102] *Projecting graphs using native projections - Neo4j Graph Data Science*. [Online]. Available: https://neo4j.com/docs/graph-data-science/current/management-ops/projections/graph-project/.

[103] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel Heuristics for Scalable Community Detection," *Parallel Computing*, vol. 47, pp. 19–37, October 2014. [Online]. Available: https://arxiv.org/abs/1410.1237v2.

[104] B. Manaskasemsak and A. Rungsawang, "An Efficient Partition-Based Parallel PageRank Algorithm," in *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, IEEE, pp. 257–263.

[105] U. Meyer and P. Sanders, "delta-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, pp. 114–152, 2003. [Online]. Available: www.elsevier.com/locate/jalgor.

[106] R. Sainudiin, K. Yogeeswaran, K. Nash, and R. Sahioun, "Characterizing the Twitter network of prominent politicians and SPLC-defined hate groups in the 2016 US presidential election," *Social Network Analysis and Mining*, vol. 9, no. 1, p. 34, December 2019.

[107] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing Ordered Graph Algorithms with GraphIt," November 2019.

[108] *AWS Pricing | Databricks*. [Online]. Available: https://www.databricks.com/product/aws-pricing.

[109] *MotherDuck: Serverless Data Analytics with DuckDB*. [Online]. Available: https://motherduck.com/.

[110] *The Enterprise Knowledge Graph Platform | Stardog*. [Online]. Available: https://www.stardog.com.

[111] *Cube — Semantic Layer for Building Data Applications*. [Online]. Available: https://cube.dev/.

[112] *Context-Aware Knowledge Graph Chatbot With GPT-4 and Neo4j*. [Online]. Available: https://neo4j.com/developer-blog/context-aware-knowledge-graph-chatbot-with-gpt-4-and-neo4j/.

[113] *LangChain*. [Online]. Available: https://lablab.ai/tech/langchain.

[114] *About Neo4j Bloom - Neo4j Bloom*. [Online]. Available: https://neo4j.com/docs/bloom-user-guide/current/about-bloom/.

[115] *Neo4j Aura overview - Neo4j Aura*. [Online]. Available: https://neo4j.com/docs/aura/.

[116] *Do pasto ao prato — juntos, aumentando a transparência da cadeia da carne no Brasil*. [Online]. Available: https://www.dopastoaoprato.com.br/.