Politecnico di Torino

Politecnico di Torino

Master's Degree in Data Science and Engineering

A.y. 2024/2025

October 2025

Adaptation and Implementation of an automated Corporate ETL Framework on Microsoft Fabric

A Technical Approach to Integrating Data Workflows into a Modern Cloud Platform

Academic Supervisor:
Prof. Paolo Garza

Company Supervisor:
Dr. Luca Bregata

Candidate:
Guillermo Josè Gallucci

Abstract

The exponential growth of data generated by computational systems, users, and devices has introduced significant challenges for real-time processing, resource optimization, analysis, and the provision of training material for AI systems. On-premise infrastructures, while capable of high computational power, are constrained by hardware and tool configuration, limiting scalability and collaboration. To address these limitations, the research focuses on Microsoft Fabric, a SaaS cloud-based platform integrating data processing and real-time analytics.

The research, carried out in partnership with Mediamente Consulting, aims to adapt and implement the corporate data integration automated framework within Microsoft Fabric, and to evaluate its performance relative to the on-premise version. Furthermore, the study seeks to identify a best-practice architecture among the various solutions provided by Fabric.

Data from CSV and Excel sources is ingested into Microsoft Fabric's centralized storage, OneLake. The pipeline loads the data in Delta Tables within a Data Lake architecture. Downstream processing layers are designed for incremental loads, propagating only new or updated records, which are subsequently evaluated to verify compliance with data quality and referential integrity constraints. The final stage has three different implementations to evaluate performance: Spark, SQL-based, and Dataflows.

It resulted in outstanding performance from the Spark- and SQL-based implementations compared to the Dataflows. Nevertheless, the overall performance in Fabric was worse than the metrics achieved in its on-premise version, except when dealing with large datasets, where Fabric outperformed. These results are due to Fabric's optimized cloud engines. The SQL engine benefits from query optimization, Spark excels with distributed computation for very large datasets, and Dataflows introduces overhead and abstraction due to its low-code abstraction, resulting in lower performance.

Table of Contents

List of Figures

Chapter 1

Introduction

In today's increasingly digital and automated world, data represents a vital resource, experiencing exponential growth on a yearly basis [1] as Figure 1.1 displays. Often referred to as "the new oil", data must be accurately captured, meticulously cleaned, and efficiently processed to ensure high quality before being delivered to end users. This requires an informed, precise, and timely decision-making processes.

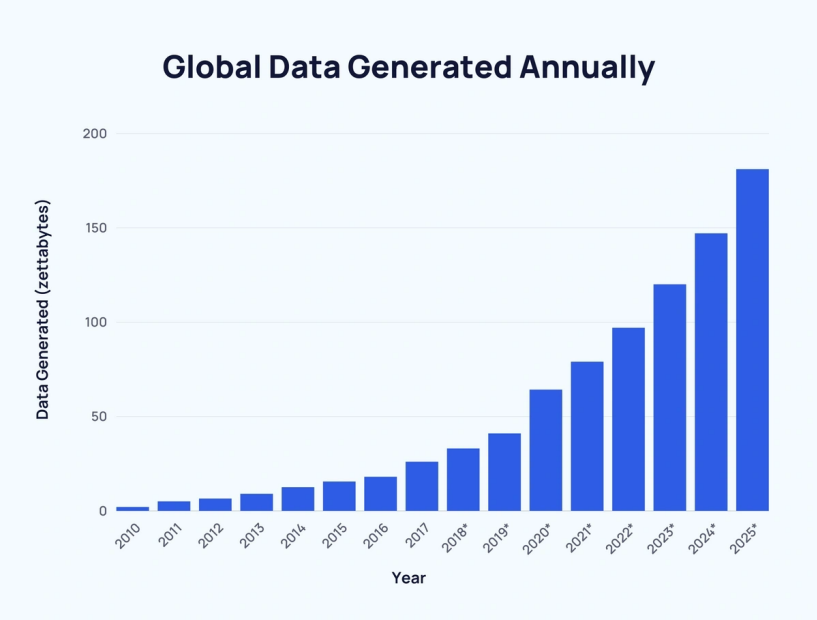


Figure 1.1: Global Data Generated Annually [1]

Moreover, data are not only essential for managerial decisions, but also serve as the fundamental input for Artificial Intelligence and Machine and Deep Learning systems. These systems rely on datasets -commonly named as training sets- to learn patterns and behaviors; subsequently, models are validated using separate validation

sets and finally tested on tests sets to evaluate the autonomous decisions they make.

Due to the sheer volume, velocity, and variety of modern data, optimizing the entire data pipeline has become a critical necessity. Efficiency and scalability are no longer optional: they are foundational requirements to ensure that data can be ingested, transformed, and delivered in a timely manner. Real-time or near-real-time analytics capabilities are essential, as delayed insights can result in missed opportunities, suboptimal decisions, and system failures. The value of data does not lie in its accumulation, but rather in its timely transformation into actionable information.

A data pipeline that cannot support rapid processing and intelligent distribution across business units limits the organization's ability to respond to market dynamics, customer behavior, and internal operational needs. Therefore, the implementation of a robust, automated and cloud-native ETL framework is crucial not only to maintain data quality and integrity but also to ensure that data can effectively manage strategic planning and real-time decision-making processes across the company.

1.1 Goal

The objective of this thesis is to design and implement an automated process in Microsoft Fabric, a cloud-based environment capable of efficiently managing the extraction of operational data from one or more sources, applying the necessary transformation procedures to meet predefined quality standards, and ultimately loading the refined data into a corporate Data Warehouse, then, evaluate if it can be a possible competitor to established on-premise tools. This structured pipeline ensures that data are prepared and made available for analysis and informed decision-making across the organization.

The project has been developed in collaboration with Mediamente Consulting, a technology and digital consulting firm that provides tailored solutions to support the innovation and digital transformation of client organizations. The company operates through several and specialized business units, the one that supervised the development of this project is the Data Integration unit, which oversees data ingestion, transformation, and synchronization processes, as well as the construction of data warehouses, data marts and data lakes, in addition to enabling data-driven operations through robust and scalable ETL architectures.

Several years ago, the Data Integration business unit developed a corporate framework to manage automated and efficient ETL processes. This framework was originally implemented using Oracle Data Integrator (ODI), an on-premise software product that enables the integration of data from various sources, including Oracle databases.

However, over time, the landscape of enterprise data management has evolved significantly. Clients now require cloud-based solutions that deliver comparable or superior performance to traditional on-premise systems by reducing infrastructure costs and hardware maintenance. Furthermore, Cloud platforms allow business to allocate resources based on workload demands, which is very valuable in environments where data volumes and process needs to fluctuate rapidly. In addition, cloud solutions enable real-time collaboration among team members regardless of their department or geographic location, fostering greater agility and faster decision-making.

Recent studies indicate that approximately 94% of companies worldwide have adopted some form of cloud computing, and as of 2023, corporate data account for 60% of all data stored in the cloud [2]. As Figure 1.2 illustrates, this percentage has been consistently increasing and is expected to continue its upward trend in the upcoming years.

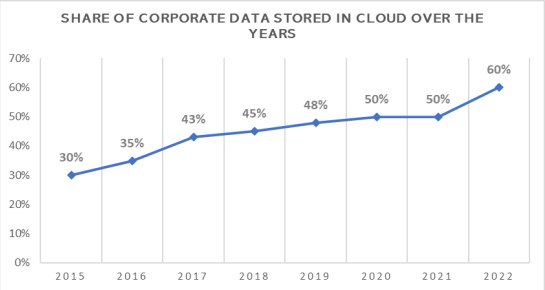


Figure 1.2: Trend of Corporate Data Stored in the Cloud Worldwide over the Years [2]

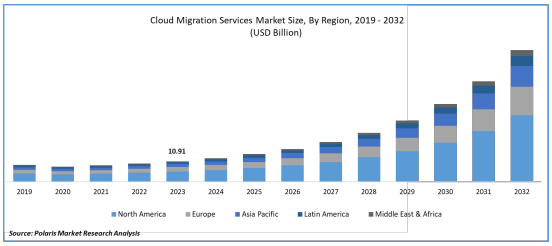


Figure 1.3: Cloud Migration Services Market Size, by region from 2019 to 2032 (estimated) [3]

Figure 1.3 shows the growth of the market for cloud migration services over the past several years and it is projected to continue expanding throughout the next decade, with the United States leading this trend and Europe closely following [3].

It is within this context that Mediamente Consulting is exploring modern alternatives to implement its automated ETL framework on cloud-based platforms, aiming to maintain high performance and data quality standards while leveraging the benefits of a modern, integrated cloud system.

The answer to this requirements and needs lie in Microsoft Fabric, that offers an end-to-end solution for data movement, transformation and analysis. Microsoft Fabric is built on top of Azure, leveraging its robust cloud infrastructure to deliver scalable computing, high availability, and tight integration with other Microsoft services such as Azure Data Factory and Power BI. This deep integration provides a cohesive environment for developing data workflows.

1.2 Thesis structure

The structure of the thesis is the following:

- Chapter 1 - Introduction: Explanation of the project's objectives and underlying motivation, the implementation of an automated ETL framework within a cloud environment.

- Chapter 2 - Data Warehouse: Definition of a Data Warehouse, its main purpose and key components. The chapter includes a comparison between the Kimball and Inmon approaches in building a Data Warehouse, highlighting the advantages and disadvantages of each, additionally, it outlines the differences between OLTP and OLAP systems, and presents the Star and Snowflake Schema data models. The chapter concludes with a brief description of Data Marts and the role of metadata in the development of the Data Warehouse.

- Chapter 3 - Data Lake: Definition Data Lakes, its core purposes, and the main advantages and disadvantages associated with its use. It also presents a comparison with the Data Warehouse architecture.

- Chapter 4 - Microsoft Fabric: Presentation of Microsoft Fabric, a cloud platform designed primarily for data extraction, transformation, and loading. It introduces the core terminology used within Fabric, provides an overview of OneLake as its centralized storage, and explores the main experiences offered by the platform, including Data Factory, Data Engineering, and Data Warehouse. The chapter also discusses Fabric's limitations and reviews both on-premise and cloud-based alternatives.

- Chapter 5 - Proposed Solution: Mediamente Consulting's ETL Framework: Definition of Mediamente Consulting's solution to the problem addressed in this thesis: a framework structured into three logical levels. Level 0 (L0) handles data extraction from sources and incremental loading to the next level. Level 1 (L1) performs Data Quality checks and Data Transformation operations. Finally, Level 2 (L2) is responsible for loading the processed data into a Data Warehouse. This chapter also explains the fundamental role of metadata that enables the automation of the ETL process.

- Chapter 6 - Implementation of the Proposed Solution within the Microsoft Fabric Environment: Detailed explanation about how the theoretical framework presented in the previous chapter is implemented on Microsoft Fabric through pipeline orchestration, each pipeline is in charge of a logical level (L0, L1). Specifically, Level 0 is realized by sequential execution of Spark Notebooks. For Level 1, three different implementation options are proposed: Spark, T-SQL, and Dataflows.

- Chapter 7 - Use Case: Mediamente Consulting ETL Framework: Example illustrating the framework functioning with a toy dataset, highlighting the key instructions to give a practical perspective on the implementation described in Chapter 6.
- Chapter 8 - Results: This chapter reports the metrics collected during the execution of the ETL pipeline. The pipeline was tested with input files of varying sizes—100, 1,000, 10,000, 100,000, and 1,000,000 rows. For each input size, five executions were performed to measure execution times. Based on these results, the mean, standard deviation, and 95% confidence interval (using the Student's t-distribution) were calculated. The evaluation covers L0 and the three different implementations of L1, concluding with a comparative analysis of the L1 alternatives.
- Chapter 9 - Discussion of results: Analysis and explanation of the results obtained.
- Chapter 10 - Conclusion: A final reflection is provided based on the results obtained, focusing on the maintainability and implementation aspects of the ETL framework. Additionally, a conclusive assessment is given to determine the most effective implementation among the proposed alternatives.

Chapter 2

Data Warehouse

A Data Warehouse is a centralized system specifically designed to collect, integrate, and store large volumes of data from heterogeneous sources, with the primary goal of supporting reporting, analysis, and decision-making processes. Unlike traditional operational systems, Data Warehouses are optimized for historical and aggregated analysis, enabling the frequent extraction of valuable business insights. These insights are typically derived from a small number of complex queries based on dynamic analytical needs. For instance, *"What was the most acquired pharmaceutical drug across Italian pharmacies before the COVID-19 outbreak, and which one was the most acquired after?"* Such a question requires access to integrated, time-variant data, covering multiple periods, and the ability to perform comparative analysis over aggregated purchase trends.

The main characteristics of a Data Warehouse, as defined by William H. Inmon, (considered the father of data warehousing) can be defined as follows [4] :

- Subject Orientation: Data is organized around key or business domains. For example, for a pharmaceutical company, the applications may include management of drug inventories, tracking of sales transactions, monitoring of distribution logistics and recording of prescriptions
- Integration: Data is collected from heterogeneous sources, often using different formats, naming conventions or standards. As the data is fed to the warehouse, it is converted, reformatted, encoded, transformed then unified into a consistent structure.
- Non-volatility: When data is loaded in the Data Warehouse, it is loaded in a static format called snapshot, when changes occur, a new snapshot record is written; data is not updated or deleted by end-users, it can only be accessed. This enables the preservation of historical data inside the Data Warehouse.
- Time-varying: Implies that data inside the Data Warehouse is accurate in a specific point in time, indicated by a timestamp or a transaction date. A time

horizon refers to the temporal scope represented within a given data environment, a Data Warehouse is capable of storing historical data over extended periods, often ranging from 5 to 10 years, while an operational system can retain data for a short period of time, usually between 60 and 90 days.

2.1 OLAP and OLTP

A Data Warehouse is usually supported by OLAP (Online Analytical Processing) technology, since it enables efficient exploration and analysis of large volumes of historical data. OLAP is specifically designed to support complex queries and multi-dimensional analysis, making it ideal for decision-making and business intelligence task, these systems are optimized for read-intensive operations and analytical performance, they allow users to perform operations such as drill-down (examining more details in data), roll-up (summarizing data), slice (filtering a specific dimension), and dice (applying multiple filters). To enable fast querying and aggregation, OLAP technologies typically use de-normalized data models like a Star Schema, where data is organized into fact and dimension tables.

The opposite of OLAP technologies, are OLTP (Online Transaction Processing) technologies, which are typically used in operational systems such as e-commerce platforms, banking applications, and enterprise resource planning (ERP) systems. OLTP systems are designed to handle day-to-day transactional operations, such as inserts, updates, and deletes that occur in operational databases. These systems ensure data consistency and integrity through normalized database structures.

Since the objective of this thesis is to build an ETL pipeline based on a de-normalized data model, the focus will be on OLAP technologies, as they support a Data Warehouse environment, particularly those involving data integration.

2.2 Structure of the Data Warehouse

The two Data Warehouse architectures that have endured over time and laid the foundation for modern warehousing systems are those proposed by Ralph Kimball and William H. Inmon.

2.2.1 William H. Inmon's Architecture

In Inmon's *Building the Data Warehouse* [4], data exists at multiple levels of granularity, each serving different analytical needs. These levels include:

- Older detailed data, typically stored on an alternate, bulk storage
- Current detailed data, maintained for immediate analytical access

- Lightly summarized data, often corresponding to the data mart level
- Highly summarized data, used for executive dashboards and high-level reporting, i.e., KPIs.

Data enters the Warehouse from operational systems through transformation processes, to ensure consistency, quality and standardization. Once inside the Data Warehouse, data transitions over time and through levels of aggregation: as it ages, it moves from current detail to historical detail; as it is summarized, it moves from detailed to lightly summarized, and eventually to highly summarized forms.

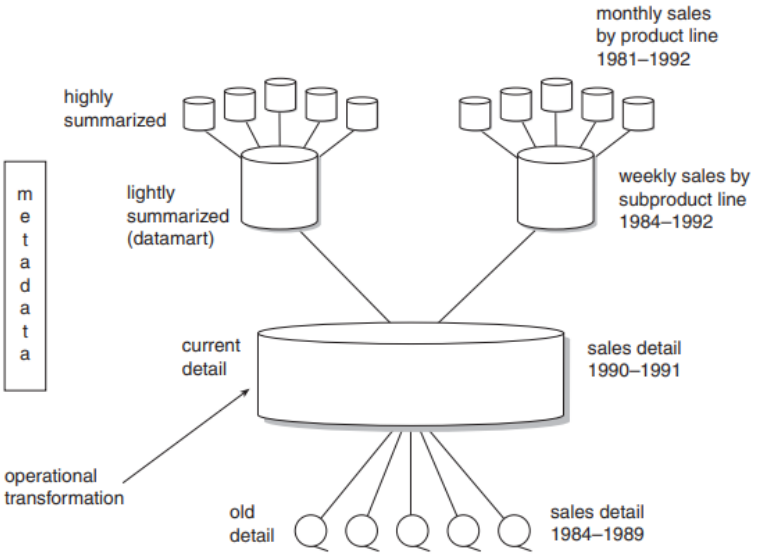


Figure 2.1: Structure of a Data Warehouse by William H. Inmon. [4]

Figure 2.1 illustrates Inmon's proposed architecture, highlighting the various levels of data granularity present within the Data Warehouse environments.

This Data Warehousing approach offers several advantages, particularly suited to large and complex enterprise environments. First and foremost, it serves as a centralized and integrated source of data for the entire organization, ensuring consistency and integrity across all departments. Its normalized structure minimizes data redundancy, reducing the risk of updates and simplifying data maintenance. Moreover, the logical model is closely aligned with real business entities and processes, making it easier for both technical and business stakeholders to understand and work with. The architecture is adaptable, allowing data warehouse to accommodate different business requirements and changes in source system. Finally, it provides a high-quality, consolidated data foundation.

Nevertheless, the Inmon approach also presents possible drawbacks. As the data model evolves and more tables are added, the overall architecture can become

increasingly complex and challenging to manage. Its implementation and maintenance demand specialized data modeling skills, which are often difficult to find and expensive. Additionally, the initial design and deployment phase tends to be time-consuming, delaying the time-to-value for the organization.

2.2.2 Ralph Kimball's Architecture

In *The Data Warehouse Toolkit* [5], Kimball presents a Data Warehousing architecture structured around distinct functional components, each responsible for a specific set of tasks within the overall data integration process. Unlike Inmon's architecture, which emphasizes hierarchical levels of data granularity,

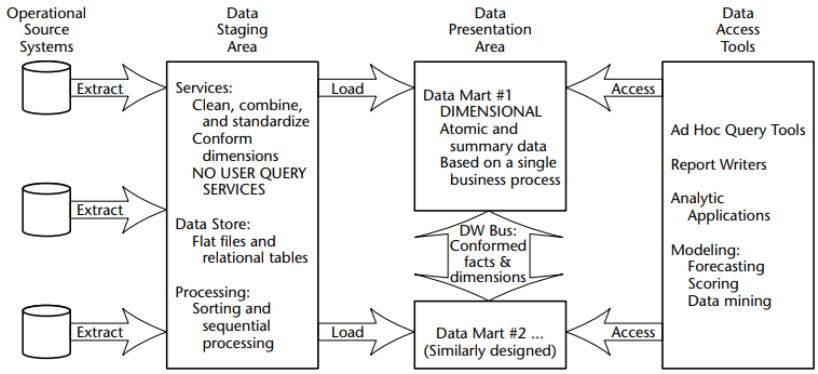


Figure 2.2: Structure of a Data Warehouse by Ralph Kimball. [5]

Figure 2.2 presents the Kimball architecture along with the components previously described.

Operational systems are responsible for capturing an organization's transactional data. These sources systems are external to the Data Warehouse, as their structure and content are beyond the control of data warehouse designers. The primary objectives of these systems are high processing performance and availability, which is why queries executed against them are narrowly scoped, often retrieving data one record at a time. It is therefore assumed that operational systems are not intended for the complex queries intended for a Data Warehouse. Furthermore, they retain only limited historical data, while a Warehouse has extended capacity.

The data staging area of the Warehouse serves a dual purpose: it acts both as a temporary storage zone and as the core environment for Extract, Transform and Load processes (ETL). This area bridges the gap between the operational source systems and the data presentation layer, functioning as a space where raw data is cleaned, integrated and restructured into a format suitable for decision-making and analysis. «*It is somewhat analogous to the kitchen of a restaurant, where raw food products*

are transformed into a fine meal. Similar to the restaurant's kitchen, the backroom data staging area is accessible only to skilled professionals. The Data Warehouse kitchen staff is busy preparing meals and simultaneously cannot be responding to customer inquiries. Customers aren't invited to eat in the kitchen. It certainly isn't safe for customers to wander into the kitchen. We wouldn't want our Data Warehouse customers to be injured by the dangerous equipment, hot surfaces, and sharp knives they may encounter in the kitchen, so we prohibit them from accessing the staging area. Besides, things happen in the kitchen that customers just shouldn't be privy to.»[5]

The extraction phase represents the initial step in the process of integrating into a Data Warehouse. This phase involves identifying and reading the relevant data from various source systems. The extracted data is then transferred to a staging area, where it undergoes preliminary processing in preparation for subsequent transformation activities. These operations typically include data cleansing (e.g., correcting typographical errors, resolving inconsistencies in domain values, handling missing or null values, and converting data into standardized format), the integration of data from different, heterogeneous sources, de-duplication and the assignment of surrogate or warehouse-specific keys. The final step in the ETL process is the loading phase. Here, the cleansed and transformed data is transferred into the presentation layer of the Data Warehouse, typically by means of bulk loading systems provided by the data mart systems. Following the loading process, the data mart indexes the newly ingested data to optimize query performance. Once the data marts are fully populated, indexed, and validated, the publishing phase commences, which means notifying the user community of the availability of the updated data.

The data presentation area serves as the portion of the Data Warehouse where data is systematically organized, stored, and made accessible for direct querying by end users, report generators, and various analytical applications. It is build with a series of integrated data marts. This area represents the visible face of the Data Warehouse and the only part of the system where the business community can directly interact with, typically with data access tools and visualization platforms.

The Kimball architecture offers several advantages. One key benefit is the speed of development, as dimensional modeling does not involve normalization, allowing for rapid execution of initial design phase. Furthermore, the system footprint is minimal, since the focus is on individual processes rather than the entire enterprise, resulting in reduced database space requirements. The architecture also supports fast data retrieval, as data is organized into clearly defined fact and dimension tables. Additionally, the approach requires only a small team of designers and planners due to the process-oriented nature of the warehouse. Query optimization is also straightforward, and manageable. Finally, Kimball promotes a conformed dimensional structure, which supports data quality and consistency across business

processes.

Despite many strengths, the Kimball Data Warehousing also presents several limitations. One of the primary concerns is that data is not fully integrated prior to reporting. Additionally, the de-normalized structure is inherent in the Kimball approach can lead to data redundancy, increasing the risk of inconsistencies during updates. Performance issues may also arises as fact tables grow in complexity. Moreover, the dimensional model is less flexible.

2.2.3 Comparison Between Inmon's and Kimball's Architectures

Aspect	**Inmon Architecture**	**Kimball Architecture**
Design approach	Top-down	Bottom-up
Data modeling	Normalized	De-normalized
Data integration	Before reporting	During reporting
Complexity and development time	More complex; longer initial development time	Faster to implement and easier to understand
Flexibility	High flexibility	Low flexibility
Query performance	Complex joins, slower	Optimized for fast querying
Data marts	Created after DW	Built first and integrated

Table 2.1: Comparison between Inmon and Kimball data warehouse architectures

2.2.4 Use Cases: Inmon and Kimball

Both the Kimball and Inmon methodologies regard the Data Warehouse as centralized repository that supports business reporting and rely on ETL processes for data loading. However, they differ fundamentally in their data modeling techniques and the sequence in which data is integrated into the Warehouse. The choices between these approaches significantly affects the initial delivery time of the Warehousing project and the system's ability to accommodate future modifications in the ETL design, therefore, the following factors should be considered:

- Reporting needs: For organization-wide, fully integrated reporting, the Inmon approach is more appropriate, while if reporting is focused on a specific business, the Kimball method tends to be more effective.
- Project Deadline: The normalized data modeling in the Inmon method is more complex and time-consuming compared to Kimball's de-normalized design. Hence, Kimball is preferred when faster project delivery is required.

- Team and Resources: Inmon is more efficient when there is a whole team of developers working behind this warehousing architecture. Kimball's simpler dimensional models can be managed by smaller teams.
- Flexibility and Change Management: Inmon's approach offers greater adaptability to frequent changes in reporting needs. Kimball is better suited for relatively stable environments with predictable requirements.

For the development of an ETL process on Microsoft Fabric, where data modeling will follow the star schema, and considering the need for a rapid pipeline deployment, the individual nature of this thesis work, as well as the limited variability in the requirements of Mediamente Consulting's clients, the most suitable architecture is the one proposed by Kimball.

2.3 Data Modeling

A data model is a conceptual framework that defines how data is structured, stored, and accessed in a database. It outlines the organization of data elements and relationships between them, providing a blueprint for how information will be stored and retrieved. They can be classified into several types, such as hierarchical, network, relational, and multidimensional models, depending on the specific needs and goals of the system.

The key components of a data model are entities, attributes, relationships and constraints. Entities represent the main objects within the system, such as clients, products or orders. Attributes describe the specific characteristics of each entity, such as a name, price, or date. Relationships define the logical connections between different entities, such as the link between a customer and the orders they place. Finally, constraints establish the rules that ensure the consistency and validity of the data, such as unique identifiers or required values.

2.3.1 Star-Schema

One of the most commonly used data models in Data Warehousing is the star schema, which is a multidimensional, de-normalized data model that organizes data into a central fact table and peripheral dimension tables.

The Fact Table

Kimball, in *The Data Warehouse Toolkit* [5], defines the fact table as the primary table in a dimensional model, used to store the quantitative performance metrics generated by the business processes. It captures the measurable outcome of specific business activities and typically resides within a single data mart to ensure clarity

and consistency. Because factual data represent the bulk of the storage in any data mart, it is essential to avoid redundancy by not duplicating these data across multiple locations in the enterprise.

In this context, the term fact refers to a numerical business measure. For example, as Figure 2.3 illustrates, by looking transactions in a drugstore and recording, for each day, the number of units sold and the corresponding revenue for each drug in each store. These recorded values are the facts. Each measurement occurs at the intersection of multiple dimensions - such as time, product, store. This intersection defines the grain of the table, which specifies the exact level of detail or granularity at which the data is stored.

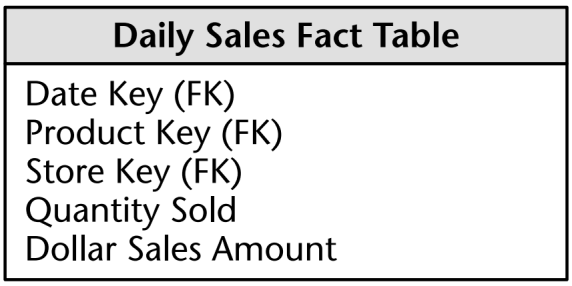


Figure 2.3: Sample fact table. [5]

A row in a fact table corresponds to a measurement, and all the measurements in a fact table must be in the same level of granularity.

A measure must posses the property of additivity because, in Data Warehousing applications, users typically analyze aggregated data rather than individual records in the fact table. While querying, the results often involve multiple rows, and summarizing these rows usually requires performing sum operations to ensure a meaningful aggregation across different dimensions. This constraint excludes the possibility of textual measures in practice, yet, such a scenario is theoretically possible, though rare. Often, textual measurements are descriptive and come from a limited set of predefined values. Designers should strive to include these textual attributes within dimension table rather than fact tables, as this allows for better correlation with other textual dimension attributes and reduces storage requirements. Redundant textual data should never be stored in fact tables. Unless the text is unique for every fact record, it belongs in a dimension table.

Returning to Figure 2.3, if there is no sales activity on a given day for a given product, the record is out of the table. It is very important to not fill the fact table with zeros representing nothing, because these values would overwhelm it and uselessly increase the size of the table. By only including true activity data, fact tables tend to be sparse. Despite this sparsity, they typically account for around 90% or more of the total storage space in a dimensional database. Fact tables are deep,

which means they have a large number of rows but relatively few columns (narrow).

In addition to numerical measures, a fact table includes multiple foreign keys. A foreign key is an attribute that creates a link between the fact table and a related dimension table by referencing the primary key of that dimension, thereby enabling multidimensional analysis. The set of all foreign keys in a fact table constitutes the table's composite primary key, which uniquely identifies each row. In this context, any table possessing a composite key is, by definition, a fact table, as it encapsulates a many-to-many relationship among dimensions. As illustrated in Figure 2.3, a foreign key such as the product key in the fact table corresponds to a marching primary key in the product dimension. When each foreign key in the fact table correctly matches its associated dimension's primary key, the schema is set to preserve referential integrity. This structural consistency ensures reliable joins between fact and dimension tables.

Dimension Tables

Dimension tables play a central role in providing contextual and descriptive information for the numeric data stored in fact tables. These tables contain attributes that describe each row in detail. As such, dimension tables are intentionally designed to include a large number of descriptive, often-like attributes and reporting, therefore, it is common for a dimension to contain between 50 and 100 attributes, each contributing to the understanding of the underlying data. From a structural perspective, dimension tables are typically shallow in terms of row count -usually consisting of fewer than one million rows- but wide in terms of the number and size of columns; each dimension is uniquely identified by a primary key (denoted as PK), which also functions as the foreign key reference in the fact table, thus maintaining the integrity of the star schema.

The dimension attributes serve as the primary source for query constraints, groupings, and report labels. When formulating analytical queries, these attributes are frequently by terms such as "by week" or "by brand", where "week" and "brand" are explicitly defined as dimension attributes. This role makes dimension attributes essential for filtering and aggregating data in user-driven reporting.

The analytical power and value of a Data Warehouse are directly proportional to the quality, completeness, and semantic clarity of its dimension attributes.

In a dimensional data model, the best attributes for dimension tables, are often textual and discrete, these attributes should be expressed using clear, business-relevant terminology rather than cryptic abbreviation or codified values. For instance, in the case of a product dimension, commonly used attributes include a short description (typically 10 to 15 characters), a long description (30 to 50 characters), brand name, category name, product size, among others, as Figure 2.4 shows.

While certain attributes, such as size, may appear numeric, they are still treated as dimension attributes because they function as static descriptors that classify and describe a product, rather than computed quantity. As a matter of fact, a key challenge in dimensional modeling arises when determining whether a particular data field should be classified as a fact or as a dimension attribute. One of the primary criteria involves the role the attribute plays in calculations. Facts are quantitative by nature, hence if a field is used in such arithmetic operations, it is most likely a fact, i.e., sales revenue, units sold, operating costs. Another important consideration is the variability of the data field, attribute that vary frequently across time or entities, particularly in a transactional context, tend to be modeled as facts. In contrast, dimension attributes are usually more stable and slowly changing, which supports their use in classification and filtering operations. For instance, the standard cost of a product might appear to be a static descriptive attribute. However, if this value is updated regularly due to supplier changes or production costs, it may be better represented as a fact to capture its dynamic nature.

A third factor, attributes that serve to filter results, define grouping categories or appear in reporting hierarchies are more appropriately classified as dimension attributes. These are the fields that analysts refer to in "by" clauses - such as "sales by store" "sale by region" or "sales by month".

Cardinality plays a crucial role in choosing if a semantic logic is a fact or a dimension, high cardinality fields that vary substantially and are time-dependent are often treated as fact, whereas low-to-moderate cardinality attributes that describe categories or classifications are usually modeled as dimension attributes.

A final consideration is whether the attribute is descriptive or transactional in nature. Dimension attributes provide context; facts, by contrast convey the "how much" or "how many".

In some cases, however, the classification remains ambiguous. A field may possess characteristics of both a fact and a dimension attribute, depending on business requirements and usage patterns. For instance, a company might treat the standard cost of a product as a stable attribute in some reporting scenarios and as a variable measurement in others. In such scenarios, it may be possible to model the data field either way, as a matter of designer's prerogative.

Star Join Schema

The star join schema, as illustrated in Figure 2.5 represents the structure of the Data Warehouse by displaying all dimension and fact tables, along with the relationships between them. These relationships are established through the use of primary keys in the dimension tables and corresponding foreign keys in the fact table.

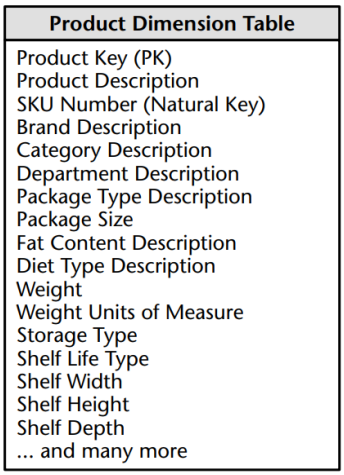


Figure 2.4: Sample dimension table. [5]

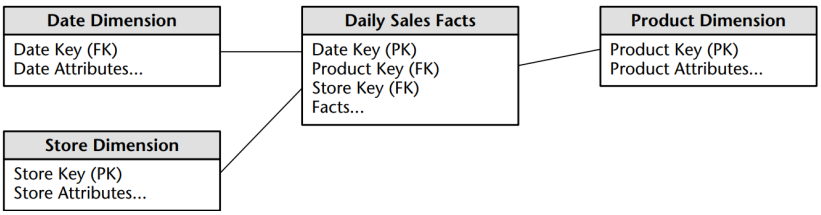


Figure 2.5: Sample Star Schema. [5]

One of the most noticeable characteristics of a dimensional schema is its inherent simplicity and symmetry, which means that every dimension is equivalent; all dimensions are symmetrically equal entry points into the fact table. This simplicity is not just a matter of design elegance - it provides tangible benefits to users by making the data model significantly easier to comprehend and navigate.

Beyond usability, the simplicity of the dimensional model also yields substantial performance advantages. Database query optimizers are able to process these schemas more efficiently due to the limited number of joins required and the predictable structure of the relationships. Specifically, the database engine can apply filters early by constraining the highly indexed dimension tables, and then proceed to access the fact table using the Cartesian product of the dimension keys that satisfy the query conditions. Remarkably, this strategy enables the evaluation of complex n-way joins in a single pass over the index of the fact table, ensuring scalable performance even with large volumes of data.

Atomic, non-aggregated data represents the most granular and expressive form of information in a Data Warehouse. For this reason, the foundation of every fact table should be based on atomic data, ensuring that the design can accommodate

the diverse and often unpredictable nature of business users ad hoc queries. By storing data at its lower level of granularity, dimensional models provide a flexible and extensible framework that supports analytical requirements both foreseen and unforeseen at the time of initial modeling.

One of the key advantages of dimensional modeling is its adaptability, it is possible to add entirely new dimensions to the schema, provided that each fact row can be associated with a single, well-defined value from the new dimension. Similarly, new facts can be incorporated into the fact table as long as they conform with the same level of granularity. In addition, dimension tables can be enriched by introducing new attributes, even if these attributes were not anticipated during the design phase.

This model also allows for increasing granularity of existing dimensions from a given point in time onward. For example, a dimension which was previously tracked at a monthly level can be broken down into daily or hourly intervals as more detailed data becomes available. Such enhancements an often be implemented in-place, simply by appending new rows or columns to existing tables without disrupting the current architecture, in other words, the dimensional model is inherently modular.

2.3.2 Snowflake-Schema

As stated in the previous section, a de-normalized dimensional model refers to a schema design in which attribute values are intentionally repeated across multiple rows within a dimension table. As illustrated in Figure 2.6, the *Department Description* column contains recurring values such as *Bakery* and *Frozen Foods*, which appear numerous times.

Product Key	Product Description	Brand Description	Category Description	Department Description	Fat Content
1	Baked Well Light Sourdough Fresh Bread	Baked Well	Bread	Bakery	Reduced Fat
2	Fluffy Sliced Whole Wheat	Fluffy	Bread	Bakery	Regular Fat
3	Fluffy Light Sliced Whole Wheat	Fluffy	Bread	Bakery	Reduced Fat
4	Fat Free Mini Cinnamon Rolls	Light	Sweeten Bread	Bakery	Non-Fat
5	Diet Lovers Vanilla 2 Gallon	Coldpack	Frozen Desserts	Frozen Foods	Non-Fat
6	Light and Creamy Butter Pecan 1 Pint	Freshlike	Frozen Desserts	Frozen Foods	Reduced Fat
7	Chocolate Lovers 1/2 Gallon	Frigid	Frozen Desserts	Frozen Foods	Regular Fat
8	Strawberry Ice Creamy 1 Pint	Icy	Frozen Desserts	Frozen Foods	Regular Fat
9	Icy Ice Cream Sandwiches	Icy	Frozen Desserts	Frozen Foods	Regular Fat

Figure 2.6: De-normalized product dimension table [5]

If this model is subjected to a normalization process, the resulting structure is referred to as a Snowflake Schema. This modeling approach introduces additional layers of normalization within dimension tables by organizing descriptive attributes into separate related tables. Some data modelers advocate for this schema due to its potential storage efficiency. For example, instead of redundantly storing a 20-byte textual description for a department across 50,000 product records, data engineers may choose to store a compact 2-byte department code in the product dimension and

delegate the corresponding description to a separate department dimension table. In doing so, the size of the primary dimension table is reduced, as it contains only cryptic identifiers rather than lengthy textual attributes. From this point of view, a snowflake schema not only reduces redundancy but also facilitates maintenance. If a descriptor, such as department name, needs to be updated, the change is applied to a single row in the normalized secondary table rather than across thousands of records in a de-normalized dimension table.

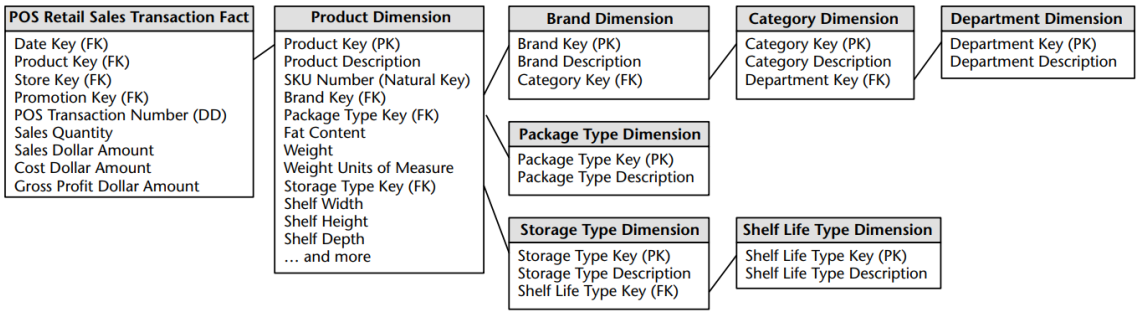


Figure 2.7: Snowflake product dimension [5]

Figure 2.7 illustrates a partial snowflake schema for the product dimension. It is important to know that there is always a join path from every peripheral dimension table to the central fact table, ensuring referential integrity across the schema.

Since the company's framework was designed for a de-normalized data model - where query performance and fast data access are prioritized over storage optimization and maintenance - and given that the data warehouse architecture follows the Kimball approach, the data model adopted for the development of this thesis will be a star schema rather than a snowflake schema.

2.4 Data Marts

The Data Mart is a type of data storage architecture. Both Data Warehouses and Data Marts are used for storing and managing data to support decision making processes. However, a data mart is project-oriented or department specific, focusing on the analytical needs of a particular business unit or functional area. As such, it is limited in scope compared to a Data Warehouse and generally contains a subset of the data stored in the data warehouse, optimized for specific user requirements.

Similarly to the Data Warehouse, a data mart is subject-oriented, although faster to implement due to its narrower scope and smaller size.

Moreover, a data mart can be independent, which means it does not rely on a centralized warehousing system. Instead, it extracts and processes data from various

heterogeneous operational sources to create a specialized data set for a specific business need. Otherwise, it can be a dependent data mart, built from an existing Data Warehouse, with this approach, data is extracted, transformed, and loaded from the Warehouse to the data mart, which then focuses in a specific business function.

An hybrid solution combining both dependent and independent approaches is also possible, extracting data from the Warehouse and from the operational sources; with this approach, organizations can merge structured and unstructured data in a single system for specific analysis.

2.5 Metadata

Metadata encompasses all information related to the system that is not the actual data itself. It exists in multiple forms, each designed to address the specific needs of different user groups within the Data Warehouse ecosystem. At the operational level, source system metadata describes the structure and format of input data sources. This include schema definitions, file layouts, and copybooks, which are essential for guiding the data extraction process. Once data enters the staging area, another layer of metadata is introduced: staging metadata, which supports the transformation and loading procedures (ETL). It defines mapping between source and target structures, specifies transformation rules, outlines data cleansing protocols, and establishes aggregation logic.

Additionally, it includes metadata related to the scheduling and execution of ETL jobs - such as batch run logs, and error tracking records. Furthermore, ETL scripts and custom code written to implement these procedures can also be considered metadata, as they encapsulate the logic behind data manipulation tasks.

On the other hand, DBMS-level metadata includes system catalog information such as system tables, partitioning configurations, indexing strategies, view definitions, and database security configurations - such as user privileges, roles, and grants.

In the implementation of the company's ETL framework on Microsoft Fabric, metadata tables play a crucial role in ensuring efficient and reliable data processing. These metadata tables are responsible for managing the lifecycle of the tables involved, orchestrating the different data ingestion and reading flows, and overseeing overall process control.

Chapter 3

Data Lake

3.1 Data Lake

A Data Lake is a centralized data repository designed to store both raw and transformed data, serving as a foundational layer for analytical and reporting workflows. Unlike traditional databases that enforce strict schema at write-time (schema-on-write), Data Lakes follow a schema-on-read approach, allowing data to be ingested in its native format and structured later as needed. This flexibility enables the storage of diverse data types, including structured data (such as relational tables), semi-structured data (e.g., CSV, JSON; XML), unstructured data (text files, emails, log files) and binary data (images, videos, audio files). As a result, querying a Data Lake often resembles a search engine-like operation, where exploratory and ad-hoc queries are performed across heterogeneous data formats.

One of the primary reasons for adopting a Data Lake is that, in many cases, the questions that data can help answer are not known in advance. By storing all data - structured, semi-structured, unstructured, and binary - in its raw format, a Data Lake preserves the potential value from it at any point in the future. Another significant advantage is that Data Lakes help eliminate data silos, which are isolated data repositories that prevent seamless access and integration. Moreover, Data Lakes are typically built on low-cost, scalable storage systems integrated with high-performance compute engines, often provide by cloud platforms. These platforms enable on-demand processing, allowing transformation in the ETL pipeline to occur at query time, rather than requiring preprocessing; this is a schema-on-read approach. Since data is stored in its original, unprocessed format, a data lake is inherently adaptable and highly flexible. It can quickly accommodate changes in business requirements, or data sources without costly redesign. Lastly, users can access raw data in real time, enabling rapid exploration and experimentation.

Despite their flexibility, scalability and real-time access to data, Data Lakes also presents notable disadvantages. One of the primary concerns is that data is ingested

and stored in its raw form, without any immediate quality checks and standardization. This can lead to a disorganized and unmanageable repository, called a data swamp, where data lacks structure, clarity, and reliability. In such environments, without robust governance mechanisms, including metadata management, data cataloging, and access controls, the Lake becomes difficult to navigate and interpret, making the stored information practically unusable. Besides, since data is extracted from heterogeneous sources, it often remains non-integrated, creating silos within the lake itself and complicating cross - functional analysis. The absence of a clear, predefined purpose can lead to inefficient storage use, thus, increasing operational costs. Furthermore, complex analytical queries over raw or semi-structured data may demand high computational resources, which can result in significant performance bottlenecks that affect the entire system.

3.1.1 Comparison Between Data Warehouses and Data Lakes

Aspect	**Data Lake**	**Data Warehouse**
Data Storage	Contains raw, unstructured data, and can store the data indefinitely.	Contains structured, cleaned and processed data.
Users	Data is used by scientists and engineers due to interest in its raw forms.	Data is used by managers and business-end users for analytical purposes.
Analysis	Predictive analytics, machine learning, data visualization, BI, big data analytics.	Data visualization, BI, data analytics.
Schema	Schema-on-read.	Schema-on-write
Processing	ELT (Extract, Load, Transform). In this process, the data is extracted from its source for storage in the data lake, and structured only when needed.	ETL (Extract, Transform, Load). In this process, data is extracted from its source(s), scrubbed, then structured so it's ready for business-end analysis.
Cost	Inexpensive, also data lakes are also less time-consuming to manage, which reduces operational costs.	Data warehouses cost more than data lakes, and also require more time to manage, resulting in additional operational costs.

Table 3.1: Comparison between Data Lake and Data Warehouse

In this project, the initial source files and tables will be stored in a Data Lake, then, as the ETL framework enters the second level of processing, tables will be stored in the Warehouse.

Chapter 4

Microsoft Fabric

Microsoft Fabric is an advanced, integrated data analytics platform designed to unify the entire data life-cycle within a single, coherent, and scalable environment. Specifically, Fabric seamlessly combines the process of data extraction and loading (Extract, Load), processing, ingestion, and transformation (Transform), real-time-event routing, and comprehensive, interactive reporting.

A key distinguishing feature of Microsoft Fabric is its ability to harmoniously integrate multiple tools and components that have traditionally been separated, such as Data Warehouses, ETL, pipelines, business intelligence tools like Power BI, and machine learning environments. Fabric also integrates within its system Office 365 and it is supported by Microsoft Copilot. Besides, Copilot Microsoft Fabric natively incorporates artificial intelligence capabilities, providing predictive analytics, automation, and advanced data analysis tools without requiring manual integrations or external solutions.

The platform is delivered as a Software-as-a-Service (SaaS) platform, a software distribution model in which a cloud provider hosts applications and make them available to end users over the internet.

4.1 Terminology used in Microsoft Fabric

A *capacity* refers to a set of computing resources that are available at a given time for use within Microsoft Fabric. There resources are allocated to support the execution of operations such as data processing, and analysis across the platform.

An *Experience* is a collection of specialized features and tools designed to fulfill a specific purpose or use case. In Microsoft Fabric, experiences are tailored to various data-centric domains and include: Fabric Data Warehouse, Fabric Data Engineering, Fabric Data Science, Real-Time Intelligence, Data Factory, Power BI. Each experience

provides a contextual interface and a suite of functionalities optimized for its intended tasks.

An *item* is a unit of functionality within a given experience. Users can create, modify, and delete items depending on their permissions. Each item type supports specific capabilities and workflows. For instance, within the Data Engineering experience, common item types include: Lakehouses and Spark Job Definition.

A *Tenant* represents a unique instance of Microsoft Fabric assigned to an organization. It is linked to the organization's Microsoft Entra ID and provides identity and access management, tenant-wide configuration, and centralized governance across experiences. An organization posses a single Tenant.

4.2 OneLake

OneLake is a core component of Microsoft Fabric, built on top of Azure Data Lake Storage (ADLS) Gen2, providing a SaaS data storage solution unified at the tenant level. It is designed to serve both professional developers and citizen developers (non-technical users) requiring streamlined access to advanced data capabilities. Unlike ADLS, which necessitates manual configuration steps - including creation of storage accounts, role-based access controls (RBAC), regional and redundancy configurations, and management via Azure Resource Manager - OneLake is provisioned natively at the time a Microsoft Fabric tenant is granted, requiring no additional setup.

OneLake addresses data fragmentation by providing a unified storage architecture that prevents formation of data silos. This design facilitates consistent data discoverability, secure sharing, and coherent enforcement of governance policies across the organization, thereby promoting centralized and governed data management.

There is only one OneLake instance for each organizational tenant. It offers a global namespace that spans users, regions, and cloud environments. Its hierarchical structure, shown by Figure 4.1, consists of:

- The tenant at the root level, representing the highest organizational scope.
- Multiple workspaces within the tenant, which serve as logical organizational units enabling different departments or teams to delegate ownership and implement access control policies with fine granularity. Workspaces function as containers for data assets, and all data stored within OneLake is accesses through these data objects.
- Several lakehouses within each workspace.

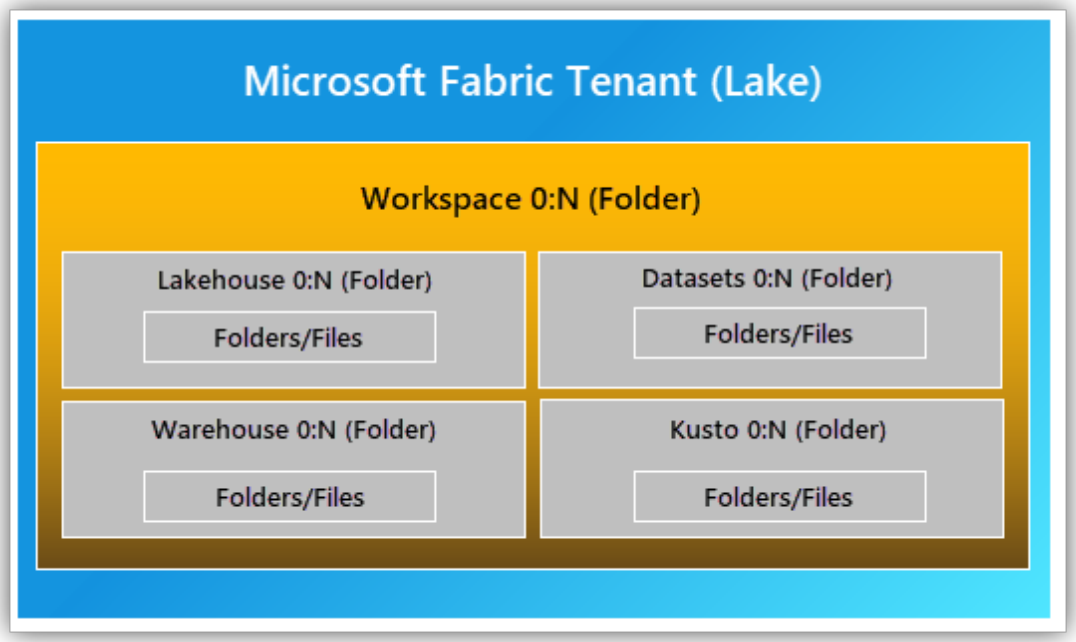


Figure 4.1: Microsoft Fabric's architecture [6]

Analogous to Microsoft Office applications save Word, Excel, and PowerPoint files to OneDrive, Microsoft Fabric stores its key data artifacts (lakehouses, warehouses...) within OneLake. Importantly, all Fabric data entities, persist their underlying data in OneLake using the Delta Parquet format. This standardization facilitates seamless interoperability: for example, a data engineer may ingest data into a lakehouse using Apache Spark, while concurrently, a SQL developer can load data into a fully transactional data warehouse using T-SQL; both activities contribute to the same data lake.

Figure 4.2 shows how data generated by Microsoft Fabric workloads, as well as data from ADLS-compatible applications, is stored in Delta Parquet format within OneLake, regardless of the workspace to which it belongs. Although data resides in separate workspaces, users who have appropriate permissions can access data across different workspaces - for example, a user with access to Workspace B can also access Workspace A's data if granted the necessary permissions.

The key point is that all data is physically stored in a single location, but logically organized into workspaces, making it accessible to a variety of tools without the need for duplication. Users can easily explore data in OneLake through the OneLake File Explorer for Windows.

Microsoft Fabric avoids data redundancy through the use of *shortcuts*. A shortcut

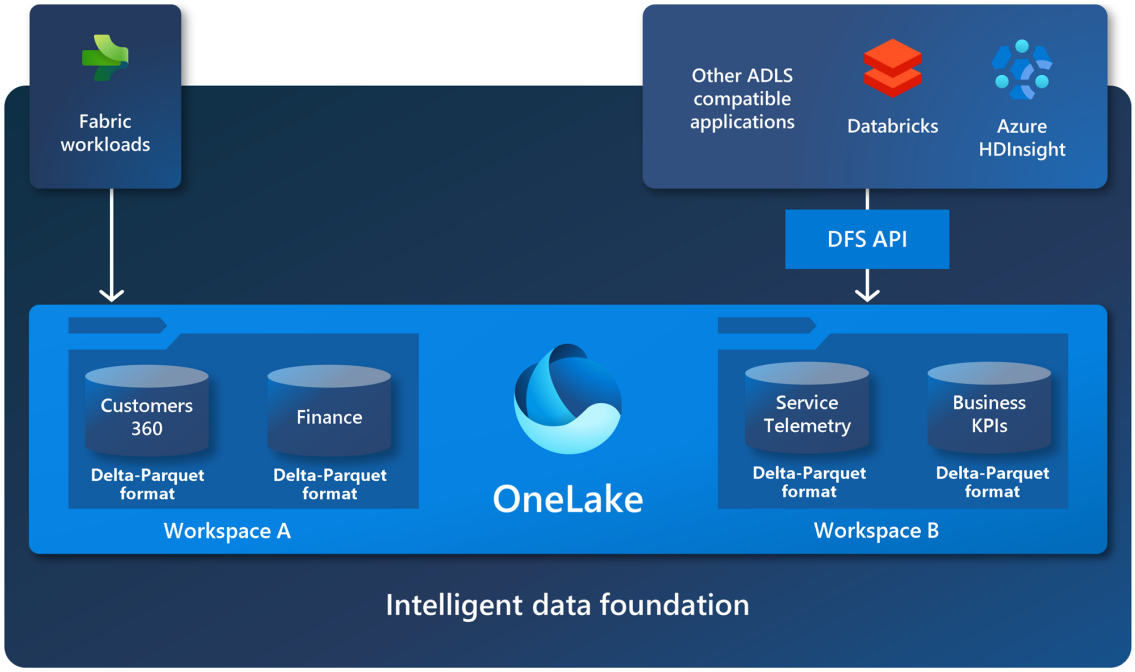


Figure 4.2: Unified data from different sources in Delta Parquet Format [7]

is a reference to data stored in other file locations, which may reside within the same workspace, across different workspace, or even outside of OneLake, in external sources such as ADLS, Amazon S3, or Dataverse. Regardless of the physical location, shortcuts make files and folders appear as if they are stored locally within the workspace, enabling unified access and interaction without duplication. Shortcuts are dynamic rather than static, meaning they always reflect the current state of the data they reference. If the underlying information changes, those modifications are immediately visible in OneLake through the shortcut. Hence, users always have access to up-to-date without replication or manual synchronization; nevertheless, historical versions of the referenced data are not preserved, therefore, it is the user's responsibility to implement appropriate mechanisms for capturing or archiving data snapshots before any changes occur in the source, if historical traceability is required.

4.3 Experiences

4.3.1 Fabric Data Engineering

It is designed specifically to support the development, orchestration, and execution of data engineering workflows at scale. It provides a comprehensive set of tools and services that enable engineers to prepare, transform, and manage large volumes of data across structured and unstructured sources.

Lakehouse

A Lakehouse in Microsoft Fabric is built on top of OneLake. The term *Lakehouse* derives from the fusion of *Data Lake* and *Data Warehouse*, reflecting its hybrid architecture.

This models enables to store both structured and unstructured data in its raw form (as in data lake), while also supporting ACID transactions, schema enforcement, and high-performance analytical queries (as in a traditional warehouse)

In the context of Lakehouse architecture, data is typically stored using the Delta Parquet format. However, to fully understands the significance and capabilities of this format, it is essential first to examine the underlying Parquet file format on which is built. Parquet is a columnar storage file format, enabling efficient compression since values within a column are often of the same type and similar in nature, archiving high compression ratios.

Delta Parquet format is an open-source layer that can be described as the union of Parquet Format, ACID transactions, transaction log and metadata management. A substantial difference between Delta Parquet and Parquet format lies in mutability and data management capabilities. While traditional Parquet files are essentially immutable, Delta Parquet introduces an incremental and transactional model that allows modifications such as `MERGE`, `UPDATE`, and `DELETE` operations directly on the dataset. This is made possible through the Delta transaction log, which records a complete history of all changes applied to a Delta table. As a result, Delta Parquet supports time travel, enabling users to query previous versions of a table and observe how its contents have evolved over time.

From a data engineering perspective, this capability has significant advantages. For example, when working with a fact table, instead of reloading the entire dataset after each ingestion cycle, one can simply ingest only the new data (the delta) and apply it incrementally. The same approach applies to dimension tables. This leads to reduce processing time, improve performance, and lower storage and compute costs.

SQL Analytical Endpoint

Microsoft Fabric provides a SQL-based experience for Delta tables within the lakehouse. This SQL-based interface is referred to as the SQL Analytical Endpoint. Users can analyze data in Delta tables using T-SQL (Transactional SQL), define, and store functions, generate views, and implement SQL-level security. The creation of a lakehouse instance automatically provisions a SQL Analytical Endpoint, which is configured to reference the storage location of the lakehouse's Delta tables. Once a Delta table is created within the lakehouse, it becomes accessible for querying through the SQL Analytics Endpoint. It operates in read-only mode with respect to the the Delta tables in the lakehouse, nevertheless, it exists the flexibility to define

functions, create views, and enforce SQL object-level security to effectively manage data access and structural organization.

Spark Application

An Apache Spark application is a distributed data processing program developed using one of the supported Spark APIs, in Microsoft Fabric's case, is Python (PySpark). A Spark application is designed to process large scale datasets across a cluster of machines, leveraging Spark's in-memory computation and parallel processing capabilities. Each Spark application consists of a driver process, which defines the application logic and coordinates execution, and a collection of executors, which run tasks on the data distributed across the cluster.

When submitted for execution, the application is logically broken down into jobs, where each job corresponds to a specific action on a Spark DataFrame or RDD. Each job is further divided into stages, which are sequences of tasks that can be executed in parallel. The division into stages is determined by data shuffling or transformation that requires data movement across partitions. A task represents the smallest unit of work in Spark and is executed on a single partition of the data. This architecture allows Spark to scale efficiently, handling petabyte-scale datasets with fault tolerance, thanks to its DAG (Directed Acyclic Graph) execution model and support for lineage tracking and recomputation in case of node failures.

In the context of Microsoft Fabric, Spark applications are deeply integrated into the platform and can be run as part of Spark Job Definitions within the Data Engineering experience.

4.3.2 Fabric Data Factory

Microsoft Fabric's Data Factory offers similar a service to the Data Engineering experience in terms of transformation and orchestration. However, the key difference lies in the tools employed and level of coding required. Data Engineering provides a high-code environment, primary leveraging Spark notebooks where users are expected to be proficient in Python or Scala. In contrast, Data Factory is designed for users with less programming experience, offering a low-code interface through Dataflows Gen2, which enables data transformation via a graphical, drag-and-drop approach.

Dataflow Gen2

At its core, Dataflow Gen2 maintains the intuitive, low-code interface facilitated by Power Query, a data transformation and preparation engine developed by Microsoft, it allows users to perform complex operations without extensive programming language, for example renaming elements, using first rows as headers, changing data types, filtering, formatting, adding custom columns or even merging queries. These

transformations are recorded in a dedicated pane within the Power Query Editor, providing a clear, step-by-step view of the data transformation process. Users can easily modify, delete, or reorder these steps.

«Microsoft Fabric Dataflow Gen2 connects to various data sources and performs transformations using Power Query Online, which is part of Microsoft Fabric. It enables users to create and manage data pipelines that can ingest, prepare, transform, and analyze data. Finally, it delivers the output to destinations such as Azure SQL, Lakehouse, Warehouse, and Azure Data Explorer» [8]

Notably, Gen2 extends its functionality by supporting a wide array of data destinations, including Fabric Lakehouse, Warehouse and even SQL Databases.

A distinguishing feature of Dataflows Gen2 is its enhanced integration with data pipelines. Users can incorporate dataflows as activities within pipelines, facilitating more complex and orchestrated data processing scenarios. This integration supports conditional logic and dependencies.

From a usability perspective, Dataflows Gen2 introduces several user-friendly features, such as auto-save and background publishing, which streamline the development process. Enhanced monitoring and refresh history functionalities offer detailed insights into Dataflow executions, significantly aiding in troubleshooting and performance tuning.

Dataflows are designed with a graphical user interface (GUI) that closely resembles other Microsoft Office 365 applications and Power BI. This familiar interface targets users who need to perform data transformations without having any programming experience. For more advanced users, Dataflows also support the use of M scripts, enabling the definition of complex transformations through code. Moreover, Dataflows automatically manage dependencies between scripts, inputs, and outputs by allowing users to reference one script from another simply by passing its name, thus simplifying the orchestration of transformation logic.

4.3.3 Fabric Data Warehouse

The Fabric Data Warehouse experience is designed to deliver the familiar capabilities of a classical data warehouse while leveraging the scalability, high performance, and centralized storage architecture provided by Microsoft Fabric. It enables users to work with large-scale datasets using T-SQL, offering an environment that integrates directly with OneLake.

4.4 Computation Engines

Although modern applications often decouple storage from compute, data is frequently optimized for a specific processing engine, limiting its reuse across heterogeneous analytical workloads. Microsoft Fabric addresses this limitation by enabling multiple analytical engines, as shown in Figure 4.3, to operate on data stored in Delta Parquet format. This approach removes the need for data duplication when switching between engines and ensures all workloads access the same version of the data.

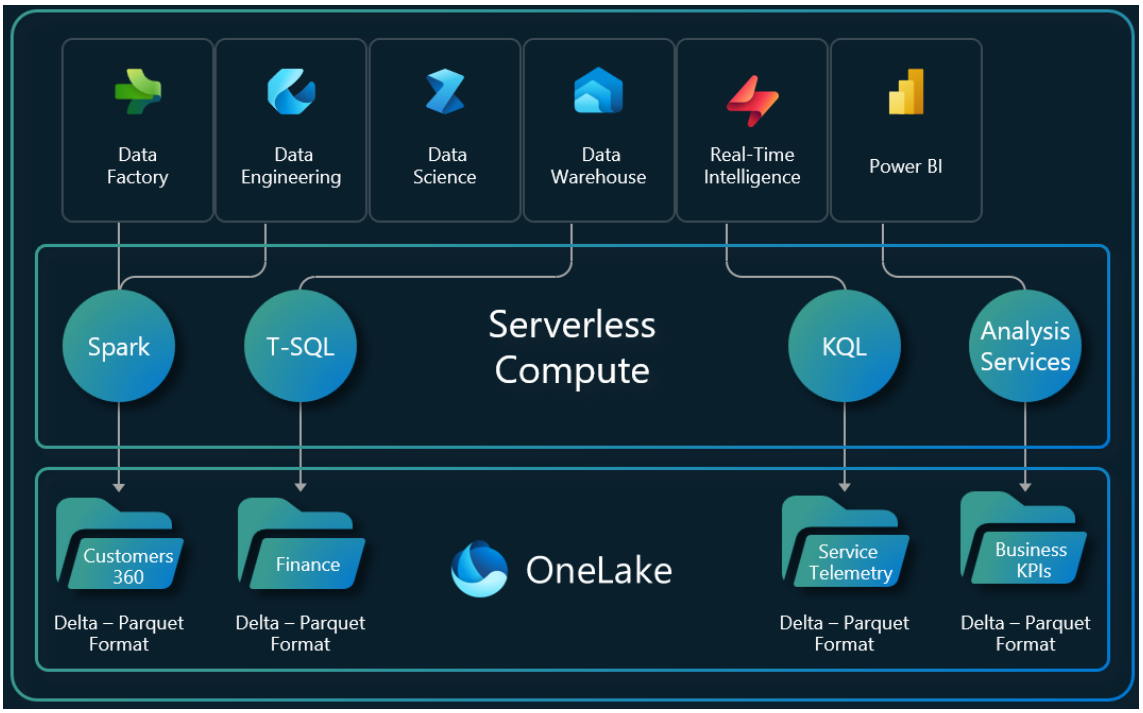


Figure 4.3: Microsoft Fabric serverless computations [7]

As a result, users are free to choose the most appropriate engine for their specific task. For instance, a team of SQL engineers can leverage the T-SQL engine to build a fully transactional Data Warehouse. Concurrently, a data scientist can directly analyze the same datasets using the Spark engine and its associated open-source libraries, without requiring data export, transformation, or specialized connectors.

4.5 Limitations of Microsoft Fabric

In the previous sections, several advantages of Microsoft Fabric were described. Nevertheless, considering that the platform released in May 2023, it still presents notable flaws and limitations that should not be overlooked.

Being a SaaS platform, Fabric inherently offers less customization compared to on-premise applications. Users have limited control over core infrastructure components such as networking, compute, storage, and security. This abstraction layer, while beneficial for ease of use and rapid deployment, reduces flexibility for advanced or highly regulated scenarios. Consequently, companies needing to enforce strict security or privacy standards may find the pre-configured and opaque nature of Fabric's security controls insufficient.

In complex enterprise use cases that involve intricate data pipelines or highly optimized query patterns, Fabric's performance can become a bottleneck. While it is effective for standard workflows, its capabilities might fall short when handling mission-critical applications that demand high configurability and precise resource tuning.

Another critical concern is the absence of standardization across similar components. For instance, Fabric offers both Data Pipelines and Dataflows, which serve comparable functions. Despite their conceptual similarity, Data Pipelines can be edited at any time and moved across workspaces by coping them. In contrast, Dataflows require explicit ownership for modifications and can only be moved after being exported locally. These inconsistencies, while seemingly minor, contribute to a fragmented user experience and can introduce friction in cross-functional teams.

4.6 Alternatives to Microsoft Fabric

An alternative on-premise data integration tool that is natively supported by Oracle Databases is Oracle Data Integrator (ODI). In contrast, Microsoft Fabric relies on a file-based architecture using Delta Parquet files as its backend storage format, which imposes some limitations on table structure definitions and standard DML operations.

"On-premise" means that the software is installed and managed directly on the company's local servers, within its own infrastructure. As a result, it does not rely on the cloud, data and applications remain entirely within the organization's systems. The company is fully responsible for updates, maintenance, and security. This flexibility can increase operational overhead.

4.6.1 Oracle Data Integrator (ODI)

In addition to being the foundation for the Mediamente Consulting ETL framework, ODI offers several advanced features that make it a powerful tool for data integration. Notably, it allows for the creation and customization of Knowledge Modules (KMs), which are useful to automate operations. For example, KMs can automatically detect primary and foreign keys of a table without requiring explicit user definitions.

Additionally, ODI supports integration with a wide range of databases and data sources, including technologies such as Cassandra and Hadoop. Furthermore, ODI benefits from comprehensive documentation and robust support from Oracle, unlike Fabric, which is relatively a new product on the market, the documentation provided by Microsoft is still limited in scope, often requiring users to adopt a learn-as-you-go approach. Finally, ODI is generally considered more robust and reliable for complex data loading and transformation processes, particularly when working with high volumes of structured data. Nevertheless, being an on-premise solution, its performance is inherently tied to the capacity and health of the underlying hardware, potentially limiting scalability compared to cloud-based alternatives, such as Fabric.

ODI also presents several notable limitations. The setup process is relatively complex, the user interface is not particularly user friendly, and error handling it often non-intuitive. In addition, integration with cloud environments is not straightforward and typically requires the use of Oracle Data Integrator Cloud, a separated tool.

4.6.2 Workato

Gartner, a leading provider of expert guidance and tools for strategic decision-making, has classified Workato as one of the top Platform-as-a-Service (PaaS) tools for 2025, as illustrated by the Magical Quadrant in Figure 4.4. It achieved the highest score in Completeness of Vision - a metric that assesses the vendor's innovation, understanding of market needs, and long-term strategy - and ranked among the top three in Ability to Execute, which evaluates the vendor's capacity to deliver products, services, customer support, and overall viability. Notably, Workato obtained an overall higher positioning than Microsoft, which is also included in the *Leaders* quadrant.

Workato stands out for its ease of use and intuitive interface, enabling the design of complex workflows through a fully visual and code-less approach. In addition to its low-code environment, Workato—like Microsoft Fabric—also allows the use of SQL and Python for more advanced data transformations and logic. It includes a library of pre-built connectors that streamlines integration across heterogeneous systems such as ERPs, CRMs, collaboration tools, and cloud services. Automation is both flexible and scalable, supported by advanced features such as error handling, conditional logic, and real-time monitoring, all enabled by the user interface. Furthermore, Workato's responsive technical support and an active community further enhance the overall platform experience, by helping users with on-the-road difficulties.

A main difference with Microsoft Fabric is the storage of data, as stated, the central storage of Fabric, OneLake, stores data as Delta Parquet files. Workato, on the other hand, relies on proprietary storage components designed for structured and file-based data management:

- Data Tables: structured, spreadsheet-style datastore built on an entity–attribute–value (EAV) model, fully managed and created via the Workato UI with no backend SQL database required.
- FileStorage: Provides secure storage for files such as CSV or JSON.

Figure 4.4: Gartner Magic Quadrant for Integration Platform as a Service [9]

Despite its leadership in the Integration Platform as a Service (iPaaS) market, Workato presents several limitations that may affect adoption and long-term usability in specific contexts. One of the primary concerns are API rate limits and functional constraints may restrict performance in data-intensive scenarios, reducing scalability for enterprise-grade use cases. Development flexibility is also affected by limited debugging tools and restrictions on API call visibility, which can hinder troubleshooting and fine-tuning of integrations. The Software Development Kit (SDK), while powerful, introduces a steep learning curve. Moreover, support for complex use cases may require custom coding, partially offsetting the benefits of the low-code environment. Although the user interface is intuitive, the configuration of more advanced workflows and technical integrations still demands a non-trivial learning period.

Chapter 5

Proposed Solution: Mediamente Consulting's ETL Framework

A possible solution to implement an automated pipeline that is capable of managing an Extract, Transformation, Load process could be structured as follows. The framework consists of three levels: L0, L1, and L2, with each level performing a different function as Figure 5.1 shows.

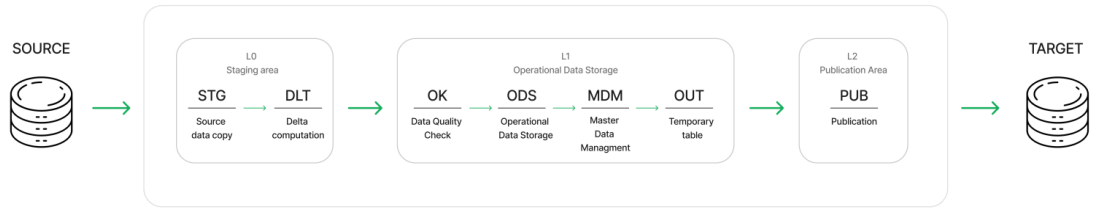


Figure 5.1: ETL framework designed by Mediamente Consulting

5.1 L0

The L0 represents the initial or staging phase of the Data Warehouse (DWH), where data extraction from various source systems. These systems can vary in nature; the most common include operational database systems and files produced by external providers.

The proposed framework supports two primary data extraction strategies.

- The first approach is *Full Extraction* involves retrieving the entire dataset from the source system without accounting for previously loaded records. While

this approach ensures data completeness and simplifies implementation, it is generally inefficient for large-scale datasets, as it demands considerable time and computational resources.

- The second approach is a combination between *Initial Load* and *Delta Computation*. Initial Load is in charge of the first loading into the DWH, while Delta Computation adopts an incremental extraction approach, wherein only changes since the previous data load are retrieved. This method is suitable for environments where the DWH is periodically updated, as it minimizes data movement. Two main scenarios can arise in the application of Delta Computation:

  – Presence of update data: When source systems maintain update timestamps, the framework can monitor these indicators to detect modified or newly inserted records. This approach is particularly effective when dealing with large tables, as it prevents unnecessary full reloads. However, a notable limitation is the inability to detect physical deletions at the source. To overcome this, a dedicated log table can be implemented to track and reconcile deletions across extractions.

  – Absence of update data: In cases where the source data lacks explicit indicators of change, the solution involves extracting the entire dataset into a staging table (`STG`), where each record is labeled with a unique job identifier (`JOBID`) representing the extraction timestamp in the format `YYYYMMDDHHMMSS`. Additionally, a human-readable timestamp (`INS_DATE`) is stored in the format `DD-MM-YYYY HH:MM:SS` reflecting the time at which data were loaded into the table. This mechanism enables the framework to infer changes and capture deletions by comparing data across extractions. However, it is computationally expensive and less efficient for large datasets.

The records are loaded into the `STG` table using an `APPEND` operation, meaning the table is never cleared during the ETL process; new records are added after the existing ones. The DLT tables also use an `APPEND` strategy. These tables can either retain all delta records from every ETL run without ever being emptied, or alternatively, store only the current and the previous N readings. In this second case, an additional table called `DLT_HIS` is populated to maintain the full historical record of all deltas generated since the beginning of the ETL process.

In summary, for each data ingestion flow, all source tables will be read first and their content will be loaded into a corresponding staging table (`STG`), with additional columns for `JOBID` and `INS_DATE`. Subsequently, only the newly inserted, deleted and modified records will be stored in delta tables named `DLT` and `DLT_HIS`, with an additional column denominated `FLG_NEG`, which accepts binary values, 0 if the current record is new, 1 if the record has been deleted from the source.

Descriptive Field 1	...	Descriptive Field n	JOBID	INS_TIME

Table 5.1: `STG` table structure

Descriptive Field 1	...	Descriptive Field n	JOBID	INS_TIME	FLG_NEG

Table 5.2: `DLT` table structure

Descriptive Field 1	...	Descriptive Field n	JOBID	INS_TIME	FLG_NEG

Table 5.3: `DLT_HIS` table structure

5.2 L1

The L1 layer is responsible for data transformation tasks, including activities such as «cleansing the data (e.g., correcting misspellings, resolving domain conflicts, handling missing values and parsing into standardized formats), integrating data from multiple sources, removing duplicates, and assigning warehouse surrogate keys»[5]

Data cleaning involves several phases [10]:

- Data Analysis: Detection of errors and inconsistencies to be removed.
- Definition of transformation workflow and mapping rules: Use of a schema translation to map sources into a common data model, for a Data Warehouse, a relational representation is typically employed. It should be defined a transformation workflow.
- Verification: Test and evaluation of the output of the transformation workflow.
- Execution of the transformation operations
- Backflow of Cleaned Data: The cleaned data should ideally be written back to the original sources to ensure that legacy applications also benefit from the improved data quality and to prevent redundant cleaning in the future extraction processes. Since Mediamente Consulting does not have any control over the sources, the proposed framework uses `DLT` tables to help mitigate repeated cleaning of the same data by storing and reusing previously processes delta records.

As mentioned, the input of this layer consists of the `DLT` tables. During the execution of the functions defined at this level, four additional tables will be generated: `OK, ODS, MDM,` and `OUT`.

5.2.1 OK

Records extracted from the `DLT` table are first processed through a temporary table, which is not physically stored. A `ROW_NUMBER` function is then applied using the following query:

```
ROW_NUMBER() OVER (PARTITION BY PRIMARY_KEY ORDER BY JOBID DESC,
                                FLG_NEG ASC)

SELECT * FROM ...  WHERE ROW_NUMBER = 1
```

This ensures that, for each record identified by a primary (or aggregate) key, only the most recent version is selected, giving priority to newly added records over deleted ones.

After this step, referential integrity constraints are verified, followed by data quality operations, which include:

- Data Validation: Ensures that the data is accurate, complete, and complies with predefined rules (e.g., correct format, valid ranges, non-null values)
- Data cleansing: Removes incorrect, duplicate, or missing data.

Records that do not pass the `ROW_NUMBER` query are permanently discarded. In contrast, records that fail referential integrity or data quality checks are stored in a table named `E$` or `ERR`. These records are reconsidered during each new data ingestion cycle when the temporary table is created, in order to determine whether new information has been added that would allow them to be successfully processed. This table includes a `DESCRIPTION` column that details the reason of failure during quality and validation checks.

Once these operations are completed, the cleaned records are stored in the OK table.

The loading mode used for any `OK` table is `TRUNCATE-INSERT`, meaning that for each data ingestion cycle, the table's contents are completely cleared before the newly processed records are inserted. Instead, the records on the `E$` tables are kept until they pass validation and quality controls or after a specific number of ingestion cycles are completed, this number is specified in a metadata table.

Descriptive Field 1	**...**	**Descriptive Field n**	**JOBID**	**INS_TIME**	**FLG_NEG**

Table 5.4: `OK` table structure

Descriptive Field 1	...	Descriptive Field n	JOBID	INS_TIME	FLG_NEG	DESCRIPTION

Table 5.5: `E$` table structure

5.2.2 Operational Data Store (`ODS`)

This step is responsible for the historization of data, where information is consolidated to ensure consistency and uniformity. At this stage, a certified copy of the operational data is stored - validated, secure, and considered reliable.

The `ODS` tables are the first to be given physical primary keys that uniquely identify each record. In previous stages, record uniqueness was managed through references to table names and composite keys; in the ODS, a concrete primary key is formally established.

The loading mode for these tables relies on a `MERGE` statement, which either inserts new records or updates existing ones based on the defined primary key. This process introduces two additional columns: `JOBID_UPD` and `UPD_TIME`, which record the update data for each row. The format of these columns is identical to `JOBID` and `INS_TIME`, respectively.

In the case of a new record being loaded into the ODS for the first time, all four columns (`JOBID`, `INS_TIME`, `JOBID_UPD`, and `UPD_TIME`) will be populated with identical values corresponding to the current ingestion cycle. If the primary key already exists in the `ODS`, only `JOBID_UPD`, and `UPD_TIME` are updated to reflect the new ingestion cycle, while `JOBID`, and `INS_TIME` preserve the data from the original insertion.

Descrpt Field 1	...	Descrpt Field n	JOBID	INS_TIME	FLG_NEG	JOBID_UPD	UPD_TIME

Table 5.6: `ODS` table structure

5.2.3 Master Data Management (`MDM`)

This stage is in charge of data enrichment, which consists in augmenting records with additional attributes derived from both internal and external sources - such as master data, addresses, or contact details.

Additionally, this stage performs the generation of surrogate keys, which are internally generated unique identifiers used in place of natural keys (e.g., customer codes). Surrogate keys are introduced to ensure data independence, as they remain stable even when natural keys change over time. Furthermore, surrogate keys facilitate data integration by avoiding conflicts that may arise from differing key definitions across heterogeneous sources. Finally, they contribute to improved performance, being generally more efficient in terms of storage space and join operations compared to natural keys.

Similarly to `ODS` tables, `MDM` tables are loaded with a `MERGE` operation.

Descrpt Fields	...	JOBID	INS_TIME	FLG_NEG	JOBID_UPD	UPD_TIME	SURROGATE KEY

Table 5.7: `MDM` table structure

5.2.4 `OUT`

The `OUT` table serves as a temporary staging area designed to contain only the most recent data modifications immediately before their final insertion into the designated target table. One of the defining characteristics of the `OUT` table is its structural consistency with the target table: it shares the exact same schema, ensuring that data transferred during the final loading phase remains coherent to the expected format without requiring further structural adjustments.

The population of the OUT table is not a generic process but is instead driven by functional requirements derived from detailed business analyses. Data are selected and transformed through a series of join operations and logic rules tailored to the specific needs of the business workflow. These transformations may include field derivations, conditional mappings, or enrichments aligned with operational or analytical goals. The OUT table, therefore, acts as a critical bridge between the raw, processed data and its final, business-ready form, enabling a flexible yet controlled integration into the target system.

At this stage, the data model used to structure the target system is also defined. The choice is typically between a star schema and a snowflake schema, depending on the complexity and normalization level required by the use case. As detailed in chapter 2, this project adopts the star schema approach.

Records are loaded into `OUT` tables using a `TRUNCATE-INSERT` operation.

Descrpt Fields	...	JOBID	INS_TIME	FLG_NEG	JOBID_UPD	UPD_TIME	SURROGATE KEY

Table 5.8: `OUT` table structure

5.3 L2

This layer represents the final stage of the data pipeline and is the closest to the end-users. At this point, data are fully loaded into the data warehouse in a form that is ready for reporting, analysis, and decision-making processes. The data contained in this layer is aggregated, structured, and optimized to support efficient and high-performance querying. It is organized in a way that aligns with business needs, enabling fast access to insights through dashboards, analytical tools, or custom reports.

5.4 Metadata tables

There are three metadata tables that support the automation of the data processing flows: *FLOW MANAGER*, *TABLE MANAGER*, and *METADATA MANAGER*

The FLOW MANAGER table contains information related to the data flow of the different business areas.

The TABLE MANAGER table stores information concerning the management of the tables involved in the process.

The METADATA MANAGER table holds metadata associated with the execution of the various Data Warehouse processes.

5.4.1 FLOW MANAGER

The FLOW MANAGER table contains the following fields:

- `IDENTITY`: Specifies the name of the execution environment in which the process is running
- `NUM_LEVEL`: Indicates the logical level of the data flow within the architecture. Conventionally, 0 corresponds to Level 0 (L0), 1 to Level 1 (L1), and 2 to Level 2 (L2).
- `GRP_NAME`: Represents the functional domain or business area associated with the data flow.
- `TRG_JOBID`: A unique identifier assigned to the job of the current ETL execution, used to track its progress and lineage.
- `SRC_JOBID`: It has no operational meaning, during L0 assumes a default value `19000101000000` and for L1 assumes the value of `TRG_JOBID`
- `STATUS`: Denotes the current state of the process. The possible values are:

  - 0: Successfully completed
  - 1: Execution in progress
  - 3: Terminated with errors.

- `LOAD`: Indicates the availability of data for the subsequent level in the data pipeline. Its values are interpreted as follows:
  - 0: Data has already been consumed
  - 1: Data is ready to be consumed
  - 2: Data is not yet ready for consumption
  - 5: Data cannot be consumed further, as the final logical level has been reached.

- `START_DATE`: Timestamp marking the start of the process execution
- `END_DATE`: Timestamp marking the end of the process execution.

The FLOW MANAGER table adopts a `MERGE` insertion strategy, if a match is found for the keys `IDENTITY, GRP_NAME, LEVEL`, and `TRG_JOBID`, `LOAD`, `STATUS`, `END_DATE` are subject to updates post-insertion.

5.4.2 TABLE MANAGER

The fields of the TABLE MANAGER contain information about:

- `IDENTITY`: Offers the same information as the `IDENTITY` field in the FLOW MANAGER
- `LEVEL`: Identical content as in FLOW MANAGER
- `GRP_NAME`: Presents data equivalent to those of the FLOW MANAGER table.
- `TABLE_NAME`: Name of the table
- `JOBID`: Identifier of the current ingestion cycle, it is equivalent to the field `TRG_JOBID` in the FLOW MANAGER table.
- `NUM_ROWS`: Number of rows read from the table during the current ingestion cycle.
- `LOAD_DATE`: Timestamp describing when the table terminated its loading

The TABLE MANAGER also follows an `APPEND` strategy, whereby a new row is inserted for each table read operation during an ingestion cycle.

5.4.3 METADATA MANAGER

The METADATA MANAGER table does not posses a fixed schema, unlike the previously described metadata table. The only consistently present attributes are `IDENTITY`, `GRP_NAME`, `PARAM_NAME`, and `PARAM_VALUE`. The latter two fields describe variables and parameters essential for pipeline automation. Examples include parameters specifying the number of ingestion cycles after which the `E$` in Layer L1 should be cleared, the maximum number of cycles allowed to persist in the `DLT` table at Level L0, or a list of email addresses to notify in case of pipeline failure. All other attributes are dynamic and may vary depending on the client, the nature of data, and the types of source-extracted tables.

5.5 Scheduling of Layers and Execution Rules

Each layer in the ETL framework is designed to function independently. However, specific coordination rules must be enforced to ensure consistency and integrity across the pipeline:

1. Dimension Processing : Each step related to a dimension (as defined in the star schema model): can proceed independently up to Layer L1, provided that there are no other active processes of the same type.

2. Parallel Execution at L0: All steps operating at Layer L0 can be executed in parallel, including multiple processes of the same type.

3. Exclusive Execution at L1: Before initial processing at Layer L1, each step must ensure it is the only process currently executing, to maintain data consistency during transformation and loading.

4. Prerequisites for L2 Loading: Loading into Layer L2 can only occur once all L1 steps have completed successfully, ensuring coherent and complete input to the data consolidation layer.

The implementation of Mediamente Consulting's framework on Microsoft Fabric will be developed up to the ODS stage within the L1 layer, including the creation of all necessary metadata tables.

Chapter 6

Implementation of the Proposed Solution within the Microsoft Fabric Environment

To automate the ETL process, Microsoft Fabric offers *Data Pipeline* components, which enable the orchestration and scheduling of various activities, by allowing to connect functional blocks with different types of branches:

- *On-Success*: If the current activity is executed successfully, the next one is triggered.
- *On-Failure*: If the current activity raises an exception during its execution, call the next one.
- *On-Completion*: Regardless of whether the current activity succeeds or fails, trigger the following one
- *On-Skip*: If the current activity is skipped, execute the next one.

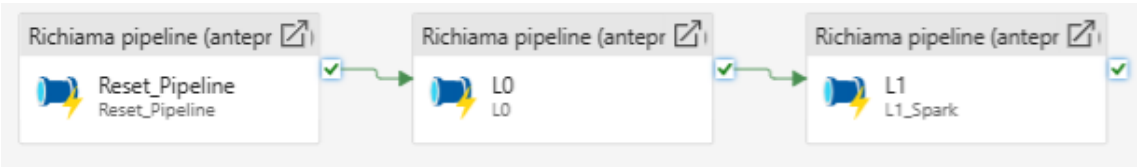


Figure 6.1: Main Pipeline

In this project, the main pipeline is composed of three sub-pipelines connected with on-success branches as 6.1 shows: *Reset_Pipeline*, *L0* and, *L1*. The L1 pipeline

can be implemented using PySpark notebooks, T-SQL stored procedures, or Microsoft Fabric's low-code interface DataFlows Gen2.

All input files utilized by the L0 pipeline are stored in a Lakehouse environment. Similarly, the tables created and populated during the execution of the L0 pipeline are also stored within the same Lakehouse. In contrast, the destination of the tables created by the L1 pipeline will either be stored in the Lakehouse or in a Warehouse, depending on the implementation strategy of L1.

6.1 Reset_Pipeline

The `Reset_Pipeline` is executed once at the beginning to prepare the environment. It performs a cleanup of the workspace by deleting all Delta tables from both the Data Warehouse and the Lakehouse, and by moving any files located in subdirectories to the main Lakehouse directory *Files*. Its execution is controlled by a Lookup activity, and an If-Else conditional block, if the conditional block enters the `True` branch, the `Reset_Pipeline` is executed, otherwise does nothing. The `Reset_Pipeline` is formed by a `Reset_All PySpark` notebook, a `Reset_Warehouse` T-SQL stored procedure and the `Update_Stater` PySpark notebook.

Lookup activity named `Get_Started` is used to read a file named `start.xlsx`, which contains a single column named `started` with one row with a single Boolean value: either `True` or `False`. This value indicates whether it is the first execution in the pipeline's lifetime. If the value is `False`, it means that the pipeline has not yet been executed, otherwise, the value is `True`. This activity outputs a pseudo-JSON text. Then, the JSON output of the Loopkup activity is passed to the If-Else conditional block, named `starter`. If the output is `False`, it triggers the execution of the `Reset_Pipeline` and subsequently the notebook `Update_Starter` updates the same Excel file, overwriting the `started` value with `True`.

The `Reset_All` notebook retrieves the name of all the Delta Lake tables stored in the Lakehouse and then issues `DROP TABLE` commands for each. In the following step, the notebook moves all files from the Lakehouse subfolders to the root Files directory of the Lakehouse. The T-SQL Stored Procedure named `Reset_WH` on the other hand, cleans the Warehouse.

The first activity is connected with an on-success branch to the L0 pipeline, which is responsible for the extraction of data from source and metadata files, also, filters and forwards the relevant records to the L1 pipeline.

6.2 L0

The L0 pipeline is responsible for the Extraction phase of the ETL process. It is implemented through a combination of PySpark notebooks and Microsoft Azure Data Factory pipeline components, including the email notifications and If-Else blocks.

As illustrated in Figure 6.2, the pipeline is initiated by the parallel execution of two notebooks, `Metadata_Creator`, that creates the metadata tables, namely `FLOW MANAGER` and `TABLE MANAGER`, which are essential to register respectively all information related to the flow of data and the tables involved; and, `Create_Schema` which ingests an input `Excel` file which is stored in the Lakehouse main directory; it contains metadata information definitions for all the tables to be processed in the pipeline.

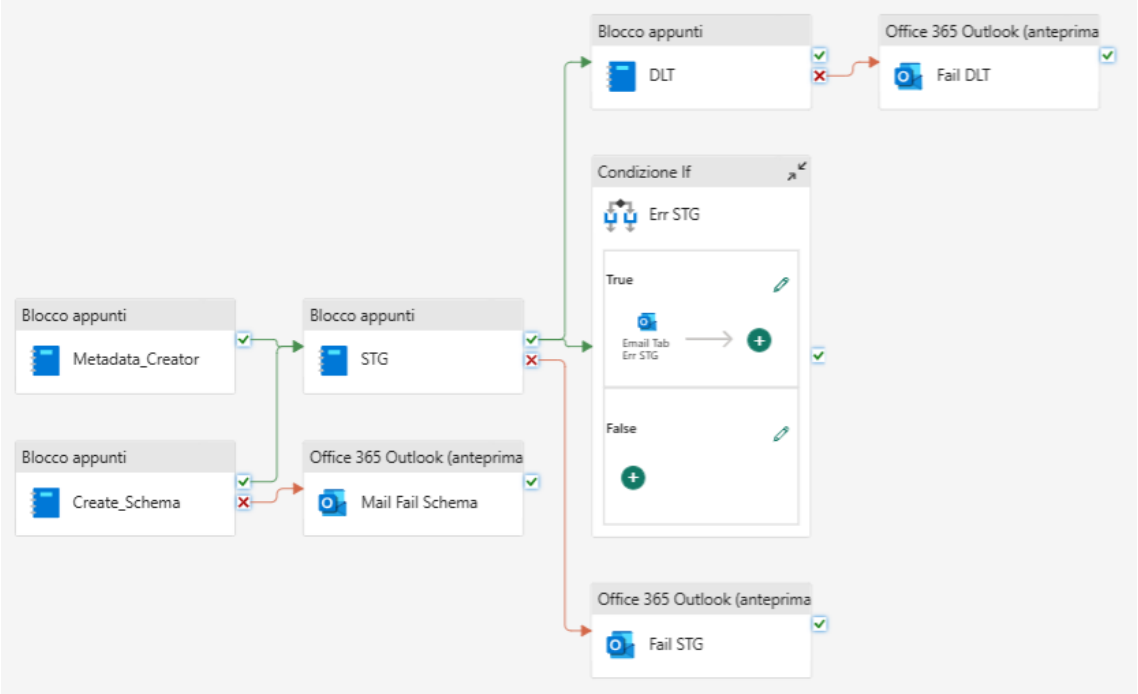


Figure 6.2: L0 Microsoft Fabric Pipeline

If the `Create_Schema` notebook fails, an automated email notification is sent to the designated supervisor of the pipeline. If it completes successfully, the pipeline proceeds with the execution for the `STG` notebook, responsible for reading and staging the source data from `CSV` files. The following conditional logic governs the flow after the `STG` execution: If `STG` fails, an email is sent to notify the failure; if `STG` succeeds, but some `CSV` could not be processed, an email is sent notifying which files failed, then triggers the execution of the `DLT` notebook; last scenario, if all files are processed

successfully, the DLT notebook is executed.

The DLT notebook performs incremental data extraction, identifies, and processes only new, changed, and deleted records to be passed forward to the L1 layer. In cases where certain tables could not be read during this stage, an additional notification is dispatched.

6.2.1 Metadata_Creator

The task of the notebook is to verify if the tables FLOW_MANAGER and TABLE_MANAGER exist within the Lakehouse storage. If they are not found, the notebook proceeds to create them. Table 6.1 and table 6.2 display respectively the schema and header for the FLOW_MANGER, table 6.3 and table 6.4 show the schema and header for the TABLE_MANAGER

Column Name	Data Type	Nullable
IDENTITY	StringType()	False
GRP_NAME	StringType()	False
LEVEL	IntegerType()	False
SRC_JOBID	StringType()	False
TRG_JOBID	StringType()	False
STATUS	IntegerType()	False
LOAD	IntegerType()	False
START_DATE	TimestampType()	False
END_DATE	TimestampType()	True

Table 6.1: FLOW_MANAGER Schema

IDENTITY	GRP_NAME	LEVEL	SRC_JOBID	TRG_JOBID	STATUS	LOAD	START_DATE	END_DATE

Table 6.2: FLOW_MANAGER Header

Column Name	Data Type	Nullable
IDENTITY	StringType()	True
LEVEL	IntegerType()	True
GRP_NAME	StringType()	True
TABLE_NAME	StringType()	True
JOBID	StringType()	True
NUM_ROWS	IntegerType()	True
LOAD_DATE	TimestampType()	True

Table 6.3: TABLE_MANAGER Schema

IDENTITY	LEVEL	GRP_NAME	TABLE_NAME	JOBID	NUM_ROWS	LOAD_DATE

Table 6.4: `TABLE_MANAGER` Header

6.2.2 Create_Schema

As an initial setup, the notebook ensures the creation of three specific subdirectories, if they are not already present. The first, *Dimensions*, contains the source `CSV` files representing the dimension tables; the second, *Movements*, stores the `CSV` files corresponding to the fact tables; and the third, *Metadata*, holds all files related to the metadata necessary for the correct interpretation and processing of the data structures.

Subsequently, a user-defined function named `parse_type(string_parameter)` is introduced, it takes a string and maps it to the corresponding data type recognized by Apache Spark.

The next step involves reading an input Excel file located in the main directory of the Lakehouse. Since Apache Spark does not natively support reading `Excel` files, the `pandas` library is employed to perform this operation.

The file contains tables related to the ETL workflow. Specifically, it provides detailed information for each table, including structural and relational properties essential for the schema construction and data validation. A simplified generic example of the content of this metadata file is presented in Table 6.5

Table	**Identity**	**Grp**	**Column**	**Type**	**PK**	**FK**	**Ref_Table**	**Ref_Col**
table_1	Dim	Tab_1	col_1_1	Int	Y	N		
table_1	Dim	Tab_1	col_1_2	String	N	N		
table_1	Dim	Tab_1	col_1_3	String	N	N		
table_1	Dim	Tab_1	col_1_4	String	N	N		
table_1	Dim	Tab_1	col_1_5	String	N	N		
table_1	Dim	Tab_1	col_1_6	Int	N	N		
table_3	Dim	Tab_3	col_3_1	Int	Y	N		
table_3	Dim	Tab_3	col_3_2	String	N	N		
table_4	Dim	Tab_4	col_4_1	Int	Y	N		
table_4	Dim	Tab_4	col_4_2	String	N	N		
table_4	Dim	Tab_4	col_4_3	Int	N	N		
table_4	Dim	Tab_4	col_4_4	Double	N	N		
table_6	Fact	Tab_6	col_6_1	Int	Y	Y	table_1	col_1_1
table_6	Fact	Tab_6	col_6_2	Int	Y	Y	table_4	col_4_1
table_6	Fact	Tab_6	col_6_4	Int	Y	Y	table_3	col_3_1
table_6	Fact	Tab_6	col_6_5	Int	N	N		
table_6	Fact	Tab_6	col_6_6	Double	N	N		

Table 6.5: Sample content of the `Excel` metadata file

It includes several key fields:

- Table: Specifies the name of each table within the data model.
- Identity: Indicates whether the table is a fact or a dimension table.
- Grp: Defines the logical group to which the table belongs, multiple tables can belong to the same group, but a table can belong to only one group.
- Column: Specifies the name of each attribute associated with the table.
- Type: Defines the corresponding data type.
- PK: Indicates whether a column serves as a part of the primary key, the presence of multiple columns marked with "Y" (Yes) for the same table denotes a composite primary key.
- FK, Ref_Table and Ref_Col: FK identifies whether a column is a foreign key, and if so, the Ref_Table and Ref_Col fields specify the referenced table and the column to reference, respectively, thus enabling the definition of inter-table relationships.

Additionally, the `Excel` metadata file may include further columns that define data quality constraints.

The `Excel` file is initially read using the `pandas.read_excel()` function, then a set of transformations is applied to the resulting `Pandas` DataFrame: binary columns with "Y"/"N" values are mapped to Boolean `True`/`False`, and the values "Fact" and "Dim" in the `Identity` column are standardized to "FACT" and "DIMENSION", respectively. Since the metadata is required throughout the entire ETL process, the transformed **Pandas** DataFrame is subsequently cast to an Apache Spark DataFrame and stored in the Lakehouse storage as a Delta Table named `Metadata_Schema`

The next step in the notebook involves generating the schema for the staging (`STG`) tables by reading the `Metadata_Schema` Delta Table. For each unique table name listed in `Metadata_Schema`, the corresponding column names and data types are extracted and used to construct a schema for a Spark DataFrame. This schema serves to define the structure of an empty STG table, also, `JOBID` and `INS_DATE` columns are added to the schema. Subsequently, the process checks whether an STG table with the same schema already exists. If it does, the algorithm compares the existing schema with the newly generated one. In the event of a mismatch, the old schema is overwritten with the updated schema, and default values are assigned to existing records for any newly added columns.

At the end of this process, an STG table is created for each distinct table defined in the `Metadata_Schema` Delta Table.

Finally, for each table, the algorithm moves the corresponding source `csv` file from the Lakehouse's main directory to the appropriate subdirectory depending on the `Identity` value for such table. If the table is labeled as a dimension by the `Identity` column, then the `csv` file is moved to the *Dimensions* subdirectory, otherwise, it is moved to the *Movements* subdirectory.

Algorithm 1 Pseudo-code for the Create_Schema notebook

```
file_dim_read ← [ ]
file_fact_read ← [ ]
df_schema ← read_excel(metadata_schema_file)
for table_name in df_schema["Table"] do
    fields ← []
    subset ← df_schema[df_schema["Table"] == table_name]
    for row in subset.iterrows() do
        col_name ← row["Column"]
        fields.append(StructField(col_name, parse_dtype(row["Type"]))
    end for
    fields.append(StructField("JOBID", StringType())
    fields.append(StructField("INS_TIME", TimestampType())
    schema ← StructType(fields)
    if table_name_STG exists in Lakehouse then
        if old_schema = schema then
            continue
        else
            overwrite_(table_name)_STG_old_schema(schema)
        end if
    else
        create_STG_Table(schema)
    end if
    identity ← subset["Identity"]
    if identity = 'Dimension' then
        move_(table_name).csv_to_Dimensions_folder
        file_dim_read.append(table_name)
    else
        move_(table_name).csv_to_Facts_folder
        file_fact_read.append(table_name)
    end if
end for
```

The notebook exports the arrays of dimensions and facts to the STG notebook.

6.2.3 STG

Firstly, the notebook retrieves the output produced by the Create_Schema notebook. Then, the notebook imports the utility functions from an auxiliary module, METADATA_TABLES_HANDLER, designed to manage operational metadata, it contains two functions write_flow_manager and write_table_manager which write on the FLOW_MANAGER and TABLE_MANAGER tables respectively.

write_flow_manager accepts the following parameters: identity, grp_name, level, src_jobid, trg_jobid, status, load, start_date, end_date. It performs a MERGE operation on the FLOW_MANAGER table using a composite key consisting of identity, grp_name, level and trg_jobid. If a record with matching keys exists, the field status, load, and end_date are updated. If no matching records exists, a new row is inserted. The end_date is left None, indicating that processing for the specified level, target job and group name has just commenced.

Unlike write_flow_manager, write_table_manager function follows an APPEND insertion mode. Each invocation adds a new record to the TABLE_MANAGER. It takes the following input parameters: identity, level, grp_name, table_name, jobid, num_records, insert_time, none of them are optional.

It is important to recall that the FLOW_MANAGER records information related to the executions within a single run of the ETL process. Specifically, it tracks the progress at the flow level, for each logical group of tables. The TABLE_MANAGER, on the other hand, stores metadata related to the individual tables being processed during the execution of the ETL process.

After importing the utility functions, the STG functions computes the JOBID corresponding to the current ETL execution and defines a function named load_STG, which takes as input a string representing the name of a CSV file to be loaded into its corresponding STG table. For each input file, the function extracts the table name and retrieves the associated GRP_NAME and IDENTITY from the Metadata_Schema Delta Table. It also queries the FLOW_MANAGER table to obtain the most recent STATUS for the logical group (GRP_NAME) to which the table belongs, filtered by LEVEL = 0 and the current TRG_JOBID. If a status is found and its value is equal to -3, an exception is raised. This mechanism is designed to handle cases in which multiple tables share the same logical group: if an error occurs during the loading of one table, all other tables belonging to the same group are skipped to prevent inconsistent group-level processing. Subsequently, a new record is inserted into the FLOW_MANAGER table. The function write_flow_manager is invoked with the following parameters: IDENTITY, GRP_NAME, LEVEL = 0, a default source job identifier SRC_JOBID = 19000101000000, the current job identifier (TRG_JOBID = JOBID), LOAD = 1, STATUS = 2, the current timestamp as START_DATE and None as END_DATE. This insertion serves to indicate the start of the data loading process from the specified CSV file into the corresponding

STG table. Next, the CSV file is read, the previously calculated JOBID and the current timestamp are appended to the extracted data, and finally the data is loaded with an append strategy into the corresponding STG Delta table; and a new record is inserted into the TABLE_MANAGER to log the recently loaded table, including the number of records read during the loading process.

If an exception or error occurs during the loading process, the file name and the corresponding group name are record in dedicated arrays. Additionally, the FLOW_MANAGER is updated by setting the STATUS to -3 for the specified IDENTITY, GRP_NAME, TRG_JOBID, and LEVEL = 0. Importantly, despite the occurrence of errors, the overall ETL process continues execution and is never halted.

The STG notebook invokes the load_STG function for each table to be loaded into its respective STG table. Upon completion, only those logical groups that did not raise exceptions during the loading phase proceed further through the pipeline, other than this, the output of the notebook also includes the associated tables and identities the job identifier (JOBID) of the current execution flow and the table names and associated groups that raised exceptions.

Algorithms 2-4 provide an overview of the core functions involved in the data ingestion process: algorithm 2 presents the pseudocode describing the logic behind the MERGE operation performed within the write_flow_manager function; algorithm 3 details the pseudocode for the corresponding INSERT operation used in the write_table_manager function; finally, algorithm 3 illustrates the complete logic for loading data from CSV files into the corresponding STG Delta tables.

Algorithm 2 Write flow manager function

```
1: function WRITE_FLOW_MANAGER(identity, grp_name, level, src, trg, status, load,
   start_date, end_date)
2:    merge delta_table (alias "t") with source (alias "s") on:
3:    t.IDENTITY = s.identity and
4:    t.GRP_NAME = s.grp_name and
5:    t.LEVEL = s.level and
6:    t.TRG_JOBID = s.trg
7:    if matched and s.END_DATE is not null then update:
8:    STATUS = s.status
9:    LOAD = s.load
10:   LEVEL = s.load
11:   SRC_JOBID = s.src
12:   END_DATE = s.end_date
13:   if not matched then insert identity, grp_name, level, src, trg, status,
   load, start_date, end_date
14:   execute merge
15: end function
```

Algorithm 3 Write table manager record

```
function WRITE_TABLE_MANAGER(identity, level, grp_name, table_name, jobid, num_rows, load_date)
   merge delta_table (alias "t") with new_row (alias "s") on:
   t.IDENTITY = s.identity and
   t.LEVEL = s.level and
   t.GRP_NAME = s.grp and
   t.TABLE_NAME = s.table_name and
   t.JOBID = s.jobid
   if matched then update identity, grp_name, level, src, trg, status, load, start_date, end_date
   if not matched then insert identity, grp_name, level, src, trg, status, load, start_date, end_date
   execute merge
end function
```

Algorithm 4 Load CSV file into STG table

```
function LOAD_STG(csv_file, no_valid_grp, error_files)
   table_name ← extract name from csv_file (without extension)
   grp_identity ← run query on Metadata_Schema to get GRP_NAME, IDENTITY where TABLE_NAME = table_name
   identity ← grp_identity["IDENTITY£]
   grp ← grp_identity["GRP_NAME"]
   status ← run query on FLOW_MANAGER to get STATUS where GRP_NAME = grp, LEVEL = 0, TRG_JOBID = jobid
   if status exists and status = -3 then
      raise exception
   end if
   write_flow_manager(identity, grp, 0, '19000101000000', jobid, 1, 2, current_time, None)
   if identity = "DIMENSION" then
      df ← read csv_file from Dimensions folder
   else
      df ← read csv_file from Movements folder
   end if
   add columns JOBID = jobid and INS_TIME = current_time to df
   write df in delta format with append mode to table table_name_STG
   write_table_manager(identity, 0, grp, table_name_STG, jobid, count_rows(df), current_time)
   return no_valid_grp, error_files
   Catch exception e
       append csv_file to error_files
       append grp to no_valid_grp
          write_flow_manager(identity, grp, 0, '19000101000000', jobid, -3, 2, None, current_time)
end function
```

6.2.4 DLT

Like the STG notebook, the DLT notebook receives as input the the output produced by its predecessor, specifically it considers only the table names and related groups that were loaded correctly in the STG tables, the notebook also imports the METADATA_TABLES_HANDLER module. It then initiates the extraction of incremental data, this is achieved by selecting only rows whose JOBID is greater than the most recent successfully processed JOBID for that table. The LAST_JOBID computation and record extraction can be explained easily with SQL syntax.

```
SELECT LAST_JOBID = ISNULL(MAX(JOBID), 19000101000000)
FROM TABLE_MANAGER TM, FLOW_MANAGER FM
WHERE JOIN_CONDITION
      AND FM.STATUS = 0
      AND FM.LIVELLO = 0
      AND TM.TABLE_NAME = nome_tabella

SELECT * AS NON_PROCESSED_DATA
FROM Table_STG
WHERE JOBID > LAST_JOBID

SELECT * AS LAST_PROCESSED_DATA
FROM Table_DLT
WHERE JOBID = LAST_JOBID
```

These conceptual queries were translated into the Apache Spark environment. The first MINUS NON_PROCESSED_DATA - LAST_PROCESSED_DATA was used to retrieve only the new records that have appeared in the STG table since the last successful Level 0 (L0) load that specific table. These new records are tagged with a new column named FLG_NEG, set to 0, indicating they represent new inserted data; the second MINUS operation, LAST_PROCESSED_DATA - NON_PROCESSED_DATA serves to identify records that were present in the previous successful load but are no longer available in the source. These are treated as physical deletions, and each record is assigned FLG_NEG = 1.

The resulting record from both MINUS operations are unified into a single dataset, assigned the current JOBID, and then written to the DLT table. The DLT table is append-only, however, to manage storage efficiently, older records in the DLT are periodically deleted after several ETL executions. For this reason, a second table named DLT_HIS exists. It contains the same records as the DLT, with the key distinction that no data is ever removed from it.

Algorithm 5 is executed for each table listed in the output of the STG notebook.

Algorithm 5 DLT Data Processing and Minus Logic

```
function PROCESS_DLT(table_name, identity, grp_name, no_valid_grp, jobid, error_files)
    Try
    if The Status indicated by the FLOW_MANAGER for the GRP_NAME on Level 0 and on the current JOBID == -3 then
        Skip the execution for this table
    end if
    joined ← join FLOW_MANAGER and TABLE_MANAGER on IDENTITA, GRP_NAME and JOBID
    filter joined where LEVEL = 0, STATUS = 0, TABLE_NAME = table_name
    result ← COALESCE(MAX(JOBID), '19000101000000') from joined
    NON_PROCESSED_DATA ← SELECT * FROM table_name_STG WHERE JOBID > result
    LAST_PROCESSED_DATA ← SELECT * FROM table_name_STG WHERE JOBID = result
    if result = '19000101000000' then
        new ← execute NON_PROCESSED_DATA and drop JOBID, INS_TIME
        old ← empty DataFrame with same schema
    else
        old ← execute LAST_PROCESSED_DATA and drop JOBID, INS_TIME
        new ← execute NON_PROCESSED_DATA and drop JOBID, INS_TIME
    end if
    new_minus_old ← new MINUS old
    old_minus_new ← old MINUS new
    Add columns JOBID = jobid, INS_TIME = now, FLG_NEG = 0 to new_minus_old
    Add columns JOBID = jobid, INS_TIME = now, FLG_NEG = 1 to old_minus_new
    dlt ← UNION of new_minus_old and old_minus_new
    write dlt to table_name_DLT in append mode
    write dlt to table_name_DLT_HIS in append mode
    write_table_manager(identity, 0, grp, table_name_DLT, jobid, count_rows(dlt), current_time)
    write_flow_manager(identity, grp_name, 0, '19000101000000', jobid, 0, 0, None, current_time)
    return no_valid_grp, error_files
    Catch exception e
        append table_name to error_files
        append grp to no_valid_grp
         write_flow_manager(identity, grp_name, 0, '19000101000000', jobid, -3, 2, None, current_time)
end function
```

The DLT table retains only the most recent records defined by the `retention_time` parameter. Specifically, it holds the data corresponding to the latest `retention_time` distinct JOBID values, as retrieved from the FLOW_MANAGER for the logical group (GRP_NAME) associated with the table. Algorithm 6 shows the pseudo-code of the previous logic

As in the STG phase, the TABLE_MANAGER is updated by inserting a new row where the table name is Table_Name_DLT, along with the number of rows written to the DLT. If any of the DLT operations described above raise an exception, the affected table

name and associated logical group are recorded. Additionally, the `FLOW_MANAGER` is updated for the group by setting the `STATUS` to `-3` and writing the `END_DATE`, signaling an error in the process for that specific load attempt. For the groups that were successfully loaded into both the `STG` and `DLT` tables, their corresponding records in the `FLOW_MANAGER` associated with the current processing level (0) and `JOBID` are updated accordingly. The `STATUS` field is set to `0`, indicating a successful execution, and both the `LOAD_DATE` and `END_DATE` fields are recorded to reflect the completion of the process.

Algorithm 6 Extract Retained JOBIDs

```
1: Inputs of the code: identity, grp, table_name, retention_time
2: Select distinct TRG_JOBID from FLOW_MANAGER as FM and
   TABLE_MANAGER as TM
3: Where:
4:    Join Conditions
5:    AND FM.IDENTITA == input identity
6:    AND FM.GRP_NAME == input grp
7:    AND TM.TABLE_NAME == input table_name
8: Order results by TRG_JOBID in descending order
9: Limit the result to retention_time rows
10: Filter the DLT table where JOBID is in the list
11: Store the result in a temporary DeltaTable DLT_TMP
12: Save DLT_TMP as a new table named <table_name>_DLT_TMP
13: Drop the original table <table_name>_DLT
14: Rename <table_name>_DLT_TMP to <table_name>_DLT
```

The output of this notebook consists solely of two elements: the `JOBID` and the names of the tables that raised exceptions during execution. The `JOBID` serves as an essential reference for the next logical level of the ETL pipeline, Level 1 (L1). On the other hand, the list of names of tables that failed is sent to the If-Else conditional block connected on success to the DLT notebook as seen in Figure 6.2, if this list is not empty, then an email is sent to the pipeline supervisor notifying the failures. It is of no interest to send other types of information to the next level, since each level operates independently. Therefore, groups that failed during L0 do not interfere with the execution of L1.

Since L0 and L1 are implemented as separate pipelines connected on success -as illustrated in Figure 6.1- the output of the `DLT` process is stored in an `Excel` file. This file is then read during the execution of the L1 pipeline.

According to the theoretical definition of the company's ETL framework, configuration elements such as the `retention_time` parameter and the list of supervisor

email addresses should be stored within the `METADATA_MANAGER`. However, Microsoft Fabric provides an alternative through its built-in library of variables, which can be used to store and manage such configuration information in a centralized and reusable manner across different pipelines and notebooks.

6.3 L1

The pipeline is responsible for transforming the incremental data extracted from the Level 0 layer to ensure compliance with relational and data quality constraints. Three different implementation approaches were developed: The first uses Spark Notebooks to perform transformations, with the resulting Level 1 (L1) tables stored in the Lakehouse; the second relies on several T-SQL stores procedures orchestrated with the help of pipeline activities, in this case, the related tables will be stored inside the Data Warehouse; the third adopts a more user-friendly, low-code approach using Dataflows Gen2, which provides a graphical interface for defining transformation logic, making it accessible to users with limited coding experience.

6.3.1 First solution: Spark

This solution reduces the number of components within the pipeline by consolidating all operations into a single notebook `PRE_LOAD, OK, ERR, ODS`. This design choice aligns with the scheduling and execution conditions defined by the company's framework: as cited in section 5.5 «at Layer L1, each step must ensure it is the only process currently executing». Specifically, once the processing of a given table begins, all the other tables must wait until that process completes - whether successfully or with failure - before initiating their own execution.

As 6.3 demonstrates, the first activity to be executed when the L1 pipeline is running is the Lookup activity `Output_L0`, which reads an Excel file stored in the Lakehouse. This file contains the output generated by the `DTL` notebook during the execution of Level 0 and includes the current `JOBID` of the flow and the list of table names - sourced from the `Metadata_Schema` Delta table - that must be processed in Level 1. All listed tables are processed in accordance with the framework's rule that mandates full independence between levels. This means that even if certain tables could not be processed or encountered errors during the Level 0 execution, they are not excluded from the processing in Level 1. The Lookup activity is connected on-success to the notebook. Additionally, in the event of a complete notebook failure or crash, a separate email is sent. The `PRE_LOAD, OK, ERR, ODS` notebook contains distinct functions that contribute to the overall data quality and consistency process. The `PRE_LOAD` function is responsible for ensuring that referential integrity constraints between tables are respected before any further processing, if the constraints is not respected, acts accordingly. The `OK` function filters out rows from each table that

violate specific data quality constraints, such as null or negative values in mandatory columns. Rows that meet the criteria are stored in the corresponding `OK` Delta table, while those that do not are redirected to the `ERR`. Finally, the `ODS` function collects and stores the validated data from the `OK` tables, preparing it to be passed on the next processing level.

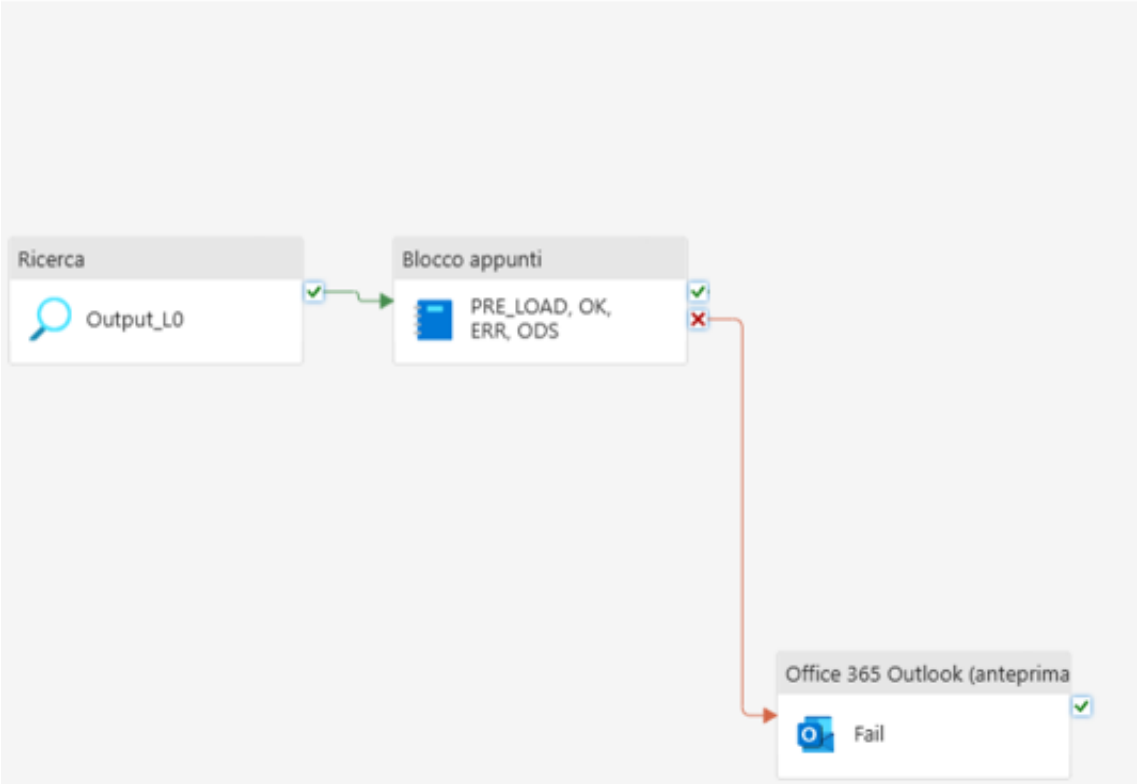


Figure 6.3: Spark implementation of the L1 pipeline

PRE_LOAD, OK, ERR, ODS

As in the notebooks of the L0 layer, this one also imports the `write_flow_manager` and `write_table_manager` functions from the `METADATA_TABLES_HANDLER` notebook. Then establishes an execution sequence for the input tables based on their dependencies. The processing order prioritizes: first, dimension tables with no references to other dimensions; second, dimension tables that reference other dimensions; and finally, fact tables. The logic is as follows: From the `Metadata_Schema` Delta table, for each table name, the referenced tables are grouped, producing a DataFrame with the structure `source table name`, `referenced table name`. Then checks, if for a given source table, references exists and the source table is a dimension, then it is labeled as a Dimension with references, if it is a fact, then it labeled as a Fact; if the given source table is a dimension, but there are no referenced tables, it is labeled as a Dimension without references.

This reasoning can be easily illustrated by algorithm 7

Algorithm 7 Define table execution sequence

```
Inputs of the code: tables_to_process
dimensions ← [], dim_to_dim ← [], facts ← []
Select TABLE_NAME, IDENTITY from Metadata_Schema
Where TABLE_NAME is in tables_to_process
Group by TABLE_NAME, IDENTITY, collect the list of referenced
tables as REFERENCED_TABLES
Store the result in a DataFrame called Metadata_Tables
for row in Metadata_Tables do
    table_name ← row["TABLE_NAME"]
    identity ← row["IDENTITY"]
    referenced_tables ← row["REFERENCED_TABLES"]
    if exists referenced_tables then
        if identity is Dimension then
            dim_to_dim.append(table_name)
        else
            facts.append(table_name)
        end if
    else
        dimensions.append(table_name)
    end if
end for
tables ← dimensions + dim_to_dim + facts
```

After this classification, the notebook defines three auxiliary functions that support the data processing logic. The first function, `check_status`, retrieves the current status of a given logical group (`grp_name`) and `JOBID` from the `FLOW_MANAGER` table at Level 1, returns `True` if the `STATUS` is `-3`. This is to avoid further processing of groups that failed in previous steps.

The second function, `row_number`, identifies the most recent successful `JOBID` for a given identity and group name in Level 1 by querying the `FLOW_MANAGER`. It then filters the rows in `<Table>_DLT` to include only those with a `JOBID` greater than the last successful one; this ensures that only records which have not yet been processed in Level 1 are considered. These filtered rows are combined with the existing rows in `<Table>_ERR` using `unionByName`, the resulting DataFrame undergoes a `Row Number` operation. This operation partitions the data by the table's primary keys and orders the rows with by `JOBID` in descending order and `FLG_NEG` in ascending order. Finally, only the rows with `ROW_NUMBER = 1` are selected and returned, ensuring that the most recent and valid entry for each key is kept. The pseudocode in Algorithm 8

clearly illustrates the logic implemented in the `row_number` function.

Algorithm 8 `ROW_NUMBER function`

```
function ROW_NUMBER(table_name, df_metadata_schema_filtered_by_table, identity,
grp_name, jobid)
   Try:
   LAST_SUCC_JOBID ← from COLAESCE(MAX(JOBID), '19000101000000'), from
FLOW_MANAGER, where LEVEL == 1 and STATUS == 0 and GRP_NAME == grp_name and
IDENTITY == identity
   DLT_DF ← select * from <table_name>_DLT where JOBID > LAST_SUCC_JOBID
   ERR_DF ← read_Delta_Table(<table_name>_ERR).select(DLT_DF.columns)
   UNION_DF ← DLT_DF unionByName ERR_DF
   primary_keys ← select COLUMN_NAMES as PK, from
df_metadata_schema_filtered_by_table, where PrimaryKey == True
   (select * over ROW_NUMBER(Partition By PK, Order by JOBID DESC, FLG_NEG
ASC) AS RN) AS Temp
   return Temp.filter(RN == 1)
   Catch exception e
       return -1
end function
```

Finally, the third function, `buildDF`, takes as input a table schema and a list of primary keys from another table, then constructs a new schema that includes: all columns present in both the input schema and the primary keys, standard technical columns such as `JOBID`, `INS_TIME`, and `FLG_NEG`, the rest of columns present in the input schema are set to `null`.

PRE_LOAD

To verify referential integrity constraints between tables, the typical approach involves performing JOIN operations. However, it is well known that `JOINs` can significantly impact performance, especially on large datasets. For this reason, a different strategy is adopted in the implementation, the pre-load operation: consist in extracting values from the foreign key column(s) of the source table (in this case, `<table>_DLT`) that are not present in the foreign key column(s) of the referenced `<table>_ODS`, instead of discarding these missing values, they are inserted into the foreign key columns(s) of the destination table (`<table>_PRE_LOAD`). During this step, all technical fields are populated - specifically, `JOBID`, `INS_TIME` and `FLG_NEG` - while the remaining fields are filled with default values.

For a given source table - obtained through the `row_number` function described earlier - the referential integrity check begins by identifying all the destination tables that it references. For each referenced destination, the `Metadata_Schema` Delta Table allows the extraction of the foreign key relationships: specifically, the columns in the source table that perform the referencing and the corresponding columns in the destination table being referenced. This information is assembled

into a DataFrame named `fks`. Then, the function iterates through each row of `fks`. For each referenced table, it retrieves the corresponding `GRP_NAME` and `IDENTITY` from the `Metadata_Schema`. If the destination table exists, the process continues by collecting the distinct key values from both the source (`source_PK_Values`) and destination (`dest_PK_Values`) tables. At this point, two scenarios are handled based on the value of a control flag passed to the `pre_load` function:

- **Dimension-to-Dimension Check (`flag = 1`):** This scenario validates the referential integrity between one dimension table and another. A set difference operation (`dest_PK_Values MINUS source_PK_Values`) is performed to identify keys that are present in the destination but not yet in the source. If the result is empty, the function is interrupted early. Otherwise, the `buildDF` function is used to construct a new DataFrame using the schema and the primary keys of the source. This DataFrame, which contains the key columns, the technical columns (`JOBID`, `INS_TIME`, and `FLG_NEG`) and default values for the non-key attributes, is inserted into the `<source>_PRE_LOAD` table.
- **Fact-to-Dimension Check (`flag = 0`):** This scenario verifies the integrity between a fact table and its associated dimensions. Here, the set difference operation is inverted: `source_PK_Values MINUS dest_PK_Values` is used to find keys present in the fact table that are missing in the referenced dimension. Again, the `buildDF` function is used with the schema of the dimension and the primary key values from the fact. The resulting DataFrame is inserted into the `<destination (dimension)>_PRE_LOAD` table. Additionally, two technical columns are appended to the DataFrame: `UPD_JOBID`, populated with the current execution identifier, and `UPD_TIME`, set to the current system timestamp, the newly modified DataFrame is inserted into the `<destination (dimension)>_ODS` table.

After each insertion, the `write_table_manager` function is called to log the operation into the `TABLE_MANAGER`, recording the number of rows inserted into the referenced table. If an error occurs at any point during the function's execution, the `GRP` and table name passed as input are logged, and a failure status code `-3` is recorded in the `FLOW_MANAGER` for level `L1`, using the current `JOBID`, source `grp_name` and `identity`.

The `PRE_LOAD` table for a given table, uses a `DELETE-INSERT` insertion mode, it means that on each execution, all existing records in the `PRE_LOAD` table are deleted and the table is entirely repopulated with the rows form the current data flow.

The `OK` and `ERR` table, which will be introduced in the following sub-section, also operate using a `DELETE-INSERT` insertion mode, unlike the tables created and populated in Level 0 (`STG`, `DLT`, `DLT_HIS`), that follow an `APPEND` insertion mode

The pseudocode describing the previous logic is implemented through algorithm 9

Algorithm 9 `pre_load function`

```
1: function PRE_LOAD(table, metadata_schema, jobid, grp_name, identity, valid_files, error_files,
   no_valid_grp, flag_dim_dim)
2:    Try:
3:    metadata_filtered ← metadata_schema.filter(TABLE_NAME == table)
4:    source_df ← row_number(table, metadata_filtered, identit, grp_name, jobid)
5:    if source_df == None then
6:       return error_files, no_valid_grp, valid_files
7:    else if source_df == -1 then
8:       raise An error verified during the row_number function
9:    end if
10:    fks ← select * from Metadata_Schema, where TABLE_NAME == table and FK == True
11:    fks ← select REFERENCED_TABLE, list of referenced column names, list of source column names
   from fks, groupBy REFERENCED_TABLE
12:    for each fk_row in fks do
13:       dest_table ← fk_row["REFERENCED_TABLE"]
14:       grp_ref, identity_ref ← select GRP_NAME, IDENTITY from Metadata_Schema where TABLE_NAME =
   dest_table
15:       if dest_table_ODS exists then
16:          dest_df ← read(dest_table_ODS)
17:       else
18:          skip
19:       end if
20:       source_cols ← fk_row[list of source column names]
21:       dest_cols ← fk_row[list of referenced column names]
22:       source_PK_Values ← source_df.select(source_cols)
23:       dest_PK_Values ← dest_df.select(dest_cols)
24:       if flag_dim_dim == 1 then
25:          missing_keys ← dest_PK_Values MINUS source_PK_Values
26:          if missing_keys is empty then
27:             skip
28:          end if
29:          preload_schema ← schema of source table_PRE_LOAD
30:          columns ← buildDF(preload_schema, source_cols)
31:          preload_df ← missing_keys.select(columns)
32:          insert preload_df into source table_PRE_LOAD
33:       else
34:          missing_keys ← source_PK_Values MINUS dest_PK_Values
35:          if missing_keys is empty then
36:             s
37:          end if
38:          columns ← buildDF(schema of dest_df, source_cols)
39:          preload_df ← missing_keys.select(columns)
40:          insert preload_df into dest_table_PRE_LOAD
41:          ods_df ← preload_df.withColumn(UPD_JOBID = jobid, UPD_TIME = sysdate)
42:          append ods_df to dest_table_ODS
43:       end if
44:       write_table_manager(identity_ref, 1, grp_ref, dest_table_PRE_LOAD, jobid,
   missing_keys.count(), sysdate)
45:       valid_files.append(table)
46:    end for
47:    return error_files, no_valid_grp, valid_files
48:    Catch exception e
49:       error_files.append(table)
50:       no_valid_grp.append(grp_name)
51:       write_flow_manager(identita, grp_name, 1, jobid, jobid, -3, 2, sysdate, sysdate)
52: end function
```

OK

The `OK` function is responsible for cleaning tables from records that do not meet the quality constraints. For this project, the relevant constraints are : Not-nullable columns: Records containing null values in mandatory fields are discarded; Numeric Columns: Negative values are not allowed an result in the record being discarded

The records that respect the constraints will be stored in an `OK` Delta table, while those that rejected will be stored in the corresponding `ERR` Delta table.

The data quality validation process begins by querying the `Metadata_Schema` associated with a given table to identify the column that must not contain `NULL` values and the columns that must not contain negative values.

Subsequently, the dataset to be validated is obtained via the `row_number`. Initially, a flag is assigned to all records that contain `NULL` values in any of the previously identified non-nullable columns. These flagged records are isolated and appended to the corresponding error table (`<table>_ERR`). In addition to these records, an extra column (`Descr_Err`) is added to describe the reason for exclusion. The remaining records, which passed the null-check phase, are further evaluated for non-negativity constraints. Records that contain negative values in columns that must only accept non-negative inputs are similarly flagged and redirected to the `<table>_ERR` table, also with an accompanying descriptive message.

At the end of the validation process, two datasets are produced, `<table>_OK` containing only the records that fully satisfy all quality constraints and `<table>_ERR` that collects invalid records along with diagnostic information about the constraint(s) they violated.

Finally, a log entry is created in the `TABLE_MANAGER` for the `<table>_OK` table, indicating the number of successfully loaded rows.

In the event of an exception during processing, the system captures the name of the affected table and its associated group. The `FLOW_MANAGER` is updated, an entry is recorded with a status code of `-3` and the end date for the given group name, `jobid` and level 1.

Algorithm 10 shows the pseudocode for the `OK` function.

Algorithm 10 `ok function`

```
function ok(table, jobid, grp_name, identity, metadata_filtered_by_table, valid_files, error_files, no_valid_grp)
   notnull_cols ← select column_name from metadata_filtered_by_table where Nullable == False
   gt0_cols ← select column_name from metadata_filtered_by_table where less_than_zero == False and Type in (INT, DOUBLE)
   row_number_df ← row_number(table, metadata_filtered_by_table, identity, grp_name, jobid)
   if row_number_df == None then
      return
   end if
   if row_number_df == -1 then
      raise An error occurred during the row_number function
   end if
   cnull_df ← row_number_df with column contains_null flagged to 1 where null values appear in notnull_cols
   clean_1_df ← select * from cnull_df where contains_null == 0
   enull_df ← select *, "contains null value" as Descr_Error from cnull_df where contains_null == 1
   cgt_df ← clean_1_df with column contains_neg flagged to 1 where negative values appear in gt0_cols
   clean_2_df ← select * from cgt_df where contains_neg == 0
   egt_df ← select *, "contains negative value" as Descr_Error from cgt_df where contains_neg == 1
   temp_err_df ← enull_df union by name egt_df
   Store temp_err_df as <table>_ERR Delta Table
   clean_2_df ← clean_2_df.withColumn("JOBID", jobid).withColumn("INS_TIME", sysdate)
   Store clean_2_df as <table>_OK Delta Table
   write_table_manager(identity, 1, grp_name, <table>_OK, jobid, num_rows, datetime.now())
   return
   Catch exception e:
      error_files.append(table)
      no_valid_grp.append(grp_name)
                write_flow_manager(identity, grp_name, 1, jobid, jobid, -3, 2, datetime.now(), datetime.now())
end function
```

MERGE

The merge function is responsible for synchronizing the computed data into the ODS table. Specifically, if a new record with the same keys already exists in the ODS table, it is updated; otherwise, a new record is inserted. The function begins by constructing the merge condition, which ensures that the keys from the input DataFrame match the corresponding keys in the ODS table. Next, the columns to be updated are defined: these include all columns from the union of the `OK` and `PRE_LOAD` DataFrames, as well as the additional columns `UPD_JOBID` and `UPD_TIME`.

Notably, the columns `JOBID` and `INS_TIME` are excluded from the update clause. The insert clause includes all columns from the computed DataFrame, along with `UPD_JOBID` and `UPD_TIME`. The final merge operation is then performed from the computed DataFrame into the ODS table. This operation leverages the merge capability provided by Delta Lake, which allows for atomic operations based on user-defined matching conditions. It requires specification of the keys to match on, the columns to update when a match is found, and the columns to insert when no match is present.

An additional benefit of this approach is that the `ODS` table is not rewritten in its entirety, instead, only the affected rows are updated or inserted, ensuring better performance and transactional integrity.

Algorithm 11 `merge` function

```
1: function MERGE(table_ODS, table, keys)
2:     merge_condition ← keys
3:     columns_to_update_set ← columns from table except JOBID and INS_TIME
4:     columns_to_update_set["UPD_JOBID"] ← table["JOBID"]
5:     columns_to_update_set["UPD_TIME"] ← sysdate
6:     columns_to_insert_set ← all columns from table
7:     columns_to_insert_set["UPD_JOBID"] ← table["JOBID"]
8:     columns_to_insert_set["UPD_TIME"] ← sysdate
9:     table_ODS.merge(table, merge_condition)
10:      .whenMatchUpdate(columns_to_update_set)
11:      .whenNotMatchedInsert(columns_to_insert_set)
12: end function
```

ODS

Finally, in the last stage of level L1, the ODS function processes the data by taking the contents of the `PRE_LOAD` and OK tables and performing a union by name operation. During this process, two additional columns are added: `UPD_JOBID`, which records the current `JOBID` value, and `UPD_TIME`, which captures the system date of the update. The purpose of this step is to merge the newly computed DataFrame into the `ODS` table. To accomplish this, the key columns required for the merge are extracted from the `Metadata_Schema` and are passed, along with the ODS table and the DataFrame, to the `merge` function.

Since this represents the final stage of the L1 processing pipeline, it is possible at this point to determine whether a logical group has successfully completed all operations defined for level 1. If no errors or exceptions are encountered during the creation and population of the tables described in the previous section - namely, the `PRE_LOAD`, `OK`, and `ERR` - then the corresponding record in the `FLOW_MANAGER` table is

updated. Specifically, for the given logical group at level 1 and for the current `JOBID`, the `STATUS`, and `LOAD` fields are set to 0, and the end date is recorded. This update serves as confirmation that the group has successfully completed all L1 processing steps.

As with the previous function, if an error occurrs during execution, the corresponding logical group name and table name are recorded. Additionally, the `FLOW_MANAGER` table is updated, for the specified `grp_name`, `JOBID`, and level 1, the `STATUS` field is set to `-3` and the end date is inserted, indicating that hte processing failed at this stage.

Algorithm 12 `ods` function

```
1: function ODS(table, identity, grp_name, jobid, error_files, no_valid_grp, valid_files)
2:     table_OK ← get_table(<table>_OK)
3:     table_PRE_LOAD ← get_table(<table>_PRE_LOAD)
4:     df_Union ← table_OK union by name table_PRE_LOAD
5:     table_ODS ← get_table(<table>_ODS)
6:     Try:
7:     keys ← select COLUMN_NAME from Metadata_Schema where TABLE_NAME == table and PK == True
8:     merge(table_ODS, df_Union, keys)
9:     write_flow_manager(identity, grp_name, 1, jobid, jobid, 0, 0, datetime.now(), datetime.now())
10:    Catch exception e:
11:        error_files.append(table)
12:        no_valid_grp.append(grp_name)
13:       write_flow_manager(identity, grp_name, 1, jobid, jobid, -3, 2, datetime.now(), datetime.now())
14: end function
```

Algorithms 11 and 13 present the pseudocode implementations of the `merge` and `ods` functions, respectively.

Prior to the main execution block that orchestrates the functions described above, for each table requiring processing, the `PRE_LOAD`, `OK`, `ERR`, and `ODS` are created for the given table and initialized as empty tables if they do not already exist. The schema for each is derived from the corresponding `DLT`: `PRE_LOAD` and `OK` share the same schema of the `DLT` table; the `ERR` table includes an additional column, `Descr_Err`, to record the reason for discarded records; the `ODS` table contains two additional columns, `UPD_JOBID` and `UPD_TIME`.

Main Execution Flow

As stated previously, the processing of tables in the L1 layer follows a specific ordering: dimension tables without foreign key are processed first, followed by dimension tables that reference other dimensions, and finally the fact tables. For each table scheduled for processing, the corresponding group name and identity are retrieved from `Metadata_Schema`. Prior to any processing , the function `check_status` is invoked to determine whether the group assigned to the table being processed has previously encountered errors. If `check_status` returns `True`, the processing for that table is skipped, and the loop proceeds to the next table. After each processing function invocation (`pre_load`, `ok`, `ods`), `check_status` is called again to ensure that groups that raise exceptions during intermediate steps are not further processed.

The processing logic depends on the table type:

- Dimension tables without references: only the functions `ok` and `ods` are executed, since no referential contraints need to be validated.
- Dimension tables with references: The functions `pre_load`, `ok` and `ods` are executed sequentially. In this case, the flag parameter passed to `pre_load` is set to 1
- Fact tables: Same sequence, but the flag parameter passed to `pre_load` is set to 0.

Algorithm 13 `get_status` function

```
1: function GET_STATUS(grp_name, jobid)
2:    status ← select STATUS from FLOW_MANAGER where GRP_NAME ==
   grp_name and LEVEL == 1 and TRG_JOBID = jobid
3:    return True if status != -3, else return False
4: end function
```

The pseudocode depicting this logic is represented by Algorithm 14.

The output returned by this notebook is the list of files that encountered errors during the execution of this phase. This list is intended to be used for sending an email notification to the designated supervisor.

Algorithm 14 Execute processing functions for each table in L1

```
Inputs of the code: tables, jobid
for table in tables do
    metadata_filtered_by_table ← select * from Metadata_Schema where TABLE_NAME = table
    grp_name, identity ← select GRP_NAME, IDENTITY from Metadata_Schema where TABLE_NAME = table
    if check_status(grp_name, jobid) then
        continue
    end if
    if table in dimensions then
        ok(table, jobid, grp_name, identity, metadata_filtered_by_table, valid_files, error_files, no_valid_grp)
        if check_status(grp_name, jobid) then
            continue
        end if
        ods(table, identity, grp_name, jobid, error_files, no_valid_grp, valid_files)
    else if table in dim_to_dim then
        flag_dim_to_dim ← 1
        pre_load(table, metadata_schema, jobid, grp_name, identity, valid_files, error_files, no_valid_grp, flag_dim_to_dim)
        if check_status(grp_name, jobid) then
            continue
        end if
        ok(table, jobid, grp_name, identity, metadata_filtered_by_table, valid_files, error_files, no_valid_grp)
        if check_status(grp_name, jobid) then
            continue
        end if
        ods(table, identity, grp_name, jobid, error_files, no_valid_grp, valid_files)
    else if table in facts then
        flag_dim_to_dim ← 0
        pre_load(table, metadata_schema, jobid, grp_name, identity, valid_files, error_files, no_valid_grp, flag_dim_to_dim)
        if check_status(grp_name, jobid) then
            continue
        end if
        ok(table, jobid, grp_name, identity, metadata_filtered_by_table, valid_files, error_files, no_valid_grp)
        if check_status(grp_name, jobid) then
            continue
        end if
        ods(table, identity, grp_name, jobid, error_files, no_valid_grp, valid_files)
    end if
end for
```

6.3.2 Second solution: T-SQL Stored Procedures

The decision to design the L1 layer within the Warehouse, rather than in the Lakehouse, stems from the opportunity to leverage the powerful computation engine provided by the Warehouse. This choice comes at the cost of reduced flexibility compared to the capabilities offered by Spark and results in a more complex, yet explicit data pipeline.

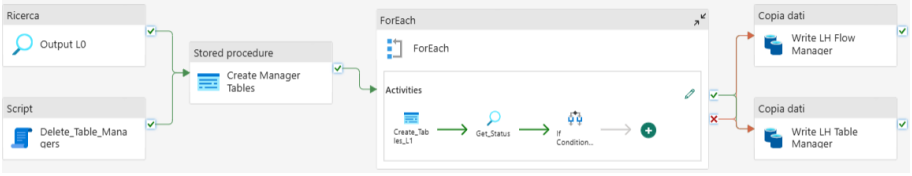


Figure 6.4: Main pipeline for Level 1

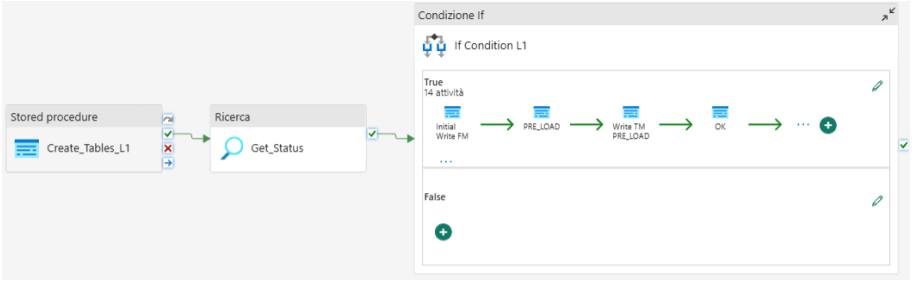


Figure 6.5: Inside the ForEach block of the L1 pipeline

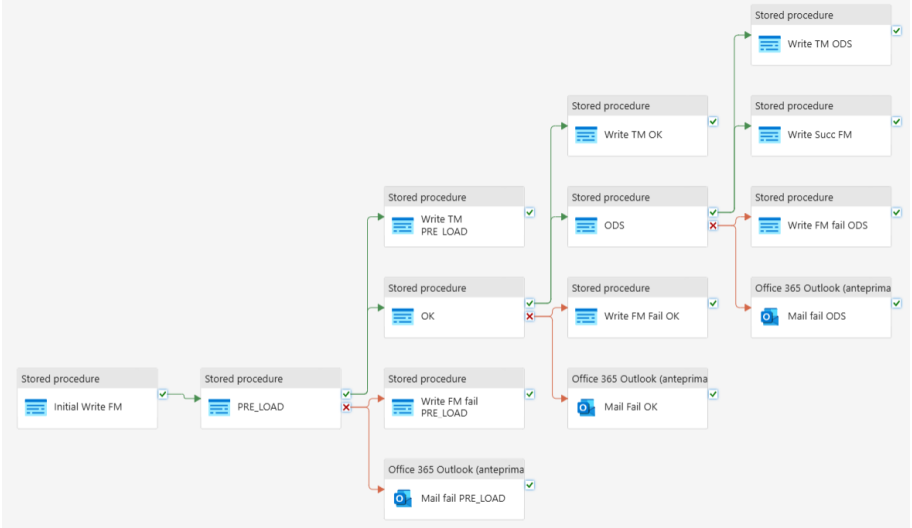


Figure 6.6: Inside the If condition block of the ForEach block of the L1 pipeline

As previously stated, the implementation of the L1 pipeline using T-SQL stored procedures necessitates the orchestration of a significant number of interdependent pipeline activities. The primary orchestrator for the L1 level is depicted by Figure 6.4. The process begins with the parallel execution of two operations:

- The first operation is carried out by the Lookup activity named Output L0, which reads the `Excel` output file produced by the `DLT` notebook during the execution of the L0 pipeline. This file contains the list of tables that are to be processed in the current L1 execution, as well as the `JOBID` associated with the execution instance
- Concurrently, a SQL script is executed to delete the content of the `FLOW_MANAGER` and `TABLE_MANAGER` tables located in the Data Warehouse. These tables stored in the Warehouse are temporary, it means that are populated during the current execution and, upon completion, their content is appended to the corresponding persistent tables stored in the Lakehouse. Consequently, at the beginning of each L1 execution, the Warehouse versions of these tables are cleared.

If the `FLOW_MANAGER` and `TABLE_MANAGER` do not exist in the Warehouse at the time of execution, they are created by the `Create Manager Tables` procedure. Following this initialization activities, a `ForEach` activity is executed, this activity iterates over the list of table names to be processed during the current L1 execution. The iterator variable in this context corresponds to the table name, as extracted form the output of the L0 pipeline.

As illustrated in Figure 6.5, the internal logic of the `ForEach` activity comprises sequential steps. For each table name in the iteration, the pipeline first invokes a Stored Procedure responsible for creating the corresponding empty tables - if they are not present - in the Warehouse, which are the `PRE_LOAD`, `OK`, `ERR`, and `ODS` tables. Subsequently, a Lookup activity is executed to retrieve the processing status of the logical group associated with the current table name. This lookup queries the `FLOW_MANAGER` using the current `JOBID` and the pipeline level (in this case, Level 1). If the retrieved status is different from `-3`, the condition within the `If` activity is satisfied, and the subsequent processing steps for that table are executed. This conditional mechanisms ensures that if any table within a given logical group fails during the processing, all other tables belonging to the same group are intentionally skipped in the same execution cycle.

The `If` condition block, as depicted in Figure 6.6, encapsulates the core processing logic for each table. Specifically, inside this block, several stored procedures are executed in sequence, these are responsible for populating the `FLOW_MANAGER` and `TABLE_MANAGER`, and the `PRE_LOAD`, `OK`, `ERR`, and `ODS` tables associated with the current table name. In addition to the data transformation and control flow logic, email notification activities are integrated into the pipeline connected *on-failure* to the store procedures. These activities are triggered in the event of an error during

the execution of a Store Procedure. Specifically, when a processing step fails, an email is automatically sent to notify the supervisor , this notification includes the step in which the failure verified and the table name that caused the exception.

Store Procedure: `Write_Manager_Tables`

This stored procedure serves the same purpose as the `METADATA_TABLES_HANDLER` notebook introduced in Level 0: It is designed to insert and update records in the `FLOW_MANAGER` and `TABLE_MANAGER`. The procedure receives the following parameters input parameters:

- `table_name`: The name of the table beign processed
- `JOBID`
- `FLAG_FM`: A binary flag indicating whether a new record should be inserted or updated in the `FLOW_MANAGER` (value `1`).
- `FLAG_TM`:A binary flag indicating whether a new record should be inserted or updated in the `TABLE_MANAGER` (value `1`).
- `STATUS, LOAD`
- `STEP`; A string that must be one of the following predefined stages : `PRE_LOAD, OK, ERR` or `ODS`

The first operation performed by the stored procedure consists in retrieving the `identity` and `grp_name` corresponding to the provided table name. This information is obtained by querying the `Metadata_Schema` Delta table, which is accessible in read-only mode from the Warehouse environment due to the linkage established between Warehouse and Lakehouse. Subsequently, the procedure computes the number of rows inserted into the table corresponding to the current step, i.e., `<table_name>_<STEP>`, during the ongoing execution, if the computed number of rows is greater than zero and the `FLAG_TM` parameter is set to `1`, a new record is inserted into the `TABLE_MANAGER`. The inserted record includes the following fields: `identity`, `level` (set to `1`), `grp_name`, `<table_name>_<STEP>`, `JOBID` and the computed `num_rows`.

With regard to the `FLOW_MANAGER` population, the procedure first checks whether a record with the given `grp_name`, `JOBID` and `identity` already exists in the `FLOW_MANAGER` with a `STATUS` equal to `-3` (indicating a previous failure). If such record is found, no further insertion is performed for that group. This condition ensures that if a failure has already been registered for a logical group during the current execution, all subsequent tables within the same group are skipped.

If the `FLAG_FM` is set to `1` and `SKIP = 0`, the procedure proceeds to either insert or update the corresponding record in the `FLOW_MANAGER`. if a record exists for the

combination `identity`, `grp_name`, `level = 1`, and `JOBID`, an update operation is performed: the store procedure sets the new `STATUS` and `LOAD` values based on the current input parameters and sets the `end_date` to the `SYSDATE`. Conversely, if no such records exists, a new insertion is carried out. This new entry includes `identity`, `grp_name`, `LEVEL`, `SRC_JOBID` and `TRG_JOBID` (both equal to `JOBID`), `STATUS`, `LOAD`, the current `start_date`, and a `NULL` value for `end_date`, which will be populated upon completion or failure of the transformation process.

Table 6.6 tabulates the input parameters for the `Initial Write FM` call

table_name	JOBID	FLAG_FM	FLAG_TM	STATUS	LOAD	STEP
<iterator>	<jobid from the DLT output Excel file>	1	0	1	2	PRE_LOAD

Table 6.6: Parameters for the Initial Write FM stored procedure call

Store Procedure: `PRE_LOAD`

It receives as input the `JOBID` of the current execution and the `table_name` of the table being processed. The logic implemented in this stored procedure closely resembles that of the `pre_load` function described in the `PRE_LOAD, OK, ERR, ODS` notebook used in the Spark-based Level 1 Pipeline. However, there are key differences arising from the limitations of T-SQL within the Fabric environment.

First, unlike in Spark SQL, where it is possible to define a `ROW_NUMBER` function directly over a distributed DataFrame to identify the latest record per key, in T-SQL such operations require manual implementation. This is due to the fact that T-SQL does not support applying window functions in the same flexible way as Python Spark does, especially when dynamic table and column names are involved in the logic. As a result, the logic must be expressed explicitly using dynamic SQL.

Another significant difference lies in the absence of support for temporary tables and table variables in Fabric T-SQL. Consequently, to maintain intermediate results, it is necessary to create a physical table. This auxiliary table named `DependenceStaging` stores metadata regarding referential integrity: for the table passed as input to the stored procedure, it lists the referenced destination tables, the columns in the destination tables used for reference, and the corresponding source columns.

A toy example to illustrate the contents of the `DependencesStaging` table is shown in Table 6.7.

Once this reference mapping table is created, the procedure iterates over its rows. For each referenced table, if referential relationships are present, the corresponding

`pre-load` data is computed following the same logic adopted in the `PRE_LOAD, OK, ERR, ODS` notebook. Subsequently, a record is inserted into the `TABLE_MANAGER` to log the reference table name and the number of rows loaded during the operation.

Considering the metadata in Table 6.5, if the current table being processed is `table_6`, that represents a fact table referencing several dimensions, the `Dependences Staging` table will be populated as follows:

Referenced_Table	Dest_Columns_To_Referentiate	Src_Columns_That_Referentiate
table_1	col_1_1	col_1_1
table_4	col_4_1	col_4_1
table_3	col_3_1	col_3_1

Table 6.7: Example of `DependencesStaging` contents when processing

On the other hand, if the current table being processed is `table_1`, which is a dimension table not referencing any other tables, the `DependencesStaging` table will remain empty. This is expected, as only fact tables (such as `table_6`) and secondary dimensions in a Snowflake schema typically contain foreign keys pointing to dimension tables, whereas primary dimensions do not have outbound dependencies.

To conclude, this stored procedure populates the `PRE_LOAD` tables of the tables referenced by the current table being processed. Specifically, it inserts into the destination tables the values of the key columns that are present in the source table but no yet available in the corresponding destination tables, ensuring compliance with referential integrity constraints

This stores procedure is linked to four distinct activities

- A success path to the `Write_Manager_Tables` store procedure named `Write TM PRE_LOAD`, which logs into the `TABLE_MANAGER` the `PRE_LOAD` of the current table being processed and the number of rows inserted
- An *on-failure* connection to the `Write_Manager_Tables` store procedure named `Write FM fail PRE_LOAD`, which sets `STATUS = - 3` and `end_date = SYSDATE` corresponding to the group associated to the current processed table, the Level 1 and the `JOBID`
- An email notification activity that informs the supervisor of the failure in the `PRE_LOAD` step for the current table.
- An *on-success* connection to the `OK` store procedure

The parameters used by the `Write TM PRE_LOAD` and `Write FM fail PRE_LOAD` procedures are detailed in Tables 6.8 and 6.9, respectively.

table_name	JOBID	FLAG_FM	FLAG_TM	STATUS	LOAD	STEP
<iterator>	<jobid from the DLT output Excel file>	0	1	1	2	PRE_LOAD

Table 6.8: Parameters for the Write TM PRE_LOAD stored procedure call

table_name	JOBID	FLAG_FM	FLAG_TM	STATUS	LOAD	STEP
<iterator>	<jobid from the DLT output Excel file>	1	0	-3	2	PRE_LOAD

Table 6.9: Write FM fail PRE_LOAD

Store Procedure: `OK`

For the `OK` stored procedure, the process used to extract the data for the table currently being processed follows the same approach described in the `PRE_LOAD` stored procedure. Specifically, the `row_number` function is manually implemented due to the limitations of Fabric T-SQL. Apart from this technical adaptation, the stored procedure replicates the same logic as the `ok` function found in the `PRE_LOAD`, `OK`, `ERR`, and `ODS` notebook used in the L1 PySpark-based implementation pipeline. It can be stated that the logic expressed in the notebook and the one implemented in the stored procedure are equivalent in terms of functionality. The main difference lies in the programming languages and execution environments.

For a given input table name, the store procedure populates the `OK` and `ERR` tables. The `OK` table contains the records that satisfy the data quality controls - specifically, check nullability and the presence of negative values in numeric columns. Conversely, the `ERR` table stores the records that fail these quality checks, along with a description indicating the reason for their exclusion.

As in the previous step, the `OK` stored procedure is connected - upon successful execution - to two subsequent procedures: the `ODS` stored procedure and the `Write_Manager_Tables` stored procedure named `Write TM OK`, which logs in the `TABLE_MANAGER` the name of the corresponding OK table and the number of rows inserted.

On failure, the `OK` procedure triggers two actions: first, it calls the `Write_Manager_Tables` stored procedure named `Write FM Fail OK`, which registers a failure for the `OK` step associated with the group name of the table being processed in the `FLOW_MANAGER` at Level 1; second, it triggers an email activity to notify the supervisor of the failure in the `OK` step

Tables 6.10 and 6.11 illustrate the parameters passed, respectively, to the `Write TM OK` and `Write FM Fail OK` stored procedures.

table_name	JOBID	FLAG_FM	FLAG_TM	STATUS	LOAD	STEP
<iterator>	<jobid from the DLT output Excel file>	0	1	1	2	OK

Table 6.10: Parameters for the Write TM OK stored procedure call

table_name	JOBID	FLAG_FM	FLAG_TM	STATUS	LOAD	STEP
<iterator>	<jobid from the DLT output Excel file>	1	0	-3	2	OK

Table 6.11: Write FM fail OK

Store Procedure: `ODS`

It is equivalent to the `ods` function implemented in the `PRE_LOAD`, `OK`, `ERR`, and `ODS` notebook of the L1 PySpark-based pipeline. In this step, the records from the `OK` and `PRE_LOAD` tables are unified into a single DataFrame, which is then used to populated the corresponding ODS table. While the overall logic mirrors the `PySpark` implementation, there is a fundamental difference related to the `MERGE` operation. The `PySpark` notebook allows to import the `DeltaTable API` which natively supports the `MERGE` operation on Delta Tables. However, in the Microsoft Fabric Warehouse Experience, the standard T-SQL `MERGE` clause is not supported.

This limitation arises because Microsoft Fabric does not rely on a traditional relational database engine. Instead, it uses Delta Lake tables stored as Parquet Files in OneLake.

Therefore, to simulate the behavior in the `ODS` stored procedure, first, an `UPDATE` operation is executed, matching on the key columns, to update records in the `ODS` table; then, an `INSERT` is performed to add any new records that were not updated. A drawback of this approach is the use of `JOIN` operations, which may reduce efficiency when processing large-scale datasets.

As with the previous steps, the `ODS` process is also connected *on-success* to the store procedure that logs the table name and the number of rows inserted into the `ODS` table in the `TABLE_MANAGER`. It is also connected, *on-failure*, to the store procedure that registers the failure in the `FLOW_MANAGER` and consequently trigers an email notification activity.

Additionally, it is connected *on-success* to a `Write_Manager_Tables` stored procedure named `Write Succ FM`, which, for the group associated with the current table name being processed (at `Level 1` and for the current `JOBID`), sets `STATUS = 0, LOAD = 0`, and `end_date = SYSDATE`, indicating successful completion of Level 1 by that group. This stored procedure will only be executed if the previous steps did not

raise errors or if none of the tables belonging to the same group failed when they were processed.

The input parameters for the different instances of the `Write_Manager_Tables` stored procedure - namely `Write TM ODS`, `Write Succ FM`, and `Write FM Fail ODS` - are detailed in Tables 6.12, 6.13, and 6.14, respectively.

table_name	JOBID	FLAG_FM	FLAG_TM	STATUS	LOAD	STEP
<iterator>	<jobid from the DLT output Excel file>	0	1	1	2	ODS

Table 6.12: Parameters for the Write TM ODS stored procedure call

table_name	JOBID	FLAG_FM	FLAG_TM	STATUS	LOAD	STEP
<iterator>	<jobid from the DLT output Excel file>	1	0	0	0	ODS

Table 6.13: Parameters for the Write Succ FM stored procedure call

table_name	JOBID	FLAG_FM	FLAG_TM	STATUS	LOAD	STEP
<iterator>	<jobid from the DLT output Excel file>	1	0	-3	2	ODS

Table 6.14: Parameters for the Write TM fail ODS stored procedure call

Figure 6.4 shows that once the `ForEach` activity finishes, the records from the `FLOW_MANAGER` and `TABLE_MANAGER` populated during the `L1` level will be appended to the existing and persistent `FLOW_MANAGER` and `TABLE_MANAGER` tables, respectively stored in the `Lakehouse`, thanks to the `Write LH Flow Manager` and `Write LH Table Manager Copy Data` activities.

6.3.3 Third Solution: Dataflow Gen2

For the purpose of this thesis, Dataflows are used exclusively in the step responsible for data cleansing, which corresponds to the function `ok` in the first solution and the stored procedure `ok` in the second solution.

Figures 6.7 - 6.11 illustrate the pipeline employed for this solution, it is shown how several store procedures and `Lookup` activities from the SQL-based solution have been retained, such as those for the insertion in `TABLE_MANAGER, FLOW_MANAGER, PRE_LOAD` and `ODS`. However, only the `OK` step has been replaced by the Dataflow. Some variations are introduced in this architecture. For example, in the two previous solutions, the steps `PRE_LOAD, OK`, and `ODS` were executed sequentially for each table.

In contrast, this approach executes `PRE_LOAD` for all tables first, then performs the `OK` step for all tables, and finally the `ODS` step for all tables

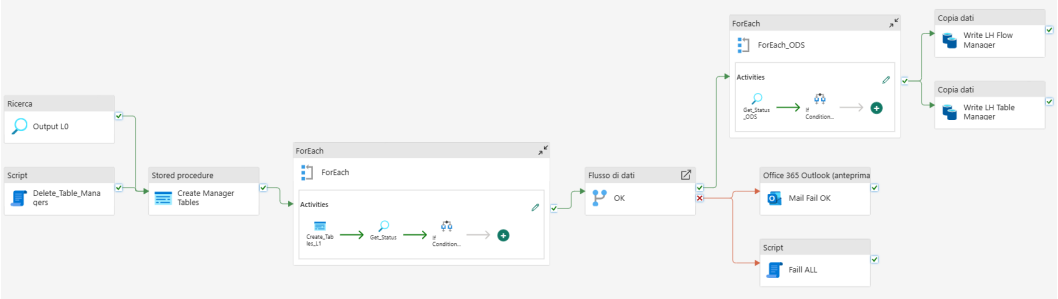


Figure 6.7: Main pipeline for the Dataflow L1 solution

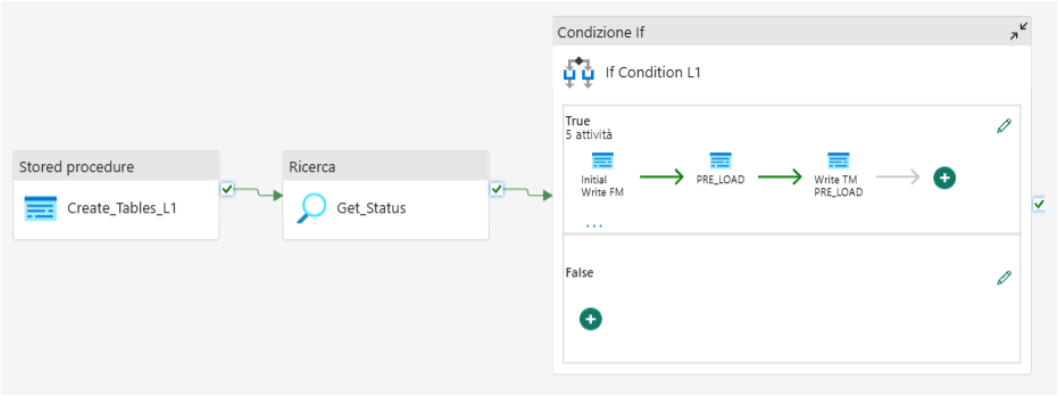


Figure 6.8: Inside the first ForEach for the Dataflow L1 solution

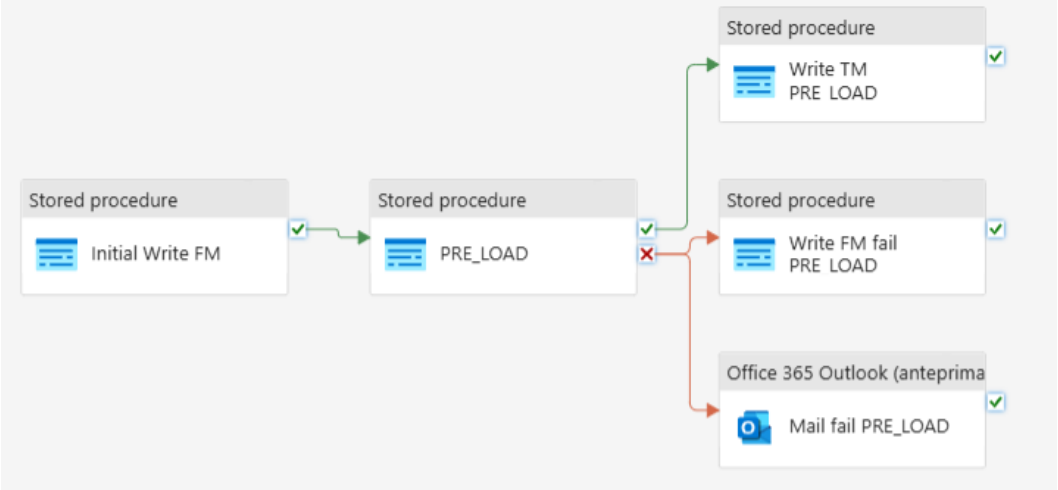


Figure 6.9: Inside the If condition block of the ForEach activity

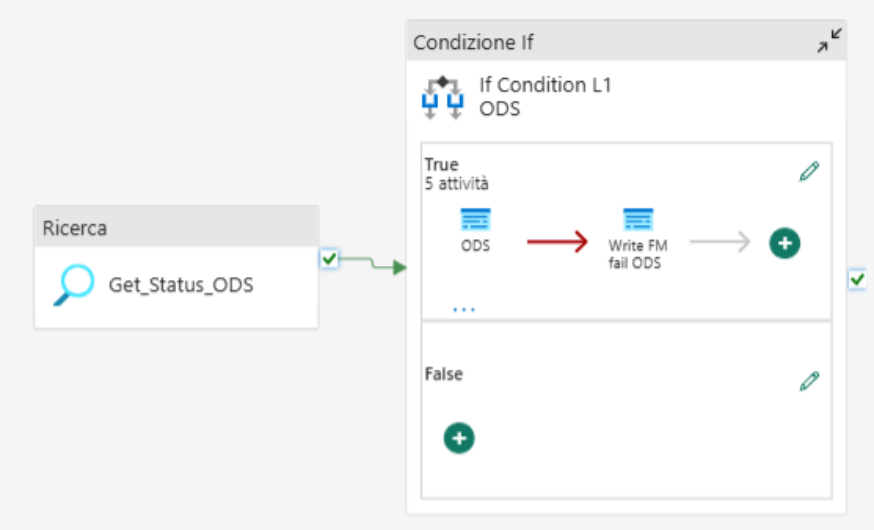


Figure 6.10: Inside the second ForEach for the Dataflow L1 solution

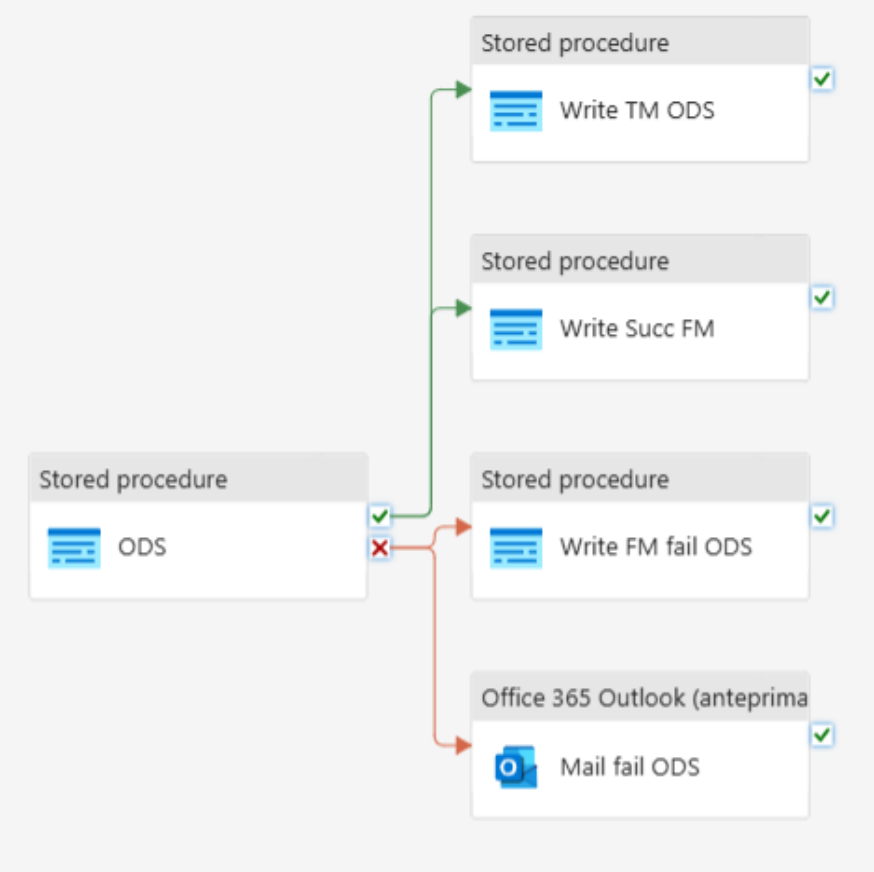


Figure 6.11: Inside the If condition block of the second ForEach activity

Figure 6.12 shows the detailed implementation of the Dataflow.

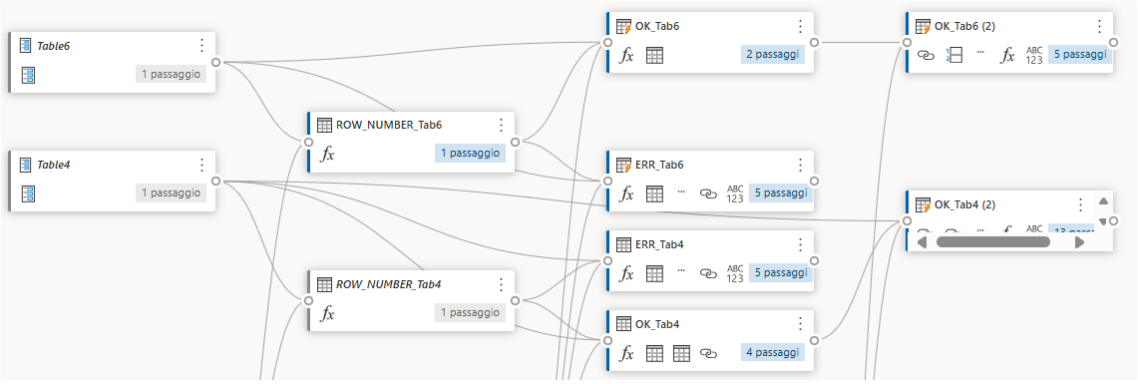


Figure 6.12: OK step, Dataflow Implementation

In this Dataflow, several components are present, including table imports such as `Table6`, `Table4`, and others. Two main functions are also defined: `GetRowNumber` and `CheckConstr`. The `GetRowNumber` function retrieves data from the `DLT` table by selecting records with a `JOBID` greater than the most recent successful `JOBID` at level 1 for the corresponding group. The function then applies a `ROW_NUMBER()` operation, partitioning the data by key and ordering it by `JOBID` in descending order and by `FLG_NEG` in ascending order; the `CheckConstr` function is responsible for validating the records according to defined constraints - Nullability and Negative values in numeric columns - the records that meet the constraints are considered valid and returned as part of the `<table>_OK` output, whil those that fail validation are redirected to the `<table>_ERR` output.

Each query named `ROW_NUMBER_Tab<x>` calls the `GetRowNumber` function, after which the `CheckConstr` function is applied to separate valid records from the invalid ones, resulting in two ouptuts: `Tab<x>_OK` and `Tab<x>_ERR`. This mechanism mirrors the logic adopted in the previous solutions.

All the tables to be processed in this step are manually imported into the local memory of the Dataflow element. Similary, the corresponding `OK` and `ERR` tables for each input are manually obtained. This is due to the nature of Dataflows in Fabric, which are explicetly designed as a low-code tool for users with limited (or without) programming skills. Consequently, it is not possible to fully parametrize and dynamically manage the logic within the Dataflow environment.

While more advanced users have the option to define transformations using `M` code scripts, this approach still does not override the fundamental low-code paradigm of Dataflows.

This limitation is also evident in the configuration of output destinations. Unlike

the input phase, where a single import action can reference multiple tables, each output requires an explicit, manual destination configuration. Users must define the destination of data via the graphical interface; additionally, for each export, the table name in the destination must be specified, along with the list of exported columns and the data type for each column. It is not possible to programmatically define how data should be stored in the target systems, which further limits dynamic configuration capabilities within Dataflows.

Another important consideration is that all operations executed within a Dataflow are treated atomically. This means that if an error occurs in the processing of a single table, none of the other tables - regardless of their independence - will be processed. As a result, with the current implementation, it was not possible to selectively update the `FLOW_MANAGER` with `STATUS = -3` to indicate which specific group failed. Instead, a workaround was adopted, in which a SQL script is executed to assign a `STATUS` of -3 to all groups involved in the current execution, this is the task of the `Fail ALL` SQL script present in figure 6.7. Nevertheless, this limitation can be addressed through an alternative implementation that makes use of the `Switch activity` within a pipeline. This activity is equivalent to an `elif` statement in Python, depending on the value of a variable, a different branch is executed. A preliminary draft solution is shown in Figure 6.13

This alternative allows a sequential execution pattern similar to that of the second solution, enabling separate tracking of executions for each table, as figures 6.14 and 6.15, in this way, it becomes possible to monitor and record the status of each group individually. However, the main drawback of this approach is that, for a total of `N` tables the same execution sequence must be replicated `N` times. Moreover, it requires the replication of `N` distinct Dataflows, each performing the same set of operations but with different input tables and output destinations, figure 6.16 displays the structure of the Dataflow with this alternative solution.

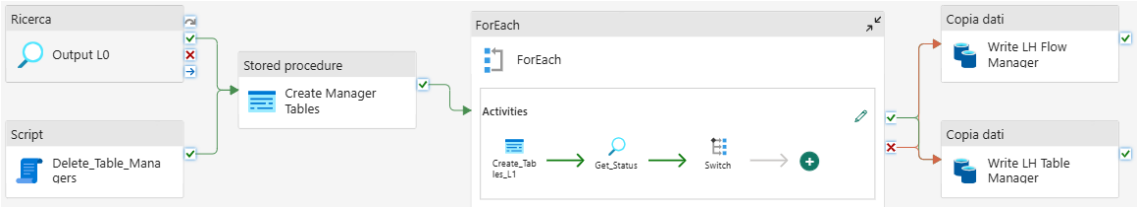


Figure 6.13: Alternative Dataflow solution for Level 1

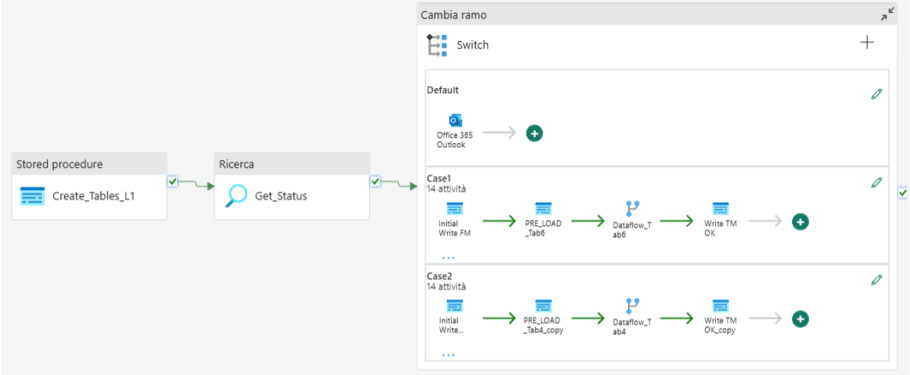


Figure 6.14: Alternative ForEach elements

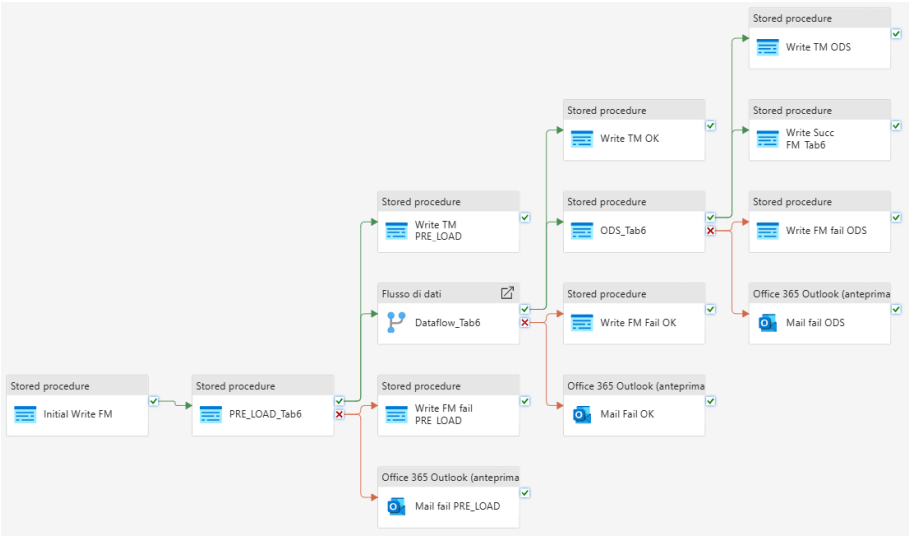


Figure 6.15: Switch Activity for the alternative L1 solution

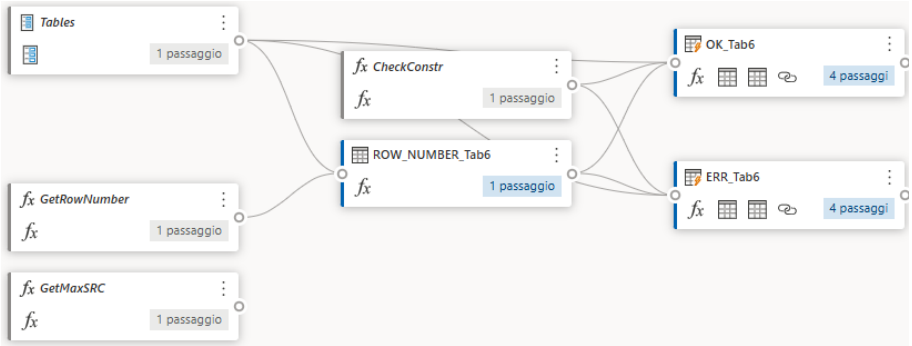


Figure 6.16: Alternative Dataflow for the Switch Activity

Finally, the query blocks located on the rightmost side of Figure 6.7 are instead responsible for updating the `TABLE_MANAGER` for each table, by writing the number of rows successfully loaded into the `OK` table during the current execution.

6.3.4 Extra: Master Data Management (MDM)

The implementation of the corporate ETL framework is designed to extend up to the `ODS` layer, as any processing beyond this point depends on specific client requirements.

Nevertheless, for the purposes of this project, the `MDM` layer has been included exclusively for demonstration purposes. This step handles the de-normalization of the Snowflake schema into a Star Schema, which is then stored in the Data Warehouse.

For source dimension tables that reference other target dimension tables, a left join is performed from the target to the source, resulting in a table named `<target_table>_MDM`. This table includes all columns from the source table for those records whose keys match the corresponding foreign keys.

The MDM is considered an extra layer, meaning it is excluded from the performance evaluations presented in the Results chapter.

Chapter 7

Use Case: Mediamente Consulting ETL Framework

In this chapter, it is presented a case study based on a sample business dataset to further support and illustrate the theoretical concepts introduced in the previous chapter.

The initial data model follows a Snowflake Schema, representing the sales of a local store. It consists of a single fact table, *Vendite (Sales)*, and five dimension tables: *Prodotti (Products)*, *Tempo (Time)*, *Clienti (Customers)*, *Pagamento (Payment)*, and *Località (Location)*. *Località* is connected to *Clienti* through a foreign key. The overall data model is depicted in Figure 7.1

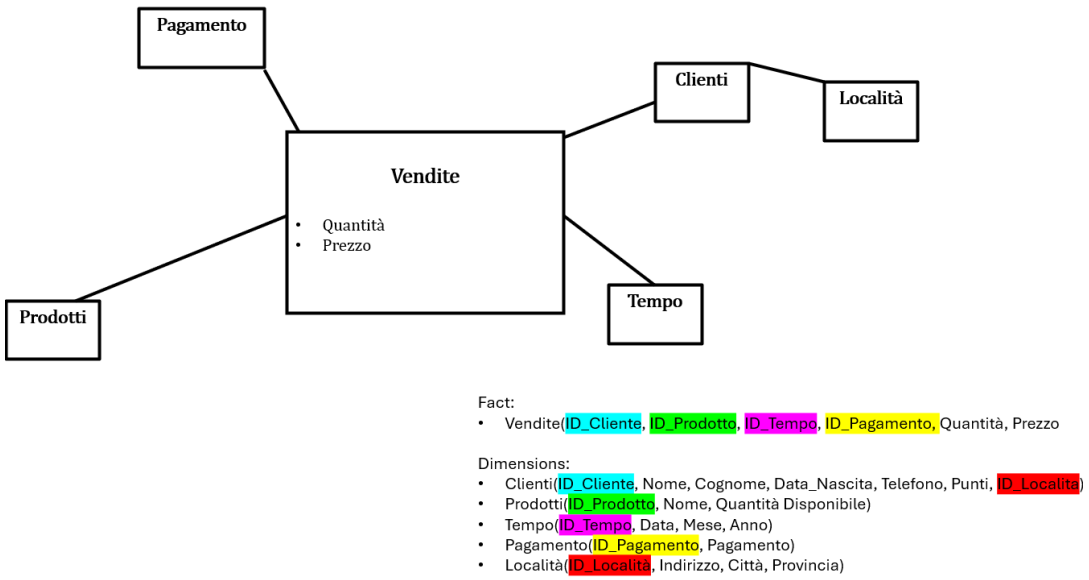


Figure 7.1: Snowflake Data Model toy example

The *Vendite* table includes six columns: `ID_Cliente, ID_Prodotto, ID_Tempo`, and `ID_Pagamento`, which together form the composite primary key of the table, also, the table contains two measures: `Quantità (Quanity)` and `Prezzo di vendita (Sales Price)`. On the other hand, the dimension tables are defined as follows:

- **Clienti**: `ID_Cliente` (Primary Key), `Nome` (Name), `Cognome` (Last Name), `Data di Nascita` (Birthdate), `Telefono` (Cellphone Number), `Punti` (Fidelity card points), `ID_Località` (Foreign Key)
- **Prodotti**: `ID_Prodotto` (PK), `Nome` (Name), `Quantità disponibile` (Available Stock)
- **Tempo**: `ID_Tempo` (PK), `Data di vendita` (Sales Date), `Mese` (Month), `Anno` (Year)
- **Pagamento**: `ID_Pagamento` (PK), `Tipo di Pagamento` (Payment Type)
- **Località**: `ID_Località` (PK), `Indirizzo` (Address), `Città` (City), `Provincia` (Province)

Since the dataset is available only in Italian while the documentation and explanations are provided in English, it is crucial to define a consistent legend for interpreting metadata values, particularly those used in the metadata tables. The adopted mapping is as follows:

- Dimension = Anagrafica
- Fact = Movimento

The data is extracted from `CSV` files loaded by the client into the Lakehouse's main directory Files, then the program moves each file to either the *Anagrafiche* or *Movimenti* subfolders, according to its related `IDENTITA` specified by the `Excel` metadata file

Figure 7.2 displays the completed `Excel` metadata file, which is used to inform the system about table identities, group affiliations, table and column names, primary and foreign keys, non-nullable fields, non-negative constraints, and inter-table relationships. Additionally the corresponding Delta Lake table generated from the metadata file and stored in the Lakehouse is portrayed.

For brevity's sake, not all tables from every processing step will be shown. The focus will be limited to a single dimension (`Clienti`), a dimension containing a reference to another dimension (`Località`), and the fact table (`Vendite`). From these, only the most relevant steps and resulting tables will be presented.

In the following sections, the process will be executed five times, with each run thoroughly explained to provide a comprehensive understanding of each ETL action performed during the workflow.

NOME_TABELLA	IDENTITA	GRP_NAME	NOME_COLONNA	TIPO	PK	NULL	MINORE_0	FK	RIFERIMENTO_TABELLA	RIFERIMENTO_FK
Clienti	ANA	CLI	ID_CLIENTE	INT	Y	N	N	N		
Clienti	ANA	CLI	NOME	STRING	N	N	N	N		
Clienti	ANA	CLI	COGNOME	STRING	N	N	N	N		
Clienti	ANA	CLI	DATA_NASCITA	STRING	N	N	N	N		
Clienti	ANA	CLI	TELEFONO	STRING	N	Y	N	N		
Clienti	ANA	CLI	PUNTI	INT	N	N	N	N		
Clienti	ANA	CLI	ID_LOC	INT	N	N	N	N		
Località	ANA	CLI	ID_LOC	INT	Y	N	N	Y	Clienti	ID_LOC
Località	ANA	CLI	INDIRIZZO	STRING	N	N	N	N		
Località	ANA	CLI	CITTA	STRING	N	N	N	N		
Località	ANA	CLI	PROVINCIA	STRING	N	N	N	N		
Pagamento	ANA	PAG	ID_PAGAMENTO	INT	Y	N	N	Y		
Pagamento	ANA	PAG	PAGAMENTO	STRING	N	N	N	N		
Prodotti	ANA	PRO	ID_PRODOTTO	INT	Y	N	N	Y		
Prodotti	ANA	PRO	NOME	STRING	N	N	N	N		
Prodotti	ANA	PRO	QUANTITA_DISPON	INT	N	N	N	N		
Prodotti	ANA	PRO	PREZZO	DOUBLE	N	N	N	N		
Tempo	ANA	TEM	ID_TEMPO	INT	Y	N	N	Y		
Tempo	ANA	TEM	DATA	DATE	N	N	N	N		
Tempo	ANA	TEM	MESE	INT	N	N	N	N		
Tempo	ANA	TEM	ANNO	INT	N	N	N	N		
Vendite	MOV	VEN	ID_CLIENTE	INT	Y	N	N	Y	Clienti	ID_CLIENTE
Vendite	MOV	VEN	ID_PRODOTTO	INT	Y	N	N	Y	Prodotti	ID_PRODOTTO
Vendite	MOV	VEN	ID_TEMPO	INT	Y	N	N	Y	Tempo	ID_TEMPO
Vendite	MOV	VEN	ID_PAGAMENTO	INT	Y	N	N	Y	Pagamento	ID_PAGAMENTO
Vendite	MOV	VEN	QUANTITA	INT	N	N	N	N		
Vendite	MOV	VEN	PREZZO	DOUBLE	N	N	N	N		

Figure 7.2: On the left, the metadata from the Excel file; on the right, its mapping to a Delta Table

7.1 First ETL Execution – 2025-07-18 21:47:51

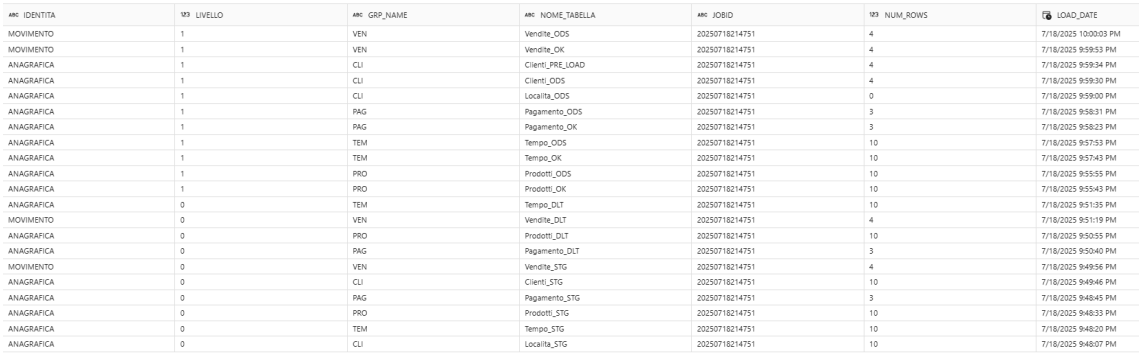
The first execution of the worfklow started on the 18th of July, 2025 at 21:47:51 UTC, therefore, the identifier `JOBID` for this execution is `20250718214751`.

The `STG` step successfully processed all tables. However, during the `DLT` phase, the table `Clienti` raised an exception and was not loaded. Consequently, a `STATUS = -3` was registered in the `FLOW_MANAGER` for the group `CLI` as shown in Figure 7.3

IDENTITA	GRP_NAME	LIVELLO	SRC_JOBID	TRG_JOBID	STATUS	LOAD	START_DATE	END_DATE
MOVIMENTO	VEN	1	20250718214751	20250718214751	0	0	7/18/2025 9:59:12 PM	7/18/2025 10:00:06 PM
ANAGRAFICA	CLI	1	20250718214751	20250718214751	0	0	7/18/2025 9:56:46 PM	7/18/2025 9:59:03 PM
ANAGRAFICA	PAG	1	20250718214751	20250718214751	0	0	7/18/2025 9:58:05 PM	7/18/2025 9:58:34 PM
ANAGRAFICA	TEM	1	20250718214751	20250718214751	0	0	7/18/2025 9:57:26 PM	7/18/2025 9:57:55 PM
ANAGRAFICA	PRO	1	20250718214751	20250718214751	0	0	7/18/2025 9:55:16 PM	7/18/2025 9:55:59 PM
ANAGRAFICA	TEM	0	19000101000000	20250718214751	0	0	7/18/2025 9:48:14 PM	7/18/2025 9:51:39 PM
MOVIMENTO	VEN	0	19000101000000	20250718214751	0	0	7/18/2025 9:49:52 PM	7/18/2025 9:51:22 PM
ANAGRAFICA	CLI	0	19000101000000	20250718214751	-3	2	7/18/2025 9:47:58 PM	7/18/2025 9:51:07 PM
ANAGRAFICA	PRO	0	19000101000000	20250718214751	0	0	7/18/2025 9:48:27 PM	7/18/2025 9:50:58 PM
ANAGRAFICA	PAG	0	19000101000000	20250718214751	0	0	7/18/2025 9:48:39 PM	7/18/2025 9:50:43 PM

Figure 7.3: Content of the `FLOW_MANAGER` after the first ETL process execution

Since `Località` belongs to the same group `CLI` as `Clienti`, and the group raised an error, `Località` was also not loaded into `Località_DLT`, as displayed by the `Table_Manager` in Figure 7.4. This happened even though `Località` itself did not raise any error, because there is a dependency between tables belonging to the same group.



IDENTITA	LIVELLO	GRP_NAME	NOME_TABELLA	JOBID	NUM_ROWS	LOAD_DATE
MOVIMENTO	1	VEN	Vendite_ODS	20250718214751	4	7/18/2025 10:00:03 PM
MOVIMENTO	1	VEN	Vendite_OK	20250718214751	4	7/18/2025 9:59:53 PM
ANAGRAFICA	1	CLI	Clienti_PRE_LOAD	20250718214751	4	7/18/2025 9:59:34 PM
ANAGRAFICA	1	CLI	Clienti_ODS	20250718214751	4	7/18/2025 9:59:30 PM
ANAGRAFICA	1	CLI	Localita_ODS	20250718214751	0	7/18/2025 9:59:00 PM
ANAGRAFICA	1	PAG	Pagamento_ODS	20250718214751	3	7/18/2025 9:58:31 PM
ANAGRAFICA	1	PAG	Pagamento_OK	20250718214751	3	7/18/2025 9:58:23 PM
ANAGRAFICA	1	TEM	Tempo_ODS	20250718214751	10	7/18/2025 9:57:53 PM
ANAGRAFICA	1	TEM	Tempo_OK	20250718214751	10	7/18/2025 9:57:43 PM
ANAGRAFICA	1	PRO	Prodotti_ODS	20250718214751	10	7/18/2025 9:55:55 PM
ANAGRAFICA	1	PRO	Prodotti_OK	20250718214751	10	7/18/2025 9:55:43 PM
ANAGRAFICA	0	TEM	Tempo_DLT	20250718214751	10	7/18/2025 9:51:35 PM
MOVIMENTO	0	VEN	Vendite_DLT	20250718214751	4	7/18/2025 9:51:19 PM
ANAGRAFICA	0	PRO	Prodotti_DLT	20250718214751	10	7/18/2025 9:50:55 PM
ANAGRAFICA	0	PAG	Pagamento_DLT	20250718214751	3	7/18/2025 9:50:40 PM
MOVIMENTO	0	VEN	Vendite_STG	20250718214751	4	7/18/2025 9:49:56 PM
ANAGRAFICA	0	CLI	Clienti_STG	20250718214751	10	7/18/2025 9:49:46 PM
ANAGRAFICA	0	PAG	Pagamento_STG	20250718214751	3	7/18/2025 9:48:45 PM
ANAGRAFICA	0	PRO	Prodotti_STG	20250718214751	10	7/18/2025 9:48:33 PM
ANAGRAFICA	0	TEM	Tempo_STG	20250718214751	10	7/18/2025 9:48:20 PM
ANAGRAFICA	0	CLI	Localita_STG	20250718214751	10	7/18/2025 9:48:07 PM

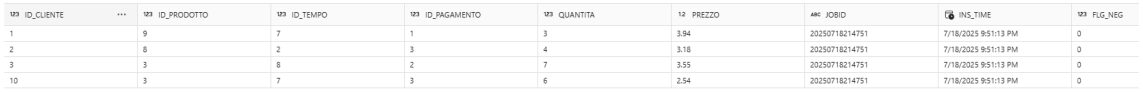
Figure 7.4: Content of the `TABLE_MANAGER` after the first ETL process execution

`Clienti` raised an error at Level 0. Nevertheless, since the processing levels are independent, `Clienti` and `Località` were processed successfully at Level 1, as shown in the `FLOW_MANAGER` in Figure 7.3, which reports `STATUS = 0` for Level 1 and group `CLI`, However, the input for Level 1 is `Clienti_DLT` table generated in the previous level, where `Clienti` failed. Therefore, this table is empty, and consequently `Clienti_ODS` is an empty table.

Focusing now on the `Vendite` table, Figures 7.5, 7.6, 7.7, and 7.8 show the table processed successfully at the different stages: `STG`, `DLT`, `OK`, and `ODS`. It is interesting to examine the keys in `Vendite_DLT` in the column `ID_Cliente`, because the `pre-load` step verifies whether these keys exist in the `ODS` tables referenced by `Vendite`. Specifically, the keys in `Vendite_DLT` should be present in `Clienti_ODS`; if they are missing, they are added, with the other fields left as `NULL` or default values. This is exactly what happens in this case: since `Clienti_ODS` is empty. `Vendite` pre-loads the missing keys into `Clienti_PRE_LOAD`, and these records are then copied into `Clienti_ODS`, as Figure 7.9 and 7.10 illustrate respectively.

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME
2	8	2	3	4	3.18	20250718214751	7/18/2025 9:49:54 PM
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:49:54 PM
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:49:54 PM
10	3	7	3	6	2.54	20250718214751	7/18/2025 9:49:54 PM

Figure 7.5: Content of the table `Vendite_STG` after the first execution.



ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:51:13 PM	0
2	8	2	3	4	3.18	20250718214751	7/18/2025 9:51:13 PM	0
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:51:13 PM	0
10	3	7	3	6	2.54	20250718214751	7/18/2025 9:51:13 PM	0

Figure 7.6: Content of the table `Vendite_DLT` after the first execution.

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:59:48 PM	0
2	8	2	3	4	3.18	20250718214751	7/18/2025 9:59:48 PM	0
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:59:48 PM	0
10	3	7	3	6	2.54	20250718214751	7/18/2025 9:59:48 PM	0

Figure 7.7: Content of the table `Vendite_OK` after the first execution.

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG	UPD_JOBID	UPD_TIME
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:59:48 PM	0	20250718214751	7/18/2025 9:59:58 PM
2	8	2	3	4	3.18	20250718214751	7/18/2025 9:59:48 PM	0	20250718214751	7/18/2025 9:59:58 PM
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:59:48 PM	0	20250718214751	7/18/2025 9:59:58 PM
10	3	7	3	6	2.54	20250718214751	7/18/2025 9:59:48 PM	0	20250718214751	7/18/2025 9:59:58 PM

Figure 7.8: Content of the table `Vendite_ODS` after the first execution.

ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME	FLG_NEG
1							20250718214751	7/18/2025 9:59:23 PM	0
3							20250718214751	7/18/2025 9:59:23 PM	0
10							20250718214751	7/18/2025 9:59:23 PM	0
2							20250718214751	7/18/2025 9:59:23 PM	0

Figure 7.9: Content of the table `Clienti_PRE_LOAD` after the first execution

ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME	FLG_NEG	UPD_JOBID	UPD_TIME
1							20250718214751	7/18/2025 9:59:23 PM	0	20250718214751	7/18/2025 9:59:27 PM
3							20250718214751	7/18/2025 9:59:23 PM	0	20250718214751	7/18/2025 9:59:27 PM
10							20250718214751	7/18/2025 9:59:23 PM	0	20250718214751	7/18/2025 9:59:27 PM
2							20250718214751	7/18/2025 9:59:23 PM	0	20250718214751	7/18/2025 9:59:27 PM

Figure 7.10: Content of the table `Clienti_ODS` at the first exection

To conclude the analysis of this first execution, the `FLOW_MANAGER` in Figure 7.3 shows that all groups successfully completed the process, except for group `CLI` at Level 0. Conversely, the `TABLE_MANAGER` in Figure 7.4 displays all the tables that were populated during this execution, along with the number of rows inserted into each of them and the time at which the population happened.

7.2 Second ETL Execution – 2025-07-18 22:15:25

Between the first and the second execution, the user uploaded new data through the `Clienti.csv` and `Vendite.csv` files.

In the second run, the new data from the `Clienti` file is correctly loaded into Level 0. As previously explained, the STG step performs an append operation, meaning that it includes both the data tagged with the previous execution's `JOBID` and the data from the current one (`20250718221525`). Since the `CLI` group has never been successfully loaded in Level 0 so far, there is no recent `JOBID` identifying a correct loading of these records, therefore, the records passed to the `DLT` step are those whose `JOBID` is greater than the default `JOBID` (`19000101000000`). In this

case, all the records in STG are considered new and will carry `FLG_NEG = 0` and the `JOBID` of the current execution. Figures 7.11 and 7.12 show the corresponding tables.

ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME
24	Alessandro	Nesta	10-12-1990	555-555-570	2000	17	20250718221525	7/18/2025 10:16:31 PM
27	Ebenezer	Scrooge	03-03-1900	555-555-687	10234	12	20250718221525	7/18/2025 10:16:31 PM
30	Bruce	Wayne	04-05-1988		10000	7	20250718221525	7/18/2025 10:16:31 PM
1	Lionel	Messi	13-03-1989	555-555-555	19646	1	20250718214751	7/18/2025 9:49:44 PM
2	Cristiano	Ronaldo	28-03-1989	555-555-556	18320	2	20250718214751	7/18/2025 9:49:44 PM
3	Kylian	Mbappé	23-01-1988	555-555-557	20340	3	20250718214751	7/18/2025 9:49:44 PM
4	Erling	Haaland	29-03-1977	555-555-558	17830	4	20250718214751	7/18/2025 9:49:44 PM
5	Kevin	De Bruyne	22-10-1976	555-555-559	21048	5	20250718214751	7/18/2025 9:49:44 PM
6	Luka	Modrić	19-08-1994	555-555-560	20446	6	20250718214751	7/18/2025 9:49:44 PM
7	Virgil	van Dijk	15-08-1967	555-555-561	19045	7	20250718214751	7/18/2025 9:49:44 PM
8	Neymar	Junior	01-07-1996	555-555-562	17578	8	20250718214751	7/18/2025 9:49:44 PM
9	Paolo	Maldini	08-05-1995	555-555-563	18859	9	20250718214751	7/18/2025 9:49:44 PM
10	Andrea	Pirlo	27-09-1972	555-555-564	18169	10	20250718214751	7/18/2025 9:49:44 PM

Figure 7.11: Content of the table `Clienti_STG` after the second execution of the ETL process

ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME	FLG_NEG
9	Paolo	Maldini	08-05-1995	555-555-563	18859	9	20250718221525	7/18/2025 10:18:38 PM	0
4	Erling	Haaland	29-03-1977	555-555-558	17830	4	20250718221525	7/18/2025 10:18:38 PM	0
2	Cristiano	Ronaldo	28-03-1989	555-555-556	18320	2	20250718221525	7/18/2025 10:18:38 PM	0
10	Andrea	Pirlo	27-09-1972	555-555-564	18169	10	20250718221525	7/18/2025 10:18:38 PM	0
5	Kevin	De Bruyne	22-10-1976	555-555-559	21048	5	20250718221525	7/18/2025 10:18:38 PM	0
6	Luka	Modrić	19-08-1994	555-555-560	20446	6	20250718221525	7/18/2025 10:18:38 PM	0
1	Lionel	Messi	13-03-1989	555-555-555	19646	1	20250718221525	7/18/2025 10:18:38 PM	0
3	Kylian	Mbappé	23-01-1988	555-555-557	20340	3	20250718221525	7/18/2025 10:18:38 PM	0
8	Neymar	Junior	01-07-1996	555-555-562	17578	8	20250718221525	7/18/2025 10:18:38 PM	0
7	Virgil	van Dijk	15-08-1967	555-555-561	19045	7	20250718221525	7/18/2025 10:18:38 PM	0
24	Alessandro	Nesta	10-12-1990	555-555-570	2000	17	20250718221525	7/18/2025 10:18:38 PM	0
27	Ebenezer	Scrooge	03-03-1900	555-555-687	10234	12	20250718221525	7/18/2025 10:18:38 PM	0
30	Bruce	Wayne	04-05-1988		10000	7	20250718221525	7/18/2025 10:18:38 PM	0

Figure 7.12: Content of the table `Clienti_DLT` after the second execution

However, the `Clienti` table raises an exception during loading in Level 1, as shown in the `FLOW_MANAGER` in Figure 7.13. Just as in Level 0 of the previous run, this failure blocks the execution of the `Località` table — although it had passed the STG step successfully — because both belong to the same group (`CLI`).

IDENTITA	GRP_NAME	LIVELLO	SRC_JOBID	TRG_JOBID	STATUS	LOAD	START_DATE	END_DATE
MOVIMENTO	VEN	1	20250718221525	20250718221525	0	0	7/18/2025 10:25:21 PM	7/18/2025 10:26:18 PM
ANAGRAFICA	PAG	1	20250718221525	20250718221525	0	0	7/18/2025 10:23:48 PM	7/18/2025 10:25:07 PM
ANAGRAFICA	TEM	1	20250718221525	20250718221525	0	0	7/18/2025 10:23:17 PM	7/18/2025 10:23:34 PM
ANAGRAFICA	CLI	1	20250718221525	20250718221525	-3	2	7/18/2025 10:23:06 PM	7/18/2025 10:23:09 PM
ANAGRAFICA	PRO	1	20250718221525	20250718221525	0	0	7/18/2025 10:22:30 PM	7/18/2025 10:22:53 PM
ANAGRAFICA	TEM	0	19000101000000	20250718221525	0	0	7/18/2025 10:15:50 PM	7/18/2025 10:19:31 PM
MOVIMENTO	VEN	0	19000101000000	20250718221525	0	0	7/18/2025 10:16:38 PM	7/18/2025 10:19:20 PM
ANAGRAFICA	CLI	0	19000101000000	20250718221525	0	0	7/18/2025 10:15:33 PM	7/18/2025 10:19:02 PM
ANAGRAFICA	PRO	0	19000101000000	20250718221525	0	0	7/18/2025 10:16:02 PM	7/18/2025 10:18:28 PM
ANAGRAFICA	PAG	0	19000101000000	20250718221525	0	0	7/18/2025 10:16:15 PM	7/18/2025 10:18:05 PM
MOVIMENTO	VEN	1	20250718214751	20250718214751	0	0	7/18/2025 9:59:12 PM	7/18/2025 10:00:06 PM
ANAGRAFICA	CLI	1	20250718214751	20250718214751	0	0	7/18/2025 9:56:46 PM	7/18/2025 9:59:03 PM
ANAGRAFICA	PAG	1	20250718214751	20250718214751	0	0	7/18/2025 9:58:05 PM	7/18/2025 9:58:34 PM
ANAGRAFICA	TEM	1	20250718214751	20250718214751	0	0	7/18/2025 9:57:26 PM	7/18/2025 9:57:55 PM
ANAGRAFICA	PRO	1	20250718214751	20250718214751	0	0	7/18/2025 9:55:16 PM	7/18/2025 9:55:59 PM
ANAGRAFICA	TEM	0	19000101000000	20250718214751	0	0	7/18/2025 9:48:14 PM	7/18/2025 9:51:39 PM
MOVIMENTO	VEN	0	19000101000000	20250718214751	0	0	7/18/2025 9:49:52 PM	7/18/2025 9:51:22 PM
ANAGRAFICA	CLI	0	19000101000000	20250718214751	-3	2	7/18/2025 9:47:58 PM	7/18/2025 9:51:07 PM
ANAGRAFICA	PAG	0	19000101000000	20250718214751	0	0	7/18/2025 9:48:39 PM	7/18/2025 9:50:43 PM
ANAGRAFICA	PRO	0	19000101000000	20250718214751	0	0	7/18/2025 9:48:27 PM	7/18/2025 9:50:58 PM

Figure 7.13: Content of the `FLOW_MANAGER` after the second execution

Regarding the `Vendite` table, new data has been correctly loaded in `STG`, as shown in Figure 7.14, while some records from the previous flow have been deleted. As illustrated in the `DLT` table (Figure 7.15), newly added records have `FLG_NEG = 0`, while deleted records have `FLG_NEG = 1`.

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME
2	8	2	3	4	35	20250718221525	7/18/2025 10:16:40 PM
10	3	7	3	6	2.01	20250718221525	7/18/2025 10:16:40 PM
4	3	3	2	3	5.37	20250718221525	7/18/2025 10:16:40 PM
0	2	7	1		2.33	20250718221525	7/18/2025 10:16:40 PM
2	8	2	3	4	3.18	20250718214751	7/18/2025 9:49:54 PM
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:49:54 PM
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:49:54 PM
10	3	7	3	6	2.54	20250718214751	7/18/2025 9:49:54 PM

Figure 7.14: Content of `Vendite_STG` after the second execution.

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG
2	8	2	3	4	35	20250718221525	7/18/2025 10:19:11 PM	0
4	3	3	2	3	5.37	20250718221525	7/18/2025 10:19:11 PM	0
10	3	7	3	6	2.01	20250718221525	7/18/2025 10:19:11 PM	0
0	2	7	1		2.33	20250718221525	7/18/2025 10:19:11 PM	0
1	9	7	1	3	3.94	20250718221525	7/18/2025 10:19:11 PM	1
2	8	2	3	4	3.18	20250718221525	7/18/2025 10:19:11 PM	1
3	3	8	2	7	3.55	20250718221525	7/18/2025 10:19:11 PM	1
10	3	7	3	6	2.54	20250718221525	7/18/2025 10:19:11 PM	1
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:51:13 PM	0
2	8	2	3	4	3.18	20250718214751	7/18/2025 9:51:13 PM	0
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:51:13 PM	0
10	3	7	3	6	2.54	20250718214751	7/18/2025 9:51:13 PM	0

Figure 7.15: Content of `Vendite_DLT` after the second execution.

During the data quality checks performed in the OK step, the row identified by aggregate key `Id_Cliente = 0, Id_Prodotto = 2, Id_Tempo = 7`, and `Id_Pagamento = 1` is discarded and moved to the `ERR` table because it contains a NULL value in a non-nullable column (`Quantità`). This is shown in Figure 7.16. Before inserting the valid records from the current data flow into `Vendite_OK`, the records from the previous flow are deleted. As a result, the table represented by Figure 7.17 will contain only up-to-date data that comply with the defined constraints

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG	Descr_Err
0	2	7	1		2.33	20250718221525	7/18/2025 10:19:11 PM	0	Contiene almeno un value null...

Figure 7.16: Content of `Vendite_ERR` after the second execution

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG
1	9	7	1	3	3.94	20250718221525	7/18/2025 10:26:00 PM	1
2	8	2	3	4	35	20250718221525	7/18/2025 10:26:00 PM	0
3	3	8	2	7	3.55	20250718221525	7/18/2025 10:26:00 PM	1
4	3	3	2	3	5.37	20250718221525	7/18/2025 10:26:00 PM	0
10	3	7	3	6	2.01	20250718221525	7/18/2025 10:26:00 PM	0

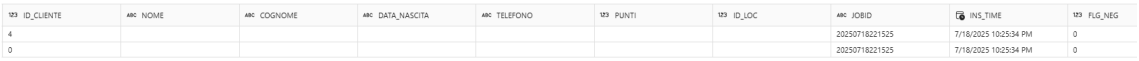
Figure 7.17: Content of `Vendite_OK` after the second execution

Finally, the valid records from `Vendite_OK` are merged into the Vendite_ODS table, portrayed by Figure 7.18

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG	UPD_JOBID	UPD_TIME
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:59:48 PM	1	20250718221525	7/18/2025 10:26:11 PM
2	8	2	3	4	35	20250718214751	7/18/2025 9:59:48 PM	0	20250718221525	7/18/2025 10:26:11 PM
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:59:48 PM	1	20250718221525	7/18/2025 10:26:11 PM
4	3	3	2	3	5.37	20250718221525	7/18/2025 10:26:00 PM	0	20250718221525	7/18/2025 10:26:11 PM
10	3	7	3	6	2.01	20250718214751	7/18/2025 9:59:48 PM	0	20250718221525	7/18/2025 10:26:11 PM

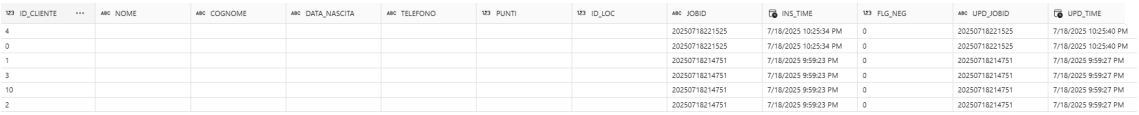
Figure 7.18: Content of `Vendite_ODS` after the second execution

As in the previous run, the `ID_Cliente` values in the `Vendite_DLT` table that are not found in `Clienti_ODS` (in this case, all of them, since `Clienti_ODS` is empty) are first inserted into `Clienti_PRE_LOAD`, with the other fields left as `NULL`, and then merged into `Clienti_ODS`, as shown in Figures 7.19 and 7.20.



ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME	FLG_NEG
4							20250718221525	7/18/2025 10:25:34 PM	0
0							20250718221525	7/18/2025 10:25:34 PM	0

Figure 7.19: Content of `Clienti_PRE_LOAD` after the second execution



ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME	FLG_NEG	UPD_JOBID	UPD_TIME
4							20250718221525	7/18/2025 10:25:34 PM	0	20250718221525	7/18/2025 10:25:40 PM
0							20250718221525	7/18/2025 10:25:34 PM	0	20250718221525	7/18/2025 10:25:40 PM
1							20250718214751	7/18/2025 9:59:23 PM	0	20250718214751	7/18/2025 9:59:27 PM
3							20250718214751	7/18/2025 9:59:23 PM	0	20250718214751	7/18/2025 9:59:27 PM
10							20250718214751	7/18/2025 9:59:23 PM	0	20250718214751	7/18/2025 9:59:27 PM
2							20250718214751	7/18/2025 9:59:23 PM	0	20250718214751	7/18/2025 9:59:27 PM

Figure 7.20: Content of `Clienti_ODS` after the second execution

7.3 Third ETL Execution – 2025-07-18 22:43:24

This execution is identified by `JOBID = 20250719224324`. The user has uploaded updated versions of the `Clienti.csv` and `Vendite.csv` files.

The new data contained in `Clienti.csv` is correctly loaded into `Clienti_STG`. As shown in Figure 7.21, only a single new record with `ID_Cliente = 30` and `JOBID = 20250719224324` is present in this execution.

ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME
30	Lebron	James	31-01-1970		57621	4	20250718224324	7/18/2025 10:44:34 PM
24	Alessandro	Nesta	10-12-1990	555-555-570	2000	17	20250718221525	7/18/2025 10:16:31 PM
27	Ebenezer	Scrooge	03-03-1900	555-555-687	10234	12	20250718221525	7/18/2025 10:16:31 PM
30	Bruce	Wayne	04-05-1988		10000	7	20250718221525	7/18/2025 10:16:31 PM
1	Lionel	Messi	13-03-1989	555-555-555	19646	1	20250718214751	7/18/2025 9:49:44 PM
2	Cristiano	Ronaldo	28-03-1989	555-555-556	18320	2	20250718214751	7/18/2025 9:49:44 PM
3	Kylian	Mbappé	23-01-1988	555-555-557	20340	3	20250718214751	7/18/2025 9:49:44 PM
4	Erling	Haaland	29-03-1977	555-555-558	17830	4	20250718214751	7/18/2025 9:49:44 PM
5	Kevin	De Bruyne	22-10-1976	555-555-559	21048	5	20250718214751	7/18/2025 9:49:44 PM
6	Luka	Modrić	19-08-1994	555-555-560	20446	6	20250718214751	7/18/2025 9:49:44 PM
7	Virgil	van Dijk	15-08-1967	555-555-561	19045	7	20250718214751	7/18/2025 9:49:44 PM
8	Neymar	Junior	01-07-1996	555-555-562	17578	8	20250718214751	7/18/2025 9:49:44 PM
9	Paolo	Maldini	08-05-1995	555-555-563	18859	9	20250718214751	7/18/2025 9:49:44 PM
10	Andrea	Pirlo	27-09-1972	555-555-564	18169	10	20250718214751	7/18/2025 9:49:44 PM

Figure 7.21: Content of `Clienti_STG` after the third execution of the ETL process

For the population of the `Clienti_DLT` table, the records are extracted from `Clienti_STG` using a `JOBID` greater than the most recent `JOBID` for which the `CLI` group successfully completed level L0. In this case, the reference `JOBID` is not the default `19000101000000`, but rather `20250718221525`, as L0 was successfully completed during the second ETL execution. These records undergo a `MINUS` operation with the records in `Clienti_STG` having `JOBID = 20250718221525` (i.e., those loaded in the previous flow). The resulting records will be labeled with the current `JOBID = 20250718221525`. As shown in Figure 7.22, the output contains

one record with `FLG_NEG = 0`, corresponding to the new `ID_Cliente = 30`, and three records with `FLG_NEG = 1`, indicating deletions from the source.

ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME	FLG_NEG
30	Lebron	James	31-01-1970		57621	4	20250718224324	7/18/2025 10:46:40 PM	0
24	Alessandro	Nesta	10-12-1990	555-555-570	2000	17	20250718224324	7/18/2025 10:46:40 PM	1
27	Ebenezer	Scrooge	03-03-1900	555-555-687	10234	12	20250718224324	7/18/2025 10:46:40 PM	1
30	Bruce	Wayne	04-05-1988		10000	7	20250718224324	7/18/2025 10:46:40 PM	1
9	Paolo	Maldini	08-05-1995	555-555-563	18859	9	20250718221525	7/18/2025 10:18:38 PM	0
4	Erling	Haaland	29-03-1977	555-555-558	17830	4	20250718221525	7/18/2025 10:18:38 PM	0
2	Cristiano	Ronaldo	28-03-1989	555-555-556	18320	2	20250718221525	7/18/2025 10:18:38 PM	0
10	Andrea	Pirlo	27-09-1972	555-555-564	18169	10	20250718221525	7/18/2025 10:18:38 PM	0
5	Kevin	De Bruyne	22-10-1976	555-555-559	21048	5	20250718221525	7/18/2025 10:18:38 PM	0
6	Luka	Modrić	19-08-1994	555-555-560	20446	6	20250718221525	7/18/2025 10:18:38 PM	0
1	Lionel	Messi	13-03-1989	555-555-555	19646	1	20250718221525	7/18/2025 10:18:38 PM	0
3	Kylian	Mbappé	23-01-1988	555-555-557	20340	3	20250718221525	7/18/2025 10:18:38 PM	0
8	Neymar	Junior	01-07-1996	555-555-562	17578	8	20250718221525	7/18/2025 10:18:38 PM	0
7	Virgil	van Dijk	15-08-1967	555-555-561	19045	7	20250718221525	7/18/2025 10:18:38 PM	0
24	Alessandro	Nesta	10-12-1990	555-555-570	2000	17	20250718221525	7/18/2025 10:18:38 PM	0
27	Ebenezer	Scrooge	03-03-1900	555-555-687	10234	12	20250718221525	7/18/2025 10:18:38 PM	0
30	Bruce	Wayne	04-05-1988		10000	7	20250718221525	7/18/2025 10:18:38 PM	0

Figure 7.22: Content of `Clienti_DLT` after the third execution of the ETL process

A special case is observed in which two records share the same `ID_Cliente = 30`. In the previous flow, the name was "Bruce Wayne", while in the current flow, it has changed to "Lebron James", indicating a modification in the source data. Since the `ROW_NUMBER` function partitions by key and orders by `JOBID` in descending order and `FLG_NEG` in ascending order, the updated record (i.e., the one with the name "Lebron James") is selected to proceed because it is labeled with `Row_Number = 1`. Furthermore, all records with a `JOBID` greater than the last `JOBID` for which the `CLI` group successfully completed level `L1` are loaded into the `Clienti_OK` table. In this case, level `L1` has never been completed for group `CLI`, so the default `JOBID = 19000101000000` is used, and all records filtered by the `ROW_NUMBER` operation are loaded into `Clienti_OK`, as shown in Figure 7.23.

It is interesting to notice how the records with `TELEFONO = None` are not discarded, this is because the `Metadata_Schema` sown in 7.2 table defines the column `TELEFONO` as nullable.

ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME	FLG_NEG
1	Lionel	Messi	13-03-1989	555-555-555	19646	1	20250718224324	7/18/2025 10:51:22 PM	0
2	Cristiano	Ronaldo	28-03-1989	555-555-556	18320	2	20250718224324	7/18/2025 10:51:22 PM	0
3	Kylian	Mbappé	23-01-1988	555-555-557	20340	3	20250718224324	7/18/2025 10:51:22 PM	0
4	Erling	Haaland	29-03-1977	555-555-558	17830	4	20250718224324	7/18/2025 10:51:22 PM	0
5	Kevin	De Bruyne	22-10-1976	555-555-559	21048	5	20250718224324	7/18/2025 10:51:22 PM	0
6	Luka	Modrić	19-08-1994	555-555-560	20446	6	20250718224324	7/18/2025 10:51:22 PM	0
7	Virgil	van Dijk	15-08-1967	555-555-561	19045	7	20250718224324	7/18/2025 10:51:22 PM	0
8	Neymar	Junior	01-07-1996	555-555-562	17578	8	20250718224324	7/18/2025 10:51:22 PM	0
9	Paolo	Maldini	08-05-1995	555-555-563	18859	9	20250718224324	7/18/2025 10:51:22 PM	0
10	Andrea	Pirlo	27-09-1972	555-555-564	18169	10	20250718224324	7/18/2025 10:51:22 PM	0
24	Alessandro	Nesta	10-12-1990	555-555-570	2000	17	20250718224324	7/18/2025 10:51:22 PM	1
27	Ebenezer	Scrooge	03-03-1900	555-555-687	10234	12	20250718224324	7/18/2025 10:51:22 PM	1
30	Lebron	James	31-01-1970		57621	4	20250718224324	7/18/2025 10:51:22 PM	0

Figure 7.23: Content of `Clienti_OK` after the third execution of the ETL process

Finally, these records are merged with the existing ones in the `Clienti_ODS` table, as highlighted in Figure 7.24. New records are inserted, and existing keys are updated. As a result, all previously incomplete records (with `NULL` attributes)

are now populated with valid data, except for the client with `ID = 0`, who remains unchanged.

ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	ID_LOC	JOBID	INS_TIME	FLG_NEG	UPD_JOBID	UPD_TIME
1	Lionel	Messi	13-03-1989	555-555-555	19646	1	20250718214751	7/18/2025 9:59:23 ...	0	20250718224324	7/18/2025 10:51:35...
2	Cristiano	Ronaldo	28-03-1989	555-555-556	18320	2	20250718214751	7/18/2025 9:59:23 ...	0	20250718224324	7/18/2025 10:51:35...
3	Kylian	Mbappé	23-01-1988	555-555-557	20340	3	20250718214751	7/18/2025 9:59:23 ...	0	20250718224324	7/18/2025 10:51:35...
4	Erling	Haaland	29-03-1977	555-555-558	17830	4	20250718221525	7/18/2025 10:25:34...	0	20250718224324	7/18/2025 10:51:35...
5	Kevin	De Bruyne	22-10-1976	555-555-559	21048	5	20250718224324	7/18/2025 10:51:22...	0	20250718224324	7/18/2025 10:51:35...
6	Luka	Modrić	19-08-1994	555-555-560	20446	6	20250718224324	7/18/2025 10:51:22...	0	20250718224324	7/18/2025 10:51:35...
7	Virgil	van Dijk	15-08-1967	555-555-561	19045	7	20250718224324	7/18/2025 10:51:22...	0	20250718224324	7/18/2025 10:51:35...
8	Neymar	Junior	01-07-1996	555-555-562	17578	8	20250718224324	7/18/2025 10:51:22...	0	20250718224324	7/18/2025 10:51:35...
9	Paolo	Maldini	08-05-1995	555-555-563	18859	9	20250718224324	7/18/2025 10:51:22...	0	20250718224324	7/18/2025 10:51:35...
10	Andrea	Pirlo	27-09-1972	555-555-564	18169	10	20250718214751	7/18/2025 9:59:23 ...	0	20250718224324	7/18/2025 10:51:35...
24	Alessandro	Nesta	10-12-1990	555-555-570	2000	17	20250718224324	7/18/2025 10:51:22...	1	20250718224324	7/18/2025 10:51:35...
27	Ebenezer	Scrooge	03-03-1900	555-555-687	10234	12	20250718224324	7/18/2025 10:51:22...	1	20250718224324	7/18/2025 10:51:35...
30	Lebron	James	31-01-1970		57621	4	20250718224324	7/18/2025 10:51:22...	0	20250718224324	7/18/2025 10:51:35...
0							20250718221525	7/18/2025 10:25:34...	0	20250718221525	7/18/2025 10:25:40...

Figure 7.24: Content of `Clienti_ODS` after the third execution of the process

The `FLOW_MANAGER` shown in Figure 7.25 illustrates that, for this execution flow, the `CLI` group has successfully completed both levels of processing. As a result, the `Località` table has also been fully loaded to the ODS layer.

IDENTITA	GRP_NAME	LIVELLO	SRC_JOBID	TRG_JOBID	STATUS	LOAD	START_DATE	END_DATE
MOVIMENTO	VEN	1	20250718224324	20250718224324	0	0	7/18/2025 10:54:37 PM	7/18/2025 10:55:22 PM
ANAGRAFICA	CLI	1	20250718224324	20250718224324	0	0	7/18/2025 10:51:05 PM	7/18/2025 10:54:27 PM
ANAGRAFICA	PAG	1	20250718224324	20250718224324	0	0	7/18/2025 10:53:24 PM	7/18/2025 10:53:37 PM
ANAGRAFICA	TEM	1	20250718224324	20250718224324	0	0	7/18/2025 10:52:54 PM	7/18/2025 10:53:14 PM
ANAGRAFICA	PRO	1	20250718224324	20250718224324	0	0	7/18/2025 10:50:29 PM	7/18/2025 10:50:52 PM
ANAGRAFICA	TEM	0	19000101000000	20250718224324	0	0	7/18/2025 10:43:49 PM	7/18/2025 10:47:25 PM
MOVIMENTO	VEN	0	19000101000000	20250718224324	0	0	7/18/2025 10:44:40 PM	7/18/2025 10:47:15 PM
ANAGRAFICA	CLI	0	19000101000000	20250718224324	0	0	7/18/2025 10:43:31 PM	7/18/2025 10:47:00 PM
ANAGRAFICA	PRO	0	19000101000000	20250718224324	0	0	7/18/2025 10:44:04 PM	7/18/2025 10:46:33 PM
ANAGRAFICA	PAG	0	19000101000000	20250718224324	0	0	7/18/2025 10:44:17 PM	7/18/2025 10:46:10 PM
MOVIMENTO	VEN	1	20250718221525	20250718221525	0	0	7/18/2025 10:25:21 PM	7/18/2025 10:26:18 PM
ANAGRAFICA	PAG	1	20250718221525	20250718221525	0	0	7/18/2025 10:23:48 PM	7/18/2025 10:25:07 PM
ANAGRAFICA	TEM	1	20250718221525	20250718221525	0	0	7/18/2025 10:23:17 PM	7/18/2025 10:23:34 PM
ANAGRAFICA	CLI	1	20250718221525	20250718221525	-3	2	7/18/2025 10:23:06 PM	7/18/2025 10:23:09 PM
ANAGRAFICA	PRO	1	20250718221525	20250718221525	0	0	7/18/2025 10:22:30 PM	7/18/2025 10:22:53 PM
ANAGRAFICA	TEM	0	19000101000000	20250718221525	0	0	7/18/2025 10:15:50 PM	7/18/2025 10:19:31 PM
MOVIMENTO	VEN	0	19000101000000	20250718221525	0	0	7/18/2025 10:16:38 PM	7/18/2025 10:19:20 PM
ANAGRAFICA	CLI	0	19000101000000	20250718221525	0	0	7/18/2025 10:15:33 PM	7/18/2025 10:19:02 PM
ANAGRAFICA	PRO	0	19000101000000	20250718221525	0	0	7/18/2025 10:16:02 PM	7/18/2025 10:18:28 PM
ANAGRAFICA	PAG	0	19000101000000	20250718221525	0	0	7/18/2025 10:16:15 PM	7/18/2025 10:18:05 PM
MOVIMENTO	VEN	1	20250718214751	20250718214751	0	0	7/18/2025 9:59:12 PM	7/18/2025 10:00:06 PM
ANAGRAFICA	CLI	1	20250718214751	20250718214751	0	0	7/18/2025 9:56:46 PM	7/18/2025 9:59:03 PM
ANAGRAFICA	PAG	1	20250718214751	20250718214751	0	0	7/18/2025 9:58:05 PM	7/18/2025 9:58:34 PM
ANAGRAFICA	TEM	1	20250718214751	20250718214751	0	0	7/18/2025 9:57:26 PM	7/18/2025 9:57:55 PM
ANAGRAFICA	PRO	1	20250718214751	20250718214751	0	0	7/18/2025 9:55:16 PM	7/18/2025 9:55:59 PM
ANAGRAFICA	TEM	0	19000101000000	20250718214751	0	0	7/18/2025 9:48:14 PM	7/18/2025 9:51:39 PM
MOVIMENTO	VEN	0	19000101000000	20250718214751	0	0	7/18/2025 9:49:52 PM	7/18/2025 9:51:22 PM
ANAGRAFICA	CLI	0	19000101000000	20250718214751	-3	2	7/18/2025 9:47:58 PM	7/18/2025 9:51:07 PM
ANAGRAFICA	PRO	0	19000101000000	20250718214751	0	0	7/18/2025 9:48:27 PM	7/18/2025 9:50:58 PM
ANAGRAFICA	PAG	0	19000101000000	20250718214751	0	0	7/18/2025 9:48:39 PM	7/18/2025 9:50:43 PM

Figure 7.25: Content of `Flow_Manager` after three executions.

It is particularly interesting to observe that the clients with `ID = 24` and `ID = 27` in Figure 7.24 have `ID_Localita` values of 17 and 12, respectively, values that are not present among the available keys in the `Localita` dimension table, Figure 7.26. Consequently, in both `Localita_PRE_LOAD` and `Localita_ODS`, we find placeholder records identified by these values, but with all other attributes set to `NULL`. This behavior is illustrated in Figures 7.27 and 7.28.

ID_LOC	INDIRIZZO	CITTA	PROVINCIA	JOBID	INS_TIME	FLG_NEG
1	Via Mercalli 1	Mantova	MN	20250718224324	7/18/2025 10:54:12 PM	0
2	Via Roma 99	Bitonto	BA	20250718224324	7/18/2025 10:54:12 PM	0
3	Viale Fiorito 18	Chiusi	SI	20250718224324	7/18/2025 10:54:12 PM	0
4	Strada Statale 99 km 55	Ferrara	FE	20250718224324	7/18/2025 10:54:12 PM	0
5	Via Viali Viola 55	Sassari	SS	20250718224324	7/18/2025 10:54:12 PM	0
6	Via Fasulla 123	Trento	TN	20250718224324	7/18/2025 10:54:12 PM	0
7	Viale Mercato 1000	Borgia	CZ	20250718224324	7/18/2025 10:54:12 PM	0
8	Via Martiri 589	Bassano del Grappa	VI	20250718224324	7/18/2025 10:54:12 PM	0
9	Viale Volta 1001	Potenza	PZ	20250718224324	7/18/2025 10:54:12 PM	0
10	Vico Quarto 50	Pompiod	AO	20250718224324	7/18/2025 10:54:12 PM	0

Figure 7.26: Content of `Località_DLT` after the third execution

ID_LOC	INDIRIZZO	CITTA	PROVINCIA	JOBID	INS_TIME	FLG_NEG
12				20250718224324	7/18/2025 10:53:56 PM	0
17				20250718224324	7/18/2025 10:53:56 PM	0

Figure 7.27: Content of `Località_PRE_LOAD` after three executions

ID_LOC	INDIRIZZO	CITTA	PROVINCIA	JOBID	INS_TIME	FLG_NEG	UPD_JOBID	UPD_TIME
1	Via Mercalli 1	Mantova	MN	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
2	Via Roma 99	Bitonto	BA	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
3	Viale Fiorito 18	Chiusi	SI	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
4	Strada Statale 99 km 55	Ferrara	FE	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
5	Via Viali Viola 55	Sassari	SS	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
6	Via Fasulla 123	Trento	TN	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
7	Viale Mercato 1000	Borgia	CZ	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
8	Via Martiri 589	Bassano del Grappa	VI	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
9	Viale Volta 1001	Potenza	PZ	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
10	Vico Quarto 50	Pompiod	AO	20250718224324	7/18/2025 10:54:12 PM	0	20250718224324	7/18/2025 10:54:20 PM
12				20250718224324	7/18/2025 10:53:56 PM	0	20250718224324	7/18/2025 10:54:20 PM
17				20250718224324	7/18/2025 10:53:56 PM	0	20250718224324	7/18/2025 10:54:20 PM

Figure 7.28: Content of `Località_ODS` after three executions

For the `Vendite` table, it is important to note that the record with `Id_Cliente = 0, Id_Prodotto = 2, Id_Tempo = 7`, and `Id_Pagamento = 1` previously discarded in the earlier execution has now been inserted by the user correctly into the `Vendite_STG` table (Figure 7.29), this time including the a value in the `Quantità` field.

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME
0	2	7	1	10	2.33	20250718224324	7/18/2025 10:44:43 PM
2	8	2	3	4	35	20250718221525	7/18/2025 10:16:40 PM
10	3	7	3	6	2.01	20250718221525	7/18/2025 10:16:40 PM
4	3	3	2	3	5.37	20250718221525	7/18/2025 10:16:40 PM
0	2	7	1		2.33	20250718221525	7/18/2025 10:16:40 PM
2	8	2	3	4	3.18	20250718214751	7/18/2025 9:49:54 PM
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:49:54 PM
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:49:54 PM
10	3	7	3	6	2.54	20250718214751	7/18/2025 9:49:54 PM

Figure 7.29: Content of `Vendite_STG` after three executions

During the execution of the `ROW_NUMBER` step in the OK phase, a temporary table is populated with both the reprocessed record (originally from `Vendite_ERR`) and the newly inserted one. Since these records share the same key, only the record with the non-null `Quantita` field proceeds due to its higher `JOBID`. As a result, only this valid record is included in `Vendite_OK`, and subsequently loaded into the `Vendite_ODS` table, as illustrated in Figure 7.30.

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG	UPD_JOBID	UPD_TIME
0	2	7	1	10	2.33	20250718224324	7/18/2025 10:55:02 PM	0	20250718224324	7/18/2025 10:55:14 PM
2	8	2	3	4	35	20250718214751	7/18/2025 9:59:48 PM	1	20250718224324	7/18/2025 10:55:14 PM
4	3	3	2	3	5.37	20250718221525	7/18/2025 10:26:00 PM	1	20250718224324	7/18/2025 10:55:14 PM
10	3	7	3	6	2.01	20250718214751	7/18/2025 9:59:48 PM	1	20250718224324	7/18/2025 10:55:14 PM
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:59:48 PM	1	20250718221525	7/18/2025 10:26:11 PM
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:59:48 PM	1	20250718221525	7/18/2025 10:26:11 PM

Figure 7.30: Content of `Vendite_ODS` after three executions

7.4 Forth and Last ETL Execution – 2025-07-18 23:18:27

The final execution is performed to demonstrate how the cleanup mechanism of the `DLT` tables explained in the previous chapter operates. Once a `DLT` contains a number of distinct `JOBIDs` equal to the defined `retention_period` - a parameter configured by the supervisor during the pipeline deployment - only the records associated with the most recent `retention_period JOBIDs` are retianed in the `DLT` table. Older records are removed from `DLT`, but this data is not lost, as it is archived in the corresponding table `DLT_HIS`.

In this case, the `retention_period` is set to 3, and the focus is placed on the `Vendite` table.

The user uploads new sales data into the lakehouse through the `Vendite.csv` file. This record, associated with `JOBID = 20250718231827`, is registered into the `Vendite_DLT` table. Figure 7.31 shows that `Vendite_DLT` now contains records from only the three distinct most recent `JOBIDs` even though four executions have occurred in this scenario. Specifically, data associated with `JOBID = 20250718214751` - the identifier of the first execution - is no longer present in `Vendite_DLT` because the automatic cleanup operation has been triggered, as per the configured retention policy. However, these records are not lost, as illustrated in Figure 7.32 the `Vendite_DLT_HIS` table retains all `DLT` records from the beginning of the case study.

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG
1	3	5	2	4	4.17	20250718231827	7/18/2025 11:21:27 PM	0
0	2	7	1	10	2.33	20250718231827	7/18/2025 11:21:27 PM	1
0	2	7	1	10	2.33	20250718224324	7/18/2025 10:47:06 PM	0
2	8	2	3	4	35	20250718224324	7/18/2025 10:47:06 PM	1
4	3	3	2	3	5.37	20250718224324	7/18/2025 10:47:06 PM	1
10	3	7	3	6	2.01	20250718224324	7/18/2025 10:47:06 PM	1
0	2	7	1		2.33	20250718224324	7/18/2025 10:47:06 PM	1
2	8	2	3	4	35	20250718221525	7/18/2025 10:19:11 PM	0
4	3	3	2	3	5.37	20250718221525	7/18/2025 10:19:11 PM	0
10	3	7	3	6	2.01	20250718221525	7/18/2025 10:19:11 PM	0
0	2	7	1		2.33	20250718221525	7/18/2025 10:19:11 PM	0
1	9	7	1	3	3.94	20250718221525	7/18/2025 10:19:11 PM	1
2	8	2	3	4	3.18	20250718221525	7/18/2025 10:19:11 PM	1
3	3	8	2	7	3.55	20250718221525	7/18/2025 10:19:11 PM	1
10	3	7	3	6	2.54	20250718221525	7/18/2025 10:19:11 PM	1

Figure 7.31: Content of `Vendite_DLT` in the last execution of the ETL process

ID_CLIENTE	ID_PRODOTTO	ID_TEMPO	ID_PAGAMENTO	QUANTITA	PREZZO	JOBID	INS_TIME	FLG_NEG
1	3	5	2	4	4.17	20250718231827	7/18/2025 11:21:27 PM	0
0	2	7	1	10	2.33	20250718231827	7/18/2025 11:21:27 PM	1
0	2	7	1	10	2.33	20250718224324	7/18/2025 10:47:06 PM	0
2	8	2	3	4	35	20250718224324	7/18/2025 10:47:06 PM	1
4	3	3	2	3	5.37	20250718224324	7/18/2025 10:47:06 PM	1
10	3	7	3	6	2.01	20250718224324	7/18/2025 10:47:06 PM	1
0	2	7	1		2.33	20250718224324	7/18/2025 10:47:06 PM	1
2	8	2	3	4	35	20250718221525	7/18/2025 10:19:11 PM	0
4	3	3	2	3	5.37	20250718221525	7/18/2025 10:19:11 PM	0
10	3	7	3	6	2.01	20250718221525	7/18/2025 10:19:11 PM	0
0	2	7	1		2.33	20250718221525	7/18/2025 10:19:11 PM	0
1	9	7	1	3	3.94	20250718221525	7/18/2025 10:19:11 PM	1
2	8	2	3	4	3.18	20250718221525	7/18/2025 10:19:11 PM	1
3	3	8	2	7	3.55	20250718221525	7/18/2025 10:19:11 PM	1
10	3	7	3	6	2.54	20250718221525	7/18/2025 10:19:11 PM	1
1	9	7	1	3	3.94	20250718214751	7/18/2025 9:51:13 PM	0
2	8	2	3	4	3.18	20250718214751	7/18/2025 9:51:13 PM	0
3	3	8	2	7	3.55	20250718214751	7/18/2025 9:51:13 PM	0
10	3	7	3	6	2.54	20250718214751	7/18/2025 9:51:13 PM	0

Figure 7.32: Content of `Vendite_DLT_HIS` in the last execution of the ETL process

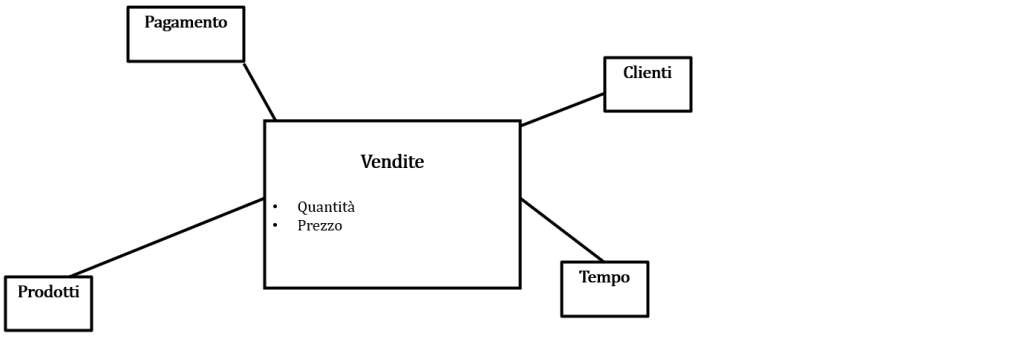
7.5 Extra: Master Data Management (MDM)

In this use case, the only dimension that references another dimension is *Località*, which references *Clienti* via the foreign key `ID_LOC`. Therefore, processing *Località* through the MDM step results in the creation of the table `Clienti_MDM`, as illustrated in Figure 7.33

ID_LOC	ID_CLIENTE	NOME	COGNOME	DATA_NASCITA	TELEFONO	PUNTI	JOBID_CLI	INS_TIME_CLI	FLG_NEG_CLI	INDIRIZZO	CITTA	PROVINCIA	JOBID_LOC	INS_TIME_LOC	FLG_NEG_LOC	UPD_JOBID	UPD_TIME
1	1	Lionel	Messi	13-03-1989	555-555-555	19646	20250723072946	7/23/2025 7:39:54 AM	0	Via Mercalli 1	Mantova	MN	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM
2	2	Cristiano	Ronaldo	28-03-1989	555-555-556	18320	20250723072946	7/23/2025 7:39:54 AM	0	Via Roma 99	Bitonto	BA	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM
3	3	Kylian	Mbappé	23-01-1988	555-555-557	20340	20250723072946	7/23/2025 7:39:54 AM	0	Viale Fiorito 18	Chiusi	SI	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM
4	4	Erling	Haaland	29-03-1977	555-555-558	17830	20250723072946	7/23/2025 7:39:54 AM	0	Strada Statale 99 km 55	Ferrara	FE	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM
5	5	Kevin	De Bruyne	22-10-1976	555-555-559	21048	20250723072946	7/23/2025 7:39:54 AM	0	Via Viali Viola 55	Sassari	SS	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM
6	6	Luka	Modrić	19-08-1984	555-555-560	20446	20250723072946	7/23/2025 7:39:54 AM	0	Via Fasulla 123	Trento	TN	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM
7	7	Virgil	van Dijk	15-08-1967	555-555-561	19045	20250723072946	7/23/2025 7:39:54 AM	0	Viale Mercato 1000	Borgia	CZ	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM
8	8	Neymar	Junior	01-07-1986	555-555-562	17578	20250723072946	7/23/2025 7:39:54 AM	0	Via Martiri 589	Bassano del Grappa	VI	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM
9	9	Paolo	Maldini	06-05-1995	555-555-563	18859	20250723072946	7/23/2025 7:39:54 AM	0	Viale Volta 1001	Potenza	PZ	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM
10	10	Andrea	Pirlo	27-09-1972	555-555-564	18169	20250723072946	7/23/2025 7:39:54 AM	0	Vico Quarto 50	Pompioni	AO	20250723072946	7/23/2025 7:43:29 AM	0	20250723072946	7/23/2025 7:43:39 AM

Figure 7.33: Content of Clienti_MDM

The Star Schema stored in the Data Warehouse is shown in Figure 7.34



Fact:
- Vendite(ID_Cliente, ID_Prodotto, ID_Tempo, ID_Pagamento, Quantità, Prezzo

Dimensions:
- Clienti(ID_Cliente, Nome, Cognome, Data_Nascita, Telefono, Punti, Indirizzo, Città, Provincia)
- Prodotti(ID_Prodotto, Nome, Quantità Disponibile)
- Tempo(ID_Tempo, Data, Mese, Anno)
- Pagamento(ID_Pagamento, Pagamento)

Figure 7.34: Use case Star schema stored in the Data Warehouse

Chapter 8

Results

In this section, the results comparing the execution performance of the three proposed alternatives in chapter 7 - Full Spark, T-SQL Stored Procedures and T-SQL + Dataflow - are presented in both tabular and graphical formats. Additionally, the Mediamente Consulting ETL framework was tested on SQL-Developer, an on-premise tool backed by Oracle Databases, to further evaluate which solution—cloud-based, with Delta Lake files (Fabric) or on-premise-database-backed (Oracle)—is more suitable for the Mediamente Consulting ETL framework. The input tables used for the test are the same as those described in the previous chapter.

To assess the behavior of each alternative under different workloads, multiple executions of the data pipeline were performed using datasets of increasing size. Specifically, the pipeline was executed with all input files containing: 100, 1,000, 10,000, 100,000, and 1,000,000 rows.

8.1 L0 performance

To gather these results, 5 executions of the ETL pipeline were performed.

L0	1st Exec.	2nd Exec.	3rd Exec.	4th Exec.	5th Exec.	Mean	STD	95% CI
100 rows	7m37s	7m19s	8m49s	7m26s	8m11s	7m52s	37s	[7m5s, 8m38s]
1,000	7m15s	9m14s	7m26s	7m40s	9m0s	8m7s	55s	[6m57s, 9m16s]
10,000	9m0s	7m42s	8m1s	8m25s	7m40s	8m9s	33s	[7m28s, 8m51s]
100,000	9m0s	7m31s	7m32s	9m9s	7m11s	8m4s	55s	[6m55s, 9m13s]
1,000,000	13m14s	11m4s	11m19s	11m59s	11m35s	11m40s	53s	[10m34s, 12m47s]

Table 8.1: Execution times (minutes, seconds) for L0 query with increasing input size, including 95% confidence interval

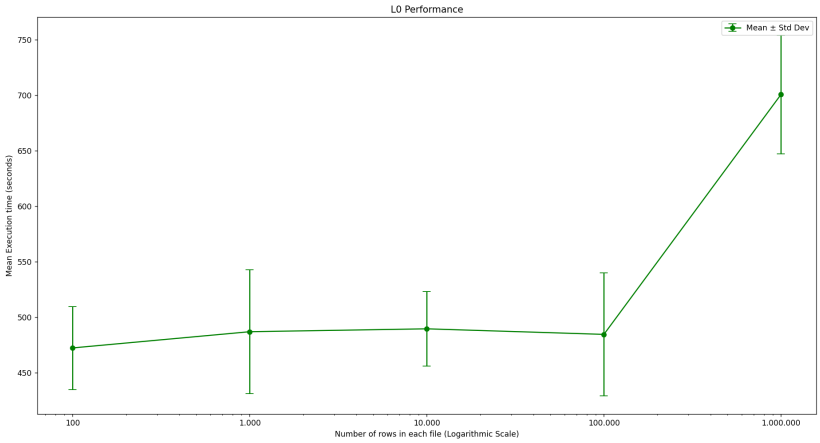


Figure 8.1: L0 execution times in function of the number of rows in each processed file

8.2 L0 Oracle-Based Performance

L0 - Oracle	1st Exec.	2nd Exec.	3rd Exec.	4th Exec.	5th Exec.	Mean	STD	95% CI
100 rows	6.48s	4.60s	4.50s	4.36s	4.91s	4.97s	0.80s	[4.00s, 5.94s]
1,000	6.35s	4.84s	5.01s	4.76s	5.14s	5.22s	0.57s	[4.51s, 5.93s]
10,000	10.52s	6.71s	6.48s	6.49s	6.93s	7.43s	1.53s	[5.93s, 8.93s]
100,000	14.36s	10.26s	10.41s	10.38s	10.23s	11.13s	1.57s	[9.59s, 12.67s]
1,000,000	44.94s	52.42s	52.57s	52.61s	52.37s	50.98s	3.26s	[46.93s, 55.03s]

Table 8.2: Execution times (seconds) for the L0 Oracle-Based implementation across increasing input size

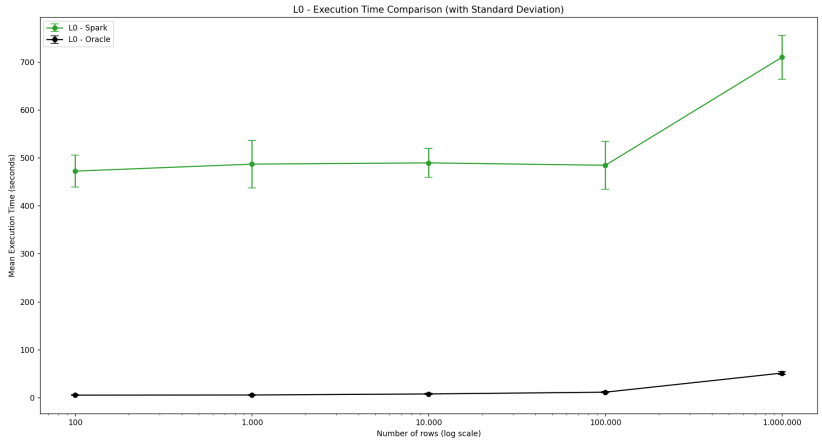


Figure 8.2: Comparison between mean execution times of the L0 Fabric-based implementation and L0 Oracle-based implementation

8.3 L1 Spark-Based Performance

L1 - Spark	1st Exec.	2nd Exec.	3rd Exec.	4th Exec.	5th Exec.	Mean	STD	95% CI
100 rows	10m9s	9m0s	8m59s	9m10s	8m45s	9m12s	32s	[8m31s, 9m53s]
1,000	12m40s	9m33s	9m21s	9m46s	10m12s	10m18s	1m21s	[8m37s, 11m59s]
10,000	9m36s	10m9s	9m0s	9m31s	9m12s	9m29s	26s	[8m56s, 10m2s]
100,000	9m31s	9m15s	8m58s	8m26s	9m47s	9m8s	27s	[8m34s, 9m42s]
1,000,000	15m0s	13m10s	14m4s	14m19s	15m13s	14m21s	48s	[13m20s, 15m21s]

Table 8.3: Execution times (minutes, seconds) for L1 (Spark implementation) with increasing input size

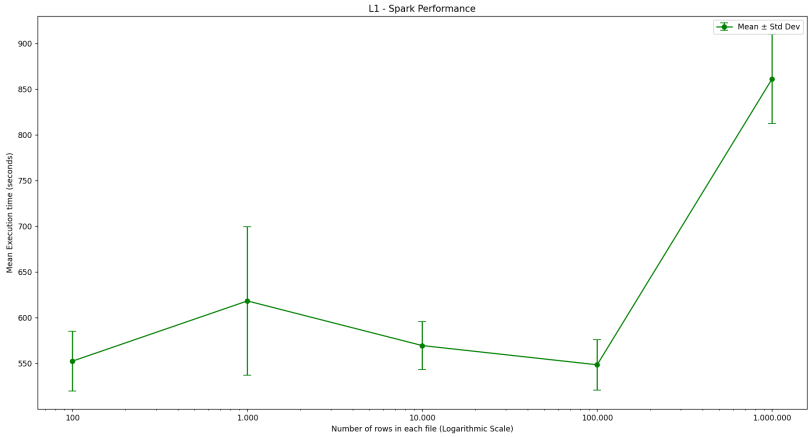


Figure 8.3: L1 - Spark-based execution times in function of the number of rows in each processed file

8.4 L1 SQL-Based Performance

L1 - T-SQL	1st Exec.	2nd Exec.	3rd Exec.	4th Exec.	5th Exec.	Mean	STD	95% CI
100 rows	7m32s	8m17s	8m41s	8m43s	8m31s	8m20s	0m29s	[7m44s, 8m57s]
1,000	7m6s	8m58s	8m41s	7m24s	8m13s	8m4s	0m48s	[7m4s, 9m4s]
10,000	8m10s	7m48s	7m15s	7m30s	7m59s	7m44s	0m22s	[7m16s, 8m11s]
100,000	9m0s	9m5s	8m58s	9m7s	8m55s	9m1s	0m4s	[8m54s, 9m7s]
1,000,000	14m59s	15m10s	14m37s	14m54s	15m13s	14m58s	0m14s	[14m40s, 15m16s]

Table 8.4: Execution times (minutes, seconds) for L1 (T-SQL implementation) with increasing input size.

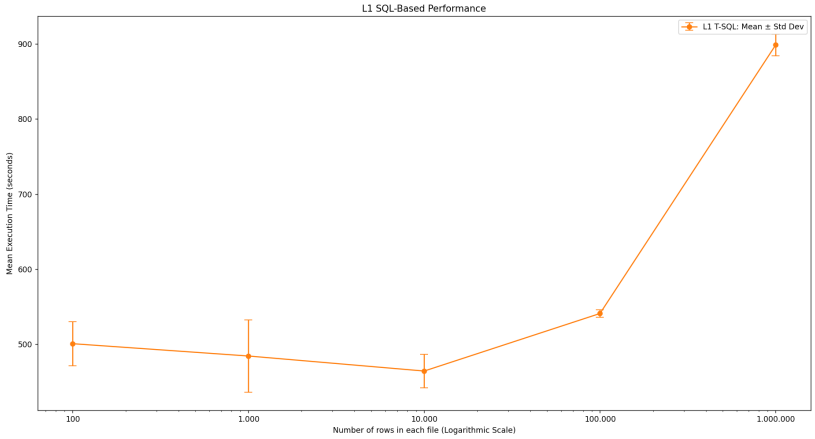


Figure 8.4: L1 - SQL-based execution times in function of the number of rows in each processed file

8.5 L1 DF-Based Performance

L1 - DF	1st Exec.	2nd Exec.	3rd Exec.	4th Exec.	5th Exec.	Mean	STD	CI 95%
100 rows	11m9s	15m11s	15m0s	14m58s	15m41s	14m23s	1m50s	[12m6s, 16m40s]
1,000	13m29s	13m14s	13m11s	13m48s	13m5s	13m21s	0m17s	[12m59s, 13m42s]
10,000	17m28s	15m42s	16m17s	17m49s	16m10s	16m41s	0m54s	[15m33s, 17m48s]
100,000	34m6s	31m28s	32m42s	36m3s	32m14s	33m18s	1m48s	[31m3s, 35m33s]
1,000,000	Fail	Fail	Fail	Fail	Fail	Fail	Fail	[Fail, Fail]

Table 8.5: Execution times (minutes, seconds) for L1 (T-SQL + Dataflow implementation) query with increasing input size

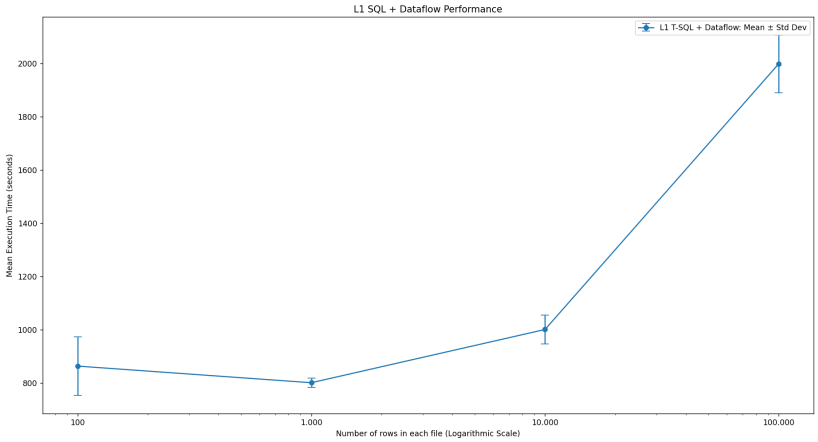


Figure 8.5: L1 - Dataflows - based execution times in function of the number of rows in each processed file

8.6 L1 Oracle-Based Performance

L1 - Oracle	1st Exec.	2nd Exec.	3rd Exec.	4th Exec.	5th Exec.	Mean	STD	95% CI
100 rows	6.7s	3.23s	3.15s	3.24s	3.95s	4.45s	1.36s	[2.78s, 6.13s]
1,000	6.42s	4.33s	4.86s	4.32s	4.91s	4.97s	0.87s	[3.73s, 6.21s]
10,000	10.24s	7.00s	7.14s	7.02s	7.35s	7.75s	1.23s	[5.72s, 8.78s]
100,000	35.706s	1m14s	1m26s	1m4s	1m6s	1m5s	16s	[49s, 1m22s]
1,000,000	2h52m	1h38m	1h44m	1h39m	1h41m	1h51m	0h29m	[1h25m, 2h17m]

Table 8.6: Execution times for L1 (Oracle implementation) with increasing input size

8.7 Comparison between the different L1 implementations

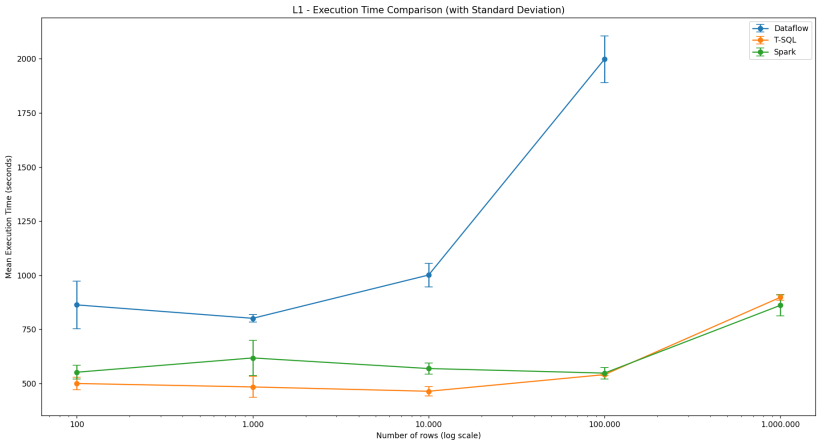


Figure 8.6: Comparison between the mean execution times of three L1 implementations

8.8 Final overall comparison between the Fabric implementations and the Oracle implementation

File Size	L0 Spark	L1 Spark	Total Fabric	L0 Oracle	L1 Oracle	Total Oracle	Δ
100	7m52s	9m12s	17m4s	4.97s	4.45s	9.42s	16m55s
1,000	8m7s	10m18s	18m25s	5.22s	4.97s	10.19s	18m15s
10,000	8m9s	9m29s	17m38s	7.43s	7.75s	15.18s	17m23s
100,000	8m4s	9m8s	17m12s	11.13s	1m5s	1m16s	15m56s
1,000,000	11m40s	14m21s	26m1s	50.98s	1h51m	1h51m51s	-1h25m50s

Table 8.7: Comparison between L0 + L1-Spark performances and L0+L1 Oracle Performance

File Size	L0 Spark	L1 SQL	Total Fabric	L0 Oracle	L1 Oracle	Total Oracle	Δ
100	7m52s	8m20s	16m12s	4.97s	4.45s	9.42s	16m3s
1,000	8m7s	8m4s	16m11s	5.22s	4.97s	10.19s	16m1s
10,000	8m9s	7m44s	15m53s	7.43s	7.75s	15.18s	15m38s
100,000	8m4s	9m1s	17m5s	11.13s	1m5s	1m16s	15m49s
1,000,000	11m40s	14m58s	26m38s	50.98s	1h51m	1h51m51s	-1h25m13s

Table 8.8: Comparison between L0 + L1-SQL performances and L0+L1 Oracle Performance

File Size	L0 Spark	L1-DF	Total Fabric	L0 Oracle	L1 Oracle	Total Oracle	Δ
100	7m52s	14m23s	22m15s	4.97s	4.45s	9.42s	22m5s
1,000	8m7s	13m21s	21m28s	5.22s	4.97s	10.19s	21m18s
10,000	8m9s	16m41s	24m50s	7.43s	7.75s	15.18s	24m35s
100,000	8m4s	33m18s	41m22s	11.13s	1m5s	1m16s	40m6s
1,000,000	11m40s	Fail	Fail	50.98s	1h51m	1h51m51s	–

Table 8.9: Comparison between L0 + L1-DF performances and L0+L1 Oracle Performance

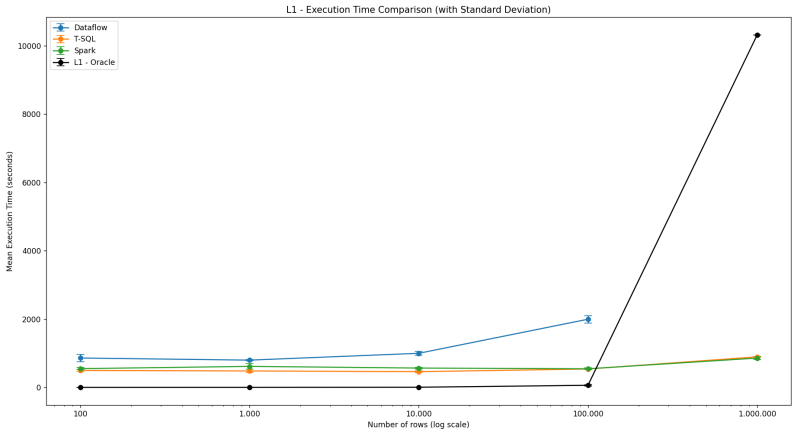


Figure 8.7: Comparison between the mean execution times of three L1 implementations, and the Oracle-based implementation

Chapter 9

Discussion of results

Table 8.1 and Figure 8.1 show that, for input sizes below 1.000.000 rows, the execution times of L0 do not exhibit a clearly increasing trend. Contrary to intuitive expectations that processing time should increase with the size of input data, the measured execution time remain relatively stable across varying inputs sizes. For instance, the mean execution times for files with size 1.000, 10.000, and 100.000 are around 8 minutes, although there is an increase in file sizes by a factor of 10. It is only when the input size reaches 1.000.000 rows that a noticeable increase in execution time occurs, with a mean of 11 minutes and 40 seconds, which is a clear deviation from the previous trend and reflects the impact of data scale on computational resources.

On the other hand, Table 8.3 and Figure 8.3 report the execution times for the Spark-based implementation of Level 1, the overall trend in performance is not strictly monotonic with respect to the input size. Surprisingly, for certain configurations, increasing the size of the input files containing 1.000 rows to those with 10.000 rows, where the mean execution time drops from 10 minutes and 18 seconds to 9 minutes and 29 seconds. The decreasing trend continues for inputs of 100.000 rows, where the execution time reaches 9 minutes and 8 seconds. Furthermore, as expected, the average execution times per number of processed rows in the L1 Spark-Based implementation are higher than those observed in Level 0. This increase is due to the greater number of operations involved in L1, including `ROW_NUMBER`, `PRE_LOAD`, `OK`, `ERR`, and `ODS` transformations. In contrast, Level 0 only performs the initial data load along with the `STG` and `DLT` steps.

The implementation of the L1 pipeline using T-SQL yields promising results. The approach shown in Figures 6.4, 6.5, and 6.6 achieves lower execution times compared to the Spark-based alternative, as also confirmed in Table 8.4, despite certain limitations such as the absence of the `MERGE` clause in the `ODS` stored procedure. Moreover, execution times obtained with T-SQL appear to be more consistent than those observed with the Python-based implementation, as indicated by lower standard

deviations.

This behavior has an explanation, in Microsoft Fabric, Spark sessions are provisioned through either starter pools or custom Spark pools offering fast startup times - typically 5-10 seconds - as soon as compute nodes are already running. However, this rapid initialization depends on shared cluster capacity availability and the pool's dynamic auto-scaling configuration. Consequently, with small datasets, the fixed overhead of session startup and task planning in Spark can outweigh actual data processing time. This results in non-linear performance, where intermediate workloads may execute more efficiently than very small workloads, then, as the input size grows, Spark has enough data to effectively leverage parallel executors, improving processing efficiency. However, as it is expected, for files with about 1,000,000 rows, due to resource contention, and memory pressure can significantly contribute to an increase in execution times. Furthermore, the higher standard deviation observed in Spark-based L1 executions may come from its multi-node distributed architecture and dynamic scheduling, which contribute to variable run-times. By contrast, T-SQL stored procedures execute within a single, highly optimized engine, resulting in much tighter runtimes across multiple executions.

Table 8.5 shows that Dataflows exhibited the highest execution times during the L1 phase of the ETL process, which in general exceeded 30 minutes for files containing 100,000 rows. When tasked with processing files with 1,000,000 rows each, the Dataflow failed entirely, as they were unable to load into local memory the content of each table.

Figure 8.6 summarizes the performance results, clearly highlighting Dataflows as the least effective solution. In contrast, T-SQL, and Spark achieved comparable execution times, with T-SQL demonstrating greater stability, as evidenced by its lower standard deviation in execution times compare to Spark

Regarding the performance of the ETL framework on SQL Developer, for file sizes smaller than 1,000,000 rows, it clearly outperformed all the implementations of both the L0 and L1 levels in Microsoft Fabric, as shown in Figures 8.2 and 8.7. Specifically, SQL Developer was able to complete the ETL process in under 2 minutes for every tested file size, whereas Fabric never managed to complete the process in less than 15 minutes, as evidenced by the mean execution times displayed in Tables 8.7, 8.8, and 8.9. However, this trend does not hold when processing files containing 1,000,000 rows. In this case, all Fabric implementations (except for the DataFrame-based one) significantly outperformed SQL Developer, as shown in Figure 8.7. In fact, for the first execution of Level 1, SQL Developer nearly reached a 3-hour runtime, as reported in Table 8.8, which is extremely slow compared to Fabric's performance. Moreover, Table 8.2 and Table 8.6 show that overall, SQL-Developer implementation is also more stable than Fabric, as demonstrated by the lower standard deviations for each file size. This is reasonable, considering that SQL Developer is an on-premise

tool where files are stored and processed locally on the hardware. Therefore, its performance is directly tied to the capabilities of the underlying infrastructure. In contrast, Microsoft Fabric is a cloud-based SaaS platform, where performance depends on the scalability and resource allocation of the cloud service.

Chapter 10

Conclusion

In conclusion, Dataflows offer a compelling solution for small organizations or teams with limited technical expertise, particularly where low-code approaches are preferred over programming languages such as Spark or T-SQL. Their graphical user interface enables users to perform data transformations without writing code. However, significant limitations arise when attempting to integrate Dataflows into a standardized and automated ETL framework, which is the core objective of this thesis.

The graphical nature of Dataflows restricts parametric configuration, especially for destination tables, which must be defined manually through the UI and cannot be dynamically controlled via M scripts. Moreover, Dataflows are atomic in structure, meaning that individual table-level traceability is not feasible unless separate Dataflows are created for each table - changing only the input and destination configuration. This approach leads to increased manual effort, reduced maintainability, and contradict the principles of re-usability and scalability essential for the development of Mediamente Consulting's ETL framework. Another major drawback is the lack of effective debugging capabilities. Frequent errors, such as those indicated by code `20302` (which aggregates various user configuration issues), produce generally generic error messages, offering no actionable insight into the root cause. Finally, Dataflows demonstrate limited scalability, particularly in scenarios involving large data volumes. The system interface becomes unresponsive under heavy loads, and in critical cases, fails to execute the transformation process entirely.

For these reasons, while Dataflows may be suitable for simple uses cases, their integration into a scalable, automated and maintainable enterprise ETL framework is not viable. Consequently, the Dataflow-based solution was discarded in favor of the SQL or Spark solutions.

Although Spark and T-SQL delivered comparable performance, with T-SQL slightly outperforming Spark in terms of execution time, performance alone is not

the sole criterion for selecting the technology to be officially integrated into the company's ETL framework. Maintainability and alignment with internal expertise are also essential factors, at Mediamente Consulting, Python based programming required for Spark is not a core competency, whereas T-SQL is a mandatory skill within the organization, in addition, the company's ETL framework developed in other Data Integration applications, like SSIS, ODI, and Workato, are developed using SQL, as part of a strategic approach to maintain a consistent and reusable ETL backbone across platforms, thus minimizing learning curves and on-boarding time. Another key aspect concerns data storage architecture. The T-SQL solution stores data directly into the enterprise Data Warehouse, a centralized and structured environment that is accessible to the client, whereas the Spark implementation outputs data into a Data Lakehouse, which is not part of the current access strategy for external stakeholders. Moreover, the T-SQL based pipeline demonstrates greater execution stability. The standard deviation of execution times observed during testing were consistently lower for T-SQL compared to Spark, indicating more predictable and stable performance. This is largely attributed to the high performance SQL engine of the Data Warehouse, as opposed to Spark where execution times can vary significantly depending on the initialization overhead of the Spark session and the dynamic provisioning of compute resources. Finally, the T-SQL-based pipeline is structurally more explicit and modular, being composed of clearly defined stored procedures, unlike the Spark implementation, where all logic is embedded within a single notebook, reducing transparency and maintainability.

For all these reasons, the T-SQL stored procedure-based solution was selected as the official implementation for the step L1 within the company's ETL framework

Nevertheless, this more effective solution becomes slower than the performance offered by SQL Developer when processing files smaller than 1,000,000 rows. This suggests that an on-premise alternative like SQL Developer may be more suitable for developing an automated ETL process. However, this is not a straightforward decision.

Being an on-premise tool, SQL Developer often requires manual configuration by users who may not be experts in database or infrastructure setup, potentially increasing deployment times. In contrast, Microsoft Fabric benefits from centralized configuration managed by the tenant administrator and offers a much more intuitive and user-friendly interface. This significantly reduces both deployment and learning times.

A major advantage of SQL Developer lies in its native integration with Oracle databases. Unlike Fabric, which relies on OneLake and Delta files as its underlying storage, SQL Developer can execute DML operations like `MERGE` and `MINUS` - which were commonly used in this project - directly and efficiently. In Fabric, these operations often require manual implementation, which increases complexity and

decreases maintainability. Moreover, Oracle databases support native enforcement of primary key constraints— a feature that simplifies operations such as `MERGE`. In Fabric, by contrast, this schema information was frequently extracted from the `Metadata_Schema` Delta Table, adding additional steps to the ETL process.

However, Fabric also has its strengths, like its seamless integration into the broader Microsoft ecosystem. For example, data models defined in the Fabric Warehouse are natively accessible in Power BI, Microsoft's business intelligence and data visualization tool.

The choice between an On-Premise Oracle Data Integration tool and Microsoft Fabric ultimately depends on the client's context. If the client requires fast and efficient data processing, a tool backed by Oracle databases—such as ODI or SQL Developer—is the preferred option. On the other hand, if the client is heavily invested in the Microsoft ecosystem, processing speed is not a primary concern, data resides in a cloud storage service such as Azure, AWS, Google Cloud..., and reporting might be part of the deliverables, then Microsoft Fabric becomes the more suitable choice.

Bibliography

[1] Fabio Duarte. *Amount of Data Created Daily (2025)*. Exploding Topics. Apr. 2025. URL: https://explodingtopics.com/blog/data-generated-per-day (cit. on p. 1).

[2] Edge Delta Team. *How Many Companies Use Cloud Computing in 2025? [10 Statistics and Insights]*. Edge Delta. May 2024. URL: https://edgedelta.com/company/blog/how-many-companies-use-cloud-computing (cit. on p. 3).

[3] Polaris Market Research. *Cloud Migration Services Market Overview*. Polaris Market Research. Feb. 2024. URL: https://www.polarismarketresearch.com/industry-analysis/cloud-migration-services-market (cit. on p. 3).

[4] W.H. Inmon. *Building the Data Warehouse*. Ed. by Robert Elliott. Third. New York, NY, United States of America: Robert Ipsen, 2002 (cit. on pp. 6–8).

[5] Margy Ross Ralph Kimball. *The Data Warehouse Toolkit*. Ed. by Robert Elliott. Second. New York, NY, United States of America: Robert Ipsen, 2002 (cit. on pp. 9, 10, 12, 13, 16–18, 35).

[6] Gerd Saurer et al. *What is Microsoft Fabric?* Microsoft. May 2025. URL: https://learn.microsoft.com/en-us/fabric/fundamentals/microsoft-fabric-overview#onelake-and-lakehouse-data-hierarchy (cit. on p. 24).

[7] Elizabeth Oldag et al. *OneLake, the OneDrive for data*. Microsoft. July 2024. URL: https://learn.microsoft.com/en-us/fabric/onelake/onelake-overview (cit. on pp. 25, 29).

[8] Gökberk Uzuntaş. *Understanding DataFlow Gen2 in Microsoft Fabric And Comparison with DataFlow Gen1*. Medium. Aug. 2024. URL: https://medium.com/@uzuntasgokberk/understanding-dataflow-gen2-in-microsoft-fabric-and-comparison-with-dataflow-gen1-18de2e547087 (cit. on p. 28).

[9] Naga Surendran. *Microsoft named a Leader in 2025 Gartner® Magic Quadrant™ for Integration Platform as a Service*. Microsoft - Gartner. June 2025. URL: https://azure.microsoft.com/en-us/blog/microsoft-named-a-leader-in-2025-gartner-magic-quadrant-for-integration-platform-as-a-service/ (cit. on p. 32).

[10] Erhard Rahm and Hong Do. «Data Cleaning: Problems and Current Approaches». In: *IEEE Data Eng. Bull.* 23 (Jan. 2000), pp. 3–13 (cit. on p. 35).