Data Lakehouse architecture for data pipelines: implementation in supply chains case study

1st Nicolás Martín
Data Science Master Programme
Uppsala University
Uppsala, Sweden
nicolas.martin.bekier@gmail.com

2nd Oskar Åsbrink
Data Science Master Programme
Uppsala University
Uppsala, Sweden
asbrinkoskar@gmail.com

***Abstract*—Managing data pipelines usually involves multi-tier architectures that allow the processing of unstructured and structured data in batches or streams. A common approach is to have a Data Lake (a central repository usually of raw data) which after several ETL (Extract, Transform, Load) operations feeds one or more Warehouses (integrated repositories of structured data), which in turn can feed various applications (e.g. business intelligence, data science, machine learning). The specific architecture is usually dependent on the use case, type of data, performance, and scaling needs. We will review the Lakehouse architecture, which arguably simplifies the architecture of data pipelines, mainly by reducing the need to use Warehouses to organize and access structured data. We explore implementing this architecture in a supply chain case study from Trase.earth (https://trase.earth), a project from Stockholm Environmental Institute and Global Canopy which is a leading supply chain data transparency initiative for mapping agricultural commodities driving deforestation. The heterogeneous nature of supply chain data and the challenges of managing and improving Trase's data pipeline without disrupting large portions of their codebase and workflow makes it a good case for examining the possibilities and limitations of a Lakehouse approach, which we test through the development of several proofs of concepts using a Databricks Data Lakehouse. To get a better grasp of the work done by Trase, we also examine a sample dataset related to beef animal movements in Brazil, using a Neo4j graph database.**

***Index Terms*—data lakehouse, data pipelines, data engineering, graph data science, supply chains, deforestation**

I. Introduction

One of the main responsibilities of data engineering is building and maintaining data pipelines [1] [2] in a way that enables the work of data scientists. Data management, especially around preparation and cleansing within ETL tasks, usually takes the biggest percentage of time from data scientists [3]. Working with heterogeneous data sources such as supply chain information (trading data, tax information, business registrations, sanitary controls, and spatial information, among others) can make managing ETL and keeping the data and models updated especially burdensome. For this project, we reviewed Trase's existing efforts for managing their data pipelines, as well as a dataset of animal transportation in Brazil.

As a guiding inquiry for this report, we explored to which extent a Lakehouse architecture could allow for the improvement of managing data pipelines in Trase. We also consider challenges and opportunities for incremental adjustments of Trase's existing codebase and workflow, in case they were to adopt such an architecture.

Alongside the technical considerations for data pipeline management, we reviewed basic literature related to Trase's work, including some of their methods for analyzing supply chain data and its link to deforestation [4] [5] [6] and some of their mapped commodities, including palm oil in Indonesia [7] and beef production in Brazil [8] [9]. We also reviewed conceptual frameworks for data characterization in the Industrial Ecology domain [10] [11]. Regarding Trase's current data pipeline workflow, we took note of their main concerns and ideas for improvement. We made a small exploration of possible frameworks, tools, and methods that could be useful, and finally decided to work around a Lakehouse architecture [12], implemented through a Databricks Data Lakehouse [13] [15]. We also reviewed two million records of sanitary reports of live beef transportation [16] between farms and slaughterhouses in the state of Pará in northern Brazil, using a Neo4j graph database.

In this report, we will focus on the exploration of the Lakehouse architecture and its relevance within Trase's case study: what challenges it responds to, what concepts it builds on, make several proofs of concept implementations with Trase's data and processes (mainly around ETL), discuss and evaluate results, limitations, and paths for future work.

II. Architectures for Data Pipeline Management

Data pipelines usually refer to how a series of data processing steps are connected and managed [17]. With the availability of ever-increasing data and reduced costs of storage and computing resources, several technologies and architectures have evolved in the past decade. One of such technology components, enabled by the proliferation of Big Data file systems in the 2010's such as Hadoop's HDFS and low-cost object stores such as S3, are Data Lakes. A Data Lake is a highly scalable centralized repository where unstructured and structured data is stored, enabling the import of big amounts of data coming in batch or real-time, collected from multiple sources, and stored in any format, without the time-consuming need for defining data structures, schemas and transformations [18]. A challenge in Data Lake architectures is that raw data

may be continuously dumped with no oversight of its contents while it's expected to be used in the future, with little or no metadata.

A. Challenges managing data from a Data Lake

For better management of the data in Data Lakes, it is common to apply a series of ETL processes that ensure structure and quality constraints, loading this transformed data into Data Warehouses. Data Warehouses emerged in the 1980's as central repositories of structured readily available for data analysis and reporting, enabling business intelligence (BI) and feeding business units' operational systems. While virtually all Fortune 500 enterprises connect their Data Lakes to Data Warehouses [12], this comes at the expense of a more complicated architecture, duplication of the data, use of proprietary formats (with vendor lock-in), additional ETL processes, as well as working with out-of-date data. Also, as advanced analytics and Machine Learning processes usually need to process the raw data, this can require additional ETL operations to connect the Warehouse back to the Lake, from where the data is processed again.

Optionally, connecting the objects in a Data Lake directly to the required applications (e.g. BI, Data Science), for instance relying on semistructured data objects such as CSV files, makes it hard to manage metadata consistently, guarantee ACID properties when managing data, having version control and rollbacks, efficient I/O, schema evolution, managing data governance, among others.

B. Data Lakehouse and Delta Tables

The Data Lakehouse architecture has emerged in the past two years, enabling efficient and secure machine learning and BI directly on the data stored in a Data Lake [19], without the need of going through a Data Warehouse and solving several of the problems mentioned in the previous section. To understand the relevance of a Lakehouse architecture, it's useful to review the relationship between Data Lakes and Warehouses. As mentioned, Data Lakes are highly scalable storage repositories that can hold large volumes of any mix of structured, semi-structured, and unstructured data. The data itself is in its native format until it is required for use [19]. Warehouses, on the other hand, process and transform data for advanced querying and analytics in a more structured database environment [19]. Data Lakehouses combine the functionalities of both [12], as seen in Figure 1, while simplifying their multi-tier architecture.

Lakehouses add traditional warehousing capabilities to existing Data Lakes, such as ACID transactions, data security, low-cost updates and deletes, and optimized performance for SQL and BI.

A prominent implementation of the Lakehouse architecture are Delta Lakes, which are based on an open-source storage layer on top of an existing Data Lake while guaranteeing ACID compliance. The core element of this storage layer are Delta Tables, which implement transaction logs in a fashion similar to .git folders of Git implementations, and which can be stored in the cloud object stores themselves. By storing a transaction log of every processing action, Delta Tables has versioning and time-travel integrated into them. They also support faster metadata searches and schema evolution (continue working with existing files without rewriting them if a table's schema changes) [13].

The Data Lakehouse model was introduced by Databricks and has been open-sourced during development [14]. Reportedly, more than half of Databricks customers have been able to simplify their overall data architectures in the last couple of years by replacing their previous Data Lake, Data Warehouse, and storage systems with a Data Lakehouse that provides all of these features [12]. Additionally, Databricks Data Lakehouse can currently run on any of the three major cloud providers (AWS, Azure, and Google Cloud Platform).

C. Streaming applications and graph data science

While streaming data sources usually require different sets of technologies, Delta Tables provide both batch and streaming modes, making it easier to use them without requiring additional architecture considerations. Regarding the compatibility between Databricks Data Lakehouse with NoSQL databases such as Neo4J (graphs), MongoDB (documents), Cassandra (wide-column), Redis (key-value), among many others, there are connectors that enable their integration. For Neo4j in particular, Databricks can connect through the bolt binary protocol to transfer data, as well as send Neo4j Cypher queries and load the results back to a Spark RDD, a Dataframe, or a Graphframe [20]. A common use case is to use Databricks to ingest data in streaming mode, send new records to feed Neo4j - or optionally indicate Neo4j to collect them from a file path [21]. Another common application is to use Neo4j graph data science algorithms [22] and load the results back to Delta tables for further processing and aggregations. For common graph algorithms such as BFS, LPA, shortest path, and PageRank, they can also be run directly in Databricks using Graphrame, without going through Neo4j, while leveraging Databricks autoscaling clusters and its Spark engine.

D. Sharing access to data

Sharing a correct version or subset of live data and managing fine-grained access permissions to it, usually requires proprietary solutions and/or deploying a computing layer to manage this process. Common solutions include sharing copies of the subsets of data, or building custom-made integrations so external readers can connect directly to it from their applications. All these solutions usually add architectural complexity and costs. As Delta Tables organizes data efficiently in parquet-like files with custom optimizations, and the transaction log keeps statistics and metadata useful for managing the table, it lends itself to making it easier to manage access to it without requiring a computing layer in the middle.

Delta Sharing [23] is an open standard for managing Delta Table shares securely. Through this protocol, the tables can be accessed directly from their source (e.g. S3, ADLS, or GCS) regardless of the computing platform used for accessing them, through a REST protocol. This protocol has several

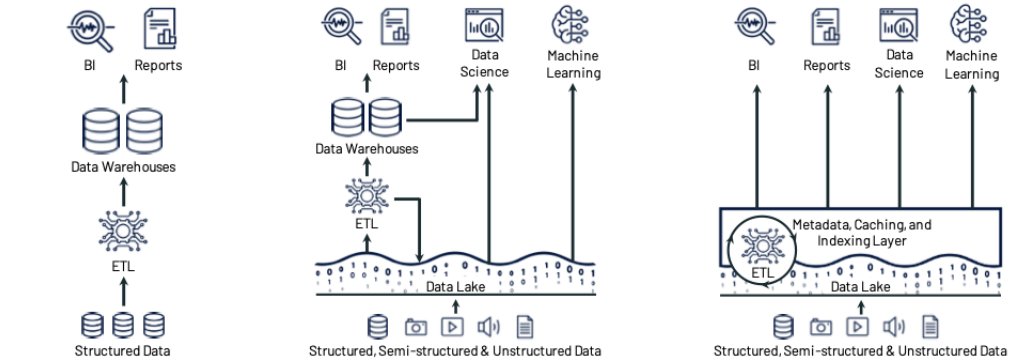

Fig. 1. Evolution of Data Platforms, from Warehouse (a) to Lakehouse (c). Taken from [12, Fig. 1]

implementations for accessing shares directly from Pandas, Spark, PowerBI, Tableau, BigQuery, among others. Depending on where results are read, they might need to fit into memory when loaded (e.g. Pandas Dataframes or PowerBI).

For filtering the results before loading them, a table can be queried to return the information of specific partitions, or use a limited set of SQL queries, for the moment based on equality or order (e.g. "$country = SE$", "$code > 15600$"). It is also possible to load only the table metadata and commit history, and to load a specific version of the table. Furthermore, if the Data is shared using Databricks Unity Catalog (which allows for managing fine-grained access permissions [24]), the table shares can already have restricted access based on partitions, versioning, column or row conditions (e.g. restrict access to columns holding personal or proprietary information, or to records that have some filtering condition). Different sets of permissions can be specified on different shares pointing to the same table, and permissions can be set using standard SQL. In this way, Delta Tables used in production environments can be read from different recipients with different levels of access, all with the most up-to-date information and without requiring copies of the data or an intermediary compute layer.

When using Unity Catalog, it is also possible to audit and monitor who has read the data and when. This can be useful to analyze usage patterns of the data shared, and eventually determine recipient access quotas. As Unity Catalog also includes the option to identify data lineage between tables up to the column level (which allows seeing data dependencies upstream and downstream), this can be used to have a more complete audit of data flows and access of information, which in some cases might be needed to comply to regulations such as GDPR or industry-specific (e.g. government or health sector).

III. Implementation

We implemented several proof of concepts of the architecture components mentioned in the previous section, using as a case study Trase's data. We mention some of this work, which can also be reviewed in the code submission complementary to the report, and better detail of the procedures can be found in the related documentation [25].

A. Delta tables

One of the reasons to prototype the use of Delta Tables and Data Lakehouse was to test a solution to create a data lineage for the many small and scattered CSV files from different processing scripts. The use of CSVs can be problematic since there might be multiple options and issues for parsing (escape characters, conflicting field delimiters, multiline fields), no native compression, and no specified schema, among others. With Delta Tables, the process of handling schemas, metadata, tracking column lineage, and updates are easier, and sharing commercially ready tables can be made simple through Delta Sharing. Any update to a Delta Table (be it new records to append, replace, merge, or column additions) can be easily managed by re-running a pipeline. Once loaded to their corresponding Delta Tables, CSVs could be saved in cold storage until needed again, thus reducing costs. To test this, we loaded a commonly used 10GB CSV file stored in AWS S3 into a Delta Table, after which it's easy to make transformations and queries to it through the Databricks platform.

B. Managing pipelines (Delta Live Tables)

Delta Live Tables (DLT) is a managed workflow for loading Delta Tables in Databricks, with easy management of data dependencies as well as expectations (constraints on the quality of the dataset) and management of non-compliant records (quarantine or delete them, or halt the process). For this project, we implemented with Databricks Delta Live Tables a couple of pipelines based on two existing Python preprocessing scripts of the Ecuador shrimp commodity, which generated an interactive visual data lineage that allows reviewing the history of the pipelines executions, quickly viewing the schemas of the tables, the records processed, among others.

Trase relies on AWS-helper functions to create and manage metadata, and load, cache, and save data from AWS in their

preprocessing scripts. For implementing Trase's pipelines in Databricks, an approach we reviewed is to make changes to the helpers' functions such that instead of loading a CSV file from an S3 bucket, it loads from the corresponding Delta Table, with the option of accessing also its schema description, comments, version history, and additional relevant metadata. Some options to manage this, as well as potential limitations, are discussed in the results section.

C. Beef animal movements with graphs in Neo4j

We loaded into a Neo4j database a JSON dataset representing 2 million GTA records (sanitary reports of animal transportation - 'Guia de Trânsito Animal' from its acronym in Portuguese [16]) for the Brazilian state of Pará between 2013 and 2020, which were previously produced by the Trase team by scraping publicly available data. For this we used a Neo4j's Cypher APOC procedure [26], iterating through the provided input files on S3, each one of approximately 200 - 500 MB in size, ingested at an average of 100MB per minute using a single instance of an 8 CPU, 16GB RAM virtual machine. For a more complete use case, the procedure could be made to read from a streaming source, and the scraping could be initiated or scheduled through Databricks Jobs. The graph was modeled with 12 node types, from which farms (197,123 nodes) and slaughterhouses (5,262 nodes) are the most relevant ones. Also, there are 19 relation link types between nodes, including amounts and descriptions of animals sent between farms and slaughterhouses, amounting to 17.1 million relationships between the nodes. Additional to basic statistics such as identifying the most relevant flows between farms and slaughterhouses (ordered by the number of sent animals), several subgraphs of possible indirect suppliers for specific slaughterhouses were queried.

D. Unity Catalog and Delta Sharing

For the implementation of the Unity Catalog and Delta Sharing, we created a new metastore (the top-level container of objects) on an existing Databricks account. A metastore contains all of the metadata that defines data objects in the lakehouse [27], and is linked to a cloud storage location (in our case, a specific S3 bucket). One or more external locations can be created to allow for the management of different data sources.

As a proof of concept, we created notebooks to load several CSVs from the S3 source with the raw data, and saved them as delta tables into catalogs, creating a catalog per country, and a database per commodity, each with its corresponding tables. After the delta tables are created they can be referenced based on the catalog, database, and table name. For example: $display(spark.table(" < catalog > . < db > . < table\_name > "))$. When only read access is required, for example for granting access to external users, a Delta Share can be configured. When creating it, 'recipients' and 'data shares' have to be created. Usually, a recipient would be the receiving organization or group of persons who will be accessing certain tables with corresponding permissions. When a recipient is created, a link is created to be shared with them, with a one-time access to download a credential file (config.share) which includes an access token string with an endpoint URL and an expiration date. This configuration file is needed to access the data from any of the platforms used for accessing it (as a Pandas or PySpark Dataframe, PowerBI, Tableau, etc), except for accessing through a Databricks account, which doesn't require a credential file. When creating a 'data share', different catalogs, databases, and tables can be specified, as well as different recipients. Also, only specific partitions of a table can be shared if needed. As a test, a Delta Share was created sharing some tables, which then were loaded by the recipient in a Pandas Dataframe. Also, some useful methods were tested for the recipient to list the tables with their corresponding share names and databases they make part of. For a specific table, its metadata and version history could also be queried.

E. Documentation

The work that has been done related to Delta Live Tables, Unity Catalog, and properly mounting S3 buckets to Databricks, among others, has been documented for Trase, as well as made publicly available [25] (without sharing potentially private data from Trase). Some of the documentation include:

- A general guide on securely using S3 buckets for access in Databricks.
- Information on Databricks Unity Catalog along with the proper configuration of related S3 buckets and user permissions.
- Documentation on Delta Lake, Delta Live Tables, and setting up the pipeline.
- Sharing resulting tables through ODBC/JDBC connectors to external users running Excel, PowerBI, or programming IDE:s for example. Note that this is different from the usage of Delta Shares, which is showcased in the code submission.

IV. Results and Discussion

The adoption of a Lakehouse architecture for Trase as implemented in Databricks has several advantages and challenges. While it presents opportunities to manage and process data in a simple, robust and scalable architecture cost-effectively, the fact that it runs by default on a Spark engine poses some suitability and compatibility issues with Trase's existing codebase, the team's current expertise, and the small size and type of data and computations typically done in Trase. We will summarize the main potentials, limitations, possible ways to resolve them, and areas for future exploration.

A. Potentials for Trase

As mentioned before, working with a Lakehouse can make it easier to work with unstructured, semistructured, and structured data, both in batch and streaming mode, without the need for a three-tiered Warehouse architecture. As Databricks runs on a Spark engine with easy-to-configure scaling, ingesting

and processing bigger datasets or doing large computations can be easily managed. All existing data on the lakehouse can be easily accessed and shared, queried with SQL, searched from a general search box, and dashboards for basic analysis and BI can be easily created. Data pipelines can be defined in a declarative language, and then be run and managed, seeing when it was executed, by whom, and live details of data health based on the defined constraints and quality checks.

B. Related costs

Databricks uses a pay-as-you-go model based on computing usage [28], which can easily be vertically or horizontally scaled. An all-purpose computing hour through AWS currently costs 0.4 USD per i3.xlarge node (which has 4 vCPU and 30GB Ram). If initially Databricks would only be used for managing the data (for example, transforming CSVs into Delta tables and sharing them through Unity Catalog) this can easily be run on a i3.large single node (2 vCPU, 15GB ram), costing 0.2 USD per hour, plus 0.172 USD for the AWS EC2 costs. Supposing this type of activity is run on average a couple of hours per working day, this would add roughly to 15 USD per month (40 hours at 0.372 USD each). If Databricks is used also as a SQL Warehouse to run more complex SQL queries, quickly search within all existing data, run BI analysis through them, use dashboards, or connect to other systems through the Databricks JDBC/ODBC driver, it is recommended to use the Databricks SQL Serverless compute. While this service costs more (currently 1.82 USD per hour for 'small' loads, such as in Trase's case), starting an instance usually takes just a couple of seconds, which automatically turns off after 10 minutes idle (though can be configured for stopping after 1 minute of inactivity to save costs). Supposing this service is used 80 hours per month (4 hours per day during working days), this would amount to 145.6 USD per month. If the use of the SQL Warehouse partially replaces other platforms used by Trase such as Splitgraph, Metabase, or PostgreSQL, it could save costs overall.

Databricks is commonly used as a development environment, be it with Databricks notebooks, or connecting from a local IDE (e.g. VS Code, PyCharm, RStudio) or through pre-configured remote environments (e.g. Github Codespaces, JupyterHub). In this use case, supposing 5 to 10 remote users connected concurrently to a single node on average 10 hours per day all working days using a 4vCPU - 16GB RAM node, would cost 126 USD per month (per hour cost of 0.4125 USD for Databricks and 0.222 USD for AWS EC2, for 200 hours). Additional clusters with more capacity (more nodes, each one with more CPUs and RAM) can be configured for processing bigger data loads or computing complex models.

C. General limitations

Databricks Notebooks run on a Spark engine within a runtime called Databricks Runtime (DBR) which though can run Python code through PySpark, has several of the most commonly used Python packages and can install additional ones as needed, they can have compatibility issues with Trase's codebase [29]. Using Spark mainly for small datasets that easily fit in a single node memory can seem inadequate, and even show lower performance (a computation that can take less than a second in a normal Python environment can take a couple of seconds when going through a Spark job). Also, for the moment, notebooks can run R code (through SparkR or Sparklyr) or Scala code only if running within a single access cluster: if users plan to run R or Scala code on their notebooks concurrently, each of them would have to start an independent cluster.

Working with interactive notebooks that facilitate concurrent edition and automatic history versioning, while can make it easier to collaborate and document (using markdown), also makes it harder to implement good development practices (modular code, unit tests, linting, CI/CD). While there are workarounds for integrating these practices into notebooks, they would require adjustments to Trase's team's workflow. Alternatively, as mentioned earlier, IDEs can be connected to Databricks, and Databricks can be connected to repositories so Trase's codebase can be accessed.

D. Limitations regarding pipeline management

Currently, Trase preprocessing scripts usually combine several CSVs, apply transformations, and write the results into final CSVs. While managing data lineage is straightforward when using Delta Live Tables syntax or Delta Tables through Databricks notebooks, doing this could require rewriting large portions of Trase's existing codebase. One of the reasons is that Databricks relies on Spark Dataframes, or optionally Pandas on Pyspark Dataframes to be able to read or write to Delta Tables. A way to circumvent this is to adjust Trase's existing AWS helper functions to convert from/to normal Pandas on Python Dataframes for reading from / writing to Delta Tables, though with this losing Spark's innate ability to run in distributed computing clusters, and requiring datasets to fit into a single node memory for them to be transformed. This could be however a solution for maintaining backward compatibility with Trase's codebase. This initial solution can also start including additional metadata (automatically as well as user-generated) directly into the tables with information that allows for easier tracking and management of table and schema comments, versioning, explicitly specifying lineage, among others.

New implementations and updates of Trase's commodities could start using Delta Tables, Delta Live Tables and Spark features natively, including its possibilities to manage pipelines more robustly, implementing a medallion architecture [30], as well as integrating tools that can further facilitate managing data transformations (be it with tools from the Spark ecosystem, or with third-party services). For the possibilities, differences, and syntax for using Delta Tables and Delta Live Tables for pipeline management, refer to the related documentation and code submission [25].

E. Future work

Currently, we have made several proofs of concept implementations using example datasets and scripts from Trase. Building a more complete prototype of the features described in this report would give a better picture of the suitability for integrating a Lakehouse architecture and of using Databricks services. Another option is to rely on certain key aspects of the Lakehouse architecture without using Databricks, by using the delta.io open source implementations [14], and reading and writing to Delta Tables directly from Pandas Dataframes, without requiring a Spark engine.

While an initial ambition was to run end-to-end an existing pipeline for a commodity (initially Ecuador shrimp), this is yet to be done, and there are different approaches on how to do it as briefly mentioned in the last section. Also, regarding the integration of bigger sources of data, including streaming up-to-date sources, as well as using graph data science, there are many possibilities to continue exploring as mentioned in the implementation section.

Overall, we find clear motives for further exploring a Lakehouse architecture within Trase in an incremental way with potential short and medium-term added values that don't require radical adjustments in the existing codebase or workflows. These initial potentials would mainly be around improvements in data management and governance that traditionally come from implementing Data Warehouses, though without the architectural complexity, costs, and vendor lock-in that usually comes from them. Using other features of the Databricks Lakehouse such as declarative pipeline management with Delta Live Tables, and making better use of the Spark ecosystem for ETL, still require further work. Also, conducting further interviews and surveys with Trase's team, and doing a detailed cost analysis can provide a better picture for further analysis and for deciding if carrying on with an implementation.

REFERENCES

[1] "Data engineering - Wikipedia." Wikipedia.org. https://en.wikipedia.org/wiki/Data_engineering (Accessed: Jan. 05, 2023).

[2] "Data engineering: A quick and simple definition." oreilly.com. https://www.oreilly.com/content/data-engineering-a-quick-and-simple-definition/ (Accessed: Jan. 05, 2023).

[3] "State of Data Science 2020." anaconda.com. https://www.anaconda.com/state-of-data-science-2020 (Accessed: Jan. 05, 2023).

[4] 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, 2015, doi: 10.1016/j.ecolecon.2015.02.003.

[5] Trimmer, C., "Supply chain mapping in Trase - Summary of data and methods," Trase. https://policycommons.net/artifacts/2485182/supply-chain-mapping-in-trase/3507610/. 2018. CID: 20.500.12592/jtsxgq.

[6] Trase, "How Trase assesses 'commodity deforestation' and 'commodity deforestation risk'." Trase, 2020 . http://resources.trase.earth/documents/data_methods/Trase_deforestation_risk_method_final%20Sept%202020.pdf. (Accessed: Jan. 06, 2023).

[7] Trase, "SEI-PCS Indonesia palm oil v1.2 supply chain map: Data sources and methods." Trase, 2022. https://resources.trase.earth/documents/data_methods/SEI_PCS_Indonesia_palm_1.2._EN.pdf (Accessed: Jan. 06, 2023).

[8] 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, 2020.

[9] Proforest, "Socio-environmental monitoring of the cattle sector in Brazil." Proforest, 2017. https://www.proforest.net/fileadmin/uploads/proforest/Documents/Publications/bn09_eng_final_web.pdf. (Accessed: Jan. 05, 2023)

[10] O. Schwab, O. Zoboli, and H. Rechberger, "A Data Characterization Framework for Material Flow Analysis," Journal of Industrial Ecology, vol. 21, no. 1, pp. 16-25, 2016, doi: 10.1111/jiec.12399.

[11] Hertwich, E. et al. (2018) "Nullius in verba1: Advancing Data Transparency in industrial ecology," Journal of Industrial Ecology, 22(1), pp. 6–17. https://doi.org/10.1111/jiec.12738.

[12] Databricks, "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics." Databricks, 2021. https://www.databricks.com/wp-content/uploads/2020/12/cidr_lakehouse.pdf. (Accessed: Jan. 06, 2023).

[13] Databricks, "Delta Lake - The Definitive Guide: Modern Data Lakehouse Architectures with Delta Lake." Databricks, 2021. https://www.databricks.com/p/ebook/delta-lake-the-definitive-guide-by-oreilly. (Accessed: Jan. 06, 2023).

[14] "Delta Lake: Home." delta.io. https://delta.io/ (Accessed: Jan. 05, 2023).

[15] "Data Lakehouse Platform by Databricks" databricks.com. https://www.databricks.com/product/data-lakehouse (Accessed: Jan. 05, 2023).

[16] "GTA." gov.br. https://www.gov.br/agricultura/pt-br/assuntos/sanidade-animal-e-vegetal/saude-animal/transito-animal/arquivos-transito-internacional/ModelodeGTA.pdf. (Accessed: Jan. 06, 2023).

[17] A. Raj, J. Bosch, H. H. Olsson, and T. J. Wang, "Modelling data pipelines," 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2020.

[18] "What is a data lake?." aws.amazon.com. https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/ (Accessed: Jan. 05, 2023).

[19] "What Is a Lakehouse? - The Databricks Blog." databricks.com. https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html (Accessed: Jan. 05, 2023).

[20] "Neo4j." docs.databricks.com. https://docs.databricks.com/external-data/neo4j.html (Accessed: Jan. 05, 2023).

[21] "049 Connected Data Lakehouse Neo4j and Databricks Reference Data Architecture - NODES2022." YouTube.com. https://www.youtube.com/watch?v=22qstAxlWMI (Accessed: Jan. 05, 2023).

[22] "Graph algorithms - Neo4j Graph Data Science." neo4j.com. https://neo4j.com/docs/graph-data-science/current/algorithms/ (Accessed: Jan. 05, 2023).

[23] "Delta Sharing — Delta Lake." delta.io. https://delta.io/sharing/ (Accessed: Jan. 05, 2023).

[24] "What is Unity Catalog? — Databricks on AWS." docs.databricks.com. https://docs.databricks.com/data-governance/unity-catalog/index.html (Accessed: Jan. 05, 2023).

[25] "Code and Documentation for Project Course(1DL507)." Github.com. https://github.com/nmartinbekier/project_course_1DL507 (Accessed: Jan. 05, 2023).

[26] "Load JSON - APOC Extended Documentation." neo4j.com. https://neo4j.com/labs/apoc/4.1/import/load-json/ (Accessed: Jan. 05, 2023).

[27] "Data objects in the Databricks Lakehouse: What is a Metastore? — Databricks on AWS." docs.databricks.com. https://docs.databricks.com/lakehouse/data-objects.html#what-is-a-metastore (Accessed: Jan. 05, 2023).

[28] "AWS Pricing - Databricks." databricks.com. https://www.databricks.com/product/aws-pricing (Accessed: Jan. 05, 2023).

[29] "Databricks Runtime 11.3 LTS: System environment." docs.databricks.com. https://docs.databricks.com/release-notes/runtime/11.3.html#system-environment (Accessed: Jan. 05, 2023).

[30] "Medallion Architecture – Databricks." databricks.com. https://www.databricks.com/glossary/medallion-architecture (Accessed: Jan. 05, 2023).

V. Statement of Contribution

For the creation of this report, we divided responsibilities in the following way:

- For document structure and main ideas within each section we developed them jointly.
- For introduction, Nicolás took the main lead.
- Within section II: “Architectures for Data Pipeline Management”, Nicolás took the main lead, except for the “Data Lakehouse” subsection.
- Within section III: “Implementation”, Oskar took the main lead, including the coordination of the code submission, the development of the documentation, the Delta tables, and the Managing pipelines subsection. Nicolás took the lead on “Beef animal movements with graphs in Neo4j” and “Unity Catalog and Delta Sharing”
- For section IV: “Results and Discussion”, Nicolás took the main lead, except for most of the considerations regarding Delta tables and Delta live tables.