

Master’s Degree

in

Data Science

[Title]

Supervisor Candidate

Alberto Montresor Giorgia Villa

Academic Year

2021/2022

Table of Contents

[Acknowledgments 2](#_Toc113806067)

[1. Introduction 2](#_Toc113806068)

[2. Data Overview 2](#_Toc113806069)

[3. Data Integration 3](#_Toc113806070)

[3.1 Employed technologies 4](#_Toc113806071)

[3.1.1 DBeaver – PostgreSQL 4](#_Toc113806072)

[3.1.2 Talend Open Studio for Data Integration 4](#_Toc113806073)

[3.2 Data storage 6](#_Toc113806074)

[3.2.1 Tables for pollutants concentration and weather condition data 7](#_Toc113806075)

[3.2.2 Tables for monitoring stations data 8](#_Toc113806076)

[3.2.3 Tables for auxiliary operations on monitoring stations data 9](#_Toc113806077)

[3.2.4 Table for log information 10](#_Toc113806078)

[3.2.5 Table for rejected rows 10](#_Toc113806079)

[3.3 Data ingestion 11](#_Toc113806080)

[3.3.1 Data cleaning and data quality 11](#_Toc113806081)

[3.3.2 Slowly Changing Dimension (SCD) 12](#_Toc113806082)

[3.3.3 ETL implementation in Talend Open Studio 14](#_Toc113806083)

[3.3.4 Data reliability over time 26](#_Toc113806084)

[4. Data Analysis and Visualization 27](#_Toc113806085)

[5. Conclusions and future work 27](#_Toc113806086)

[Appendix A 28](#_Toc113806087)

[Appendix B 30](#_Toc113806088)

[References 35](#_Toc113806089)

Acknowledgments

1. Introduction

2. Data Overview

3. Data Integration

The first part of this thesis was focused on the design and implementation of a data integration system to ingest, store, and ultimately update the data previously described.

Data integration usually refers to the challenge of establishing an architecture capable of providing the final user with a comprehensive view of data stored on heterogenous resources [1]. These resources constitute the first components of a data integration system, which then follows to construct a mapping between those sources and the final unified view. This final view acquires the name of global schema. The global schema can be then queried by the user to retrieve subsets a/o aggregated forms of the original view to answer specific user-intended questions [1]. Because of this, data integration has become a crucial aspect of management and coordination of enterprises, as it allows not only to gain actual information and analytics from raw data, but also to centralize and unify data from activities of different departments a/o business units. This lightens the job of higher-level management, as well as enhancing productivity and efficiency at the lower levels of the organisation [2].

Nevertheless, data integration comes with its own issues and challenges which can be both social and technical. From a social point of view, some data owners may be reluctant to allow their data to be used in integrated platforms, particularly if said platforms are directly managed by a third party. From a technical perspective, one of the main issues is the low interoperability between data sources and their semantic heterogeneity. Indeed, each data source may have its own protocols, schemas, and formats, making the establishment of the above-mentioned global schema a non-trivial task [3]. Finally, different data sources may employ a different naming system for the same kind of data, and inconsistencies or missing entries could arise once sources are combined. This is usually solved by applying data transformations and data cleaning procedures to the data; however, these procedures can be cumbersome to implement flawlessly, especially when working with considerable amounts of data which can hardly be inspected [1], as in the case of this thesis: the data employed in this work was mainly stored in large CSV files with a magnitude of millions of rows of data per CSV, reaching a maximum of 61.5 million of rows occupying 3.5GB in the largest CSV. As the size of the data source did not allow to inspect the data beforehand, it first had to be ingested and stored exploiting a primitive version of data integration system; once stored, it was then possible to investigate the data thanks to the use of queries to assess the necessity of data cleaning procedures. As it was indeed the case, data had to be ingested and stored a second time through an improved integration system which performed data cleaning at ingestion time. This underlines the fact that even though inarguably useful, data integration systems can be challenging to design and may need to be upgraded over time.

Considering the three main components of a data integration system (i.e., data sources, mapping, and global schema), in the concrete scenario of this work they can be materialized as follows: data sources can be described as an heterogeneous set of CSV files and APIs; the mapping between these and the global schema is implemented by an ETL pipeline developed with a dedicated software named Talend Open Studio, which is specific for data integration purposes; the global schema can be regarded as the set of tables stored on the Database Management System (DBMS) DBeaver, which functions as a rudimental data warehouse.

The following subsections include an exhaustive explanation of the design of the implemented data integration system. First, the employed tools and technologies will be described; then the data storage solutions will be presented; finally, a detailed description of the implementation of the ETL pipeline for data ingestion will be provided, together with some brief references to the designed protocols to enforce data integrity.

3.1 Employed technologies

In order to implement the data integration system, two main technologies were employed: Talend Open Studio for Data Integration and the database tool DBeaver. These were compulsory requirements provided by the contractor, as they are two widely used tools inside their organisation. More information on both technologies is provided below.

3.1.1 DBeaver – PostgreSQL

DBeaver is an open-source multiplatform database tool for both development and database administration purposes [4]. It is free of charge and supports many of the main DBMSs with JDBC or ODBC such as MySQL, Oracle, and PostgreSQL. It also supports some DBMSs which do not make use of xDBC drivers such as Redis and MongoDB. Among other features, it provides users with a graphical user interface and with the possibility to generate and export visualizations of database schemas and objects [5]. For this thesis, DBeaver was employed to work with PostgreSQL DBMS, which is an open-source relational DBMS based on the SQL language [6]. DBeaver and PostgreSQL were adopted in this work to implement the data storage stage of the data integration system.

3.1.2 Talend Open Studio for Data Integration

Talend Open Studio is an open-source software specifically designed for data integration, data management, and data quality purposes. It is a Java compiler providing the user with a graphical interface, where the user can design their own ETL pipelines through intuitive and code-free components. These components are usually connected forming a pipeline, which in Talend is named a job, functioning as a basic executable. Once a job is saved, Talend automatically compiles it, generating a Java class, which can then be integrated into other processes [7]. Figure 1 shows an example of Talend job. The icons represent the different components, while the arrows connecting them generate the flow of the pipeline. The light blue shaded areas correspond to the different subjobs the main job is composed by.

Diagram

Description automatically generated with low confidence

*Figure 1*: Example of Talend Job

Talend Open Studio was exploited to implement the main part of the data integration system, that is the data ingestion, data cleaning, and data transformation phases. For a better understanding of how these phases were developed, in Table 1 a list of the used Talend components and their functionalities have been provided [8].

|  |  |  |
| --- | --- | --- |
| Icon | Name | Description |
|  | tCreateTemporaryFile | Generates a temporary file in the desired location. |
|  | tDBClose | Closes an open connection to a database. |
|  | tDBCommit | Commits changes on an active database connection. Eventually closes the connection. |
|  | tDBConnection | Opens a connection to a database schema. |
|  | tDBInput | Exploits an open connection to load a database table row-by-row into the workspace. |
|  | tDBOutput | Exploits an open connection to perform row-by-row insert, delete, or update operations on a database table. |
|  | tDBOutputBulkExec | Reads and parses a delimited file, generates a modified file of the original, and exploits an open connection to perform a bulk insert operation on a database. |
|  | tDBRow | Executes any provided SQL query. |
|  | tExtractJSONFields | Reads fields of a Json file resulting from an API or HTTP request. |
|  | tFileDelete | Deletes file(s) in the specified location corresponding to the provided file mask. |
|  | tFileFetch | Retrieves a file from the provided ULR, saving it in the desired location. |
|  | tFileInputDelimited | Loads and parse row-by-row a delimited file. |
|  | tFileList | Iterates over all files a/o folders inside a specified location corresponding to the provided file mask. |
|  | tFileOutputDelimited | Creates or updates row-by-row a delimited file. |
|  | tFileUnarchive | Unzips the specified file in the desired location. |
|  | tFilterRow | Filters rows according to a personalised criterion. |
|  | tFixedFlowInput | Generates rows from the provided variables. |
|  | tFlowMeter | Saves flow information at runtime. |
|  | tFlowMeterCatcher | Catches and returns information stored in a tFlowMeter. |
|  | tFlowToIterate | Receives a list of files and parse them one at the time. |
|  | tHttpRequest | Sends an http request using the provided URL. |
| A picture containing diagram  Description automatically generated | tJava | Executes any provided Java code. |
|  | tJavaFlex | Executes any provided Java code for each row in the flow. |
|  | tMap | Loads data from one or more sources, then allows to manipulate those data before sending the transformed data to one or more outputs. |
|  | tPostjob | Generates a subjob to be executed after the main job has finished. |
|  | tPrejob | Generates a subjob to be executed after the main job has started. |
|  | tRest | Sends a post, get, or delete request to the provided API. |
|  | tRunJob | When triggered, runs the specified job. Useful to run jobs in sequence. |
| Graphical user interface, application  Description automatically generated | tSetGlobalVar | Sets the name and value of variables accessible from any component. |
|  | tUniqRow | Filters between unique and duplicated rows. |

*Table 1*: Employed Talend components

3.2 Data storage

As a first step to construct the required data integration system, a decision on data storage had to be made in order to find the most suitable solution considering both the type of data sources and the technologies available to the contractor.

In general, data storage refers to where and how information and data are recorded and retained on a digital system [9]. In also includes all relevant technologies capable of maintaining data stable over time and ensuring its easy accessibility, these being the two main goals of a data storage system [10]. Indeed, both human users and computers employ storage devices to save useful data for future interpolation, making it primary to these devices to ensure a certain level of protection against data loss and an intuitive and straightforward access to those data [11].

However, being able to provide such qualities has become more and more complicated as the volume and variety of data has increased exponentially over the last decades. Because of this trend, new challenges have arisen in the field of data storage, as traditional storage systems tend to easily become obsolete [12]. Data representation is one of these challenges. As a matter of fact, data can now be collected from a considerable number of different sources, each possibly providing data with different schemas, data types, granularity, and semantics. This is true not only for sources yielding data with different nature, but also for those storing identical data. For example, given a sensor registering data on hourly basis, two different sources may collect those data using a similar, yet not identical data type (e.g. float instead of double for decimal numbers), may adopt slightly different naming for the same data field (e.g. “Date” instead of “DateTime” for a field referring to a date object), or even change the granularity of data through the use of aggregations (e.g. aggregating hourly data to obtain daily information instead of maintaining the original hourly precision provided by the sensor) [12]. All these differences may cause issues when building a storage system capable of managing data from different sources, as discrepancies should be removed to provide a consistent interpretation and representation of data: failing to achieve this may in fact result in incorrect information on those data. This is directly connected to another challenge data storage systems are facing, that is data redundancy: when storing data from different sources, it can be the case that two sources, while collecting data on different phenomenon, may share a portion of data. Storing twice the data present in both sources adds complexity to the storage system without apporting any new information, and therefore should be avoided. However, recognizing which data is truly redundant may be cumbersome, again for differences in naming, data types and aggregation level of the data itself [12]. One additional challenge is connected to the concept of data consistency. Data consistency is a desirable property of storage systems as it implies that when data is queried it is always up to date. However, when handling data that may change over a period of time or that requires later validation (as in the case of some data collected for this work), data consistency can only be achieved by a scheduled revision of the stored data against the one in the source, in order to decide which data should be inserted, deleted, or updated [12]. Finally, one last challenge is represented by the variety of data formats, that can hardly be stored in traditional relational DBMS, which can only accept structured table-like data and can only be scaled vertically by increasing the power of the used server at increasing costs. Hence the need of non-relational DBMS that can store also semi-structured and unstructured data and can be easily scaled horizontally [13].

In the context of this work, it was decided to opt for the relational DBMS DBeaver. This, having verified that data from the different sources all exhibited a tabular structure and considered the requirements in terms of technologies stated by the contractor. Issues connected to data representation and consistency were also encountered. However, since those were handled at ingestion time, a more punctual explanation on these matters will be provided in the next dedicated section.

In the database, a total of 38 tables were created and populated, some to save the ingested data and some as support tables. More precisely:

* 17 tables were created to store pollutants concentration data,
* 15 tables were created to store weather conditions data,
* 2 tables were created to store monitoring stations data,
* 2 tables were created as auxiliary tables for the monitoring stations tables,
* 1 table was created to handle rejects during the ingestion stage,
* 1 table was created to log all relevant operations performed by the pipeline.

In Appendix A a summary table reporting the precise naming, number of rows and size for each of these tables and an overview of the database structure can be found.

For a more accurate insight on the structure of the tables, the following paragraphs provide the DDL of each kind of table, together with a description of the table fields and screenshot of the table view in the database.

3.2.1 Tables for pollutants concentration and weather condition data

Tables storing pollutant concentration and weather conditions present an analogous structure as they were all populated from similar CSV files. They are named *sens\_data\_####* and *weather\_data\_####* respectively, where #### corresponds to the year the data refer to. They all exhibit the following schema and fields [14]:

* **Idsensore** (INT8): unique identifier assigned to a sensor.
* **Dataora** (TIMESTAMP): date and time of value acquisition.
* **Valore** (FLOAT8): value registered by the sensor; if value is -9999, the record is missing or invalid.
* **Stato** (VARCHAR2): either VA (valid) or NA (not valid). As some entries may contain non-certain values, records can be validated after acquisition up until the end of March of the year following the registration date. Hence the need of this field.
* **Idoperatore** (INT4): either 1 (value in *valore* is the mean value registered in the timespan), 3 (value in *valore* is the maximum value registered in the timespan), or 4 (value in *valore* is the cumulative value registered in the timespan).

Figure 2 shows a screenshot of how the data appears in the database.

Table

Description automatically generated

*Figure 2*: Screenshot of table sens\_data\_1995

3.2.2 Tables for monitoring stations data

The two tables storing information on monitoring stations for pollutants and weather conditions are named *stations\_sens* and *stations\_weather* respectively. They were populated by ingesting data from two APIs with similar structures, but some naming differences that were handled at ingestion time. These tables also store a history to address the issue of Slowly Changing Dimension. However, as some records appeared to be corrupted, some additional operations had to be performed on both tables to rectified them. Their schemas and fields are as follows [15]:

* **Idsensore** (INT8): unique identifier assigned to a sensor.
* **Nometiposensore** (VARCHAR): kind of pollutant or weather condition measured by the sensor.
* **Unitamisura** (VARCHAR): unit of measure of the registered phenomenon.
* **Idstazione** (INT8): unique identifier assigned to the monitoring station.
* **Nomestazione** (VARCHAR): extended name assigned to the monitoring station.
* **Quota** (INT8): altitude at which the monitoring station is located.
* **Provincia** (VARCHAR): Italian province in which the monitoring station is located.
* **Comune** (VARCHAR): Italian municipality in which the monitoring station is located.
* **Storico** (VARCHAR2): either S (entry is active, i.e., represents the present state of the monitoring station) or N (entry is part of the history, i.e., represents a past state of the monitoring station).
* **Datastart** (TIMESTAMP): date and time corresponding to when the record became active.
* **Datastop** (TIMESTAMP): date and time corresponding to when the record became inactive. If it is set to 9999-12-31, the record is still active.
* **Utm\_nord** (INT8): value of the north coordinate in the Universal Transverse Mercator coordinate system.
* **Utm\_est** (INT8): value of the east coordinate in the Universal Transverse Mercator coordinate system.
* **Lat** (FLOAT8): latitude at which the monitoring station is located.
* **Lng** (FLOAT8): longitude at which the monitoring station is located.
* **Coordinate** (VARCHAR): compound string with latitude and longitude.

Figure 3 shows a screenshot of the data representation in the database.

Table

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

*Figure 3*: Screenshot of table stations\_sens

3.2.3 Tables for auxiliary operations on monitoring stations data

As stated above, tables storing monitoring stations data presented some inconsistencies in the implementation of the history and had to be rectified. To achieve this, two auxiliary tables were created to store intermediate data used in the cleaning operations of the pipeline. The two tables are named *stations\_check\_datastart* and *stations\_check\_datastop*, andrespectively collect information on the field *datastart* and *datastop* in the monitoring stations tables. They are populated at ingestion time to perform some rectifying operations. Their schema is as follows:

* **Idsensore** (INT8): unique identifier assigned to the sensor presenting inconsistencies i.e., a record missing either the *datastart* or *datastop* field values.
* **Prima\_acquisizione** / **ultima\_acquisizione** (TIMESTAMP): actual value of field *datastart* or *datastop* for the record, obtained through operations in the data integration pipeline.
* **Table\_name** (VARCHAR): either *stations\_sens* or *stations\_weather*, indicates which table the corrupted record comes from.

Figure 4 provides a view of how the tables appear in the database.

Text

Description automatically generated Text

Description automatically generated

*Figure 4*: Screenshot of table stations\_check\_datastart and stations\_check\_datastop

3.2.4 Table for log information

The table *log* was created to store information on the main operations performed by the ingestion pipeline. It is populated at ingestion time each time a Talend job has ended. Its schema is the following:

* **Current\_date** (TIMESTAMP): date and time the operation was performed.
* **Table\_name** (VARCHAR): table on which the operation was performed.
* **Operation** (VARCHAR): kind of performed operation.
* **Number\_of\_rows** (INT8): number of rows interested in the operation.
* **Duration\_in\_secods** (INT8): duration of the operation in seconds.

In Figure 5 a snapshot of the first rows of said table can be found.

Graphical user interface

Description automatically generated with medium confidence

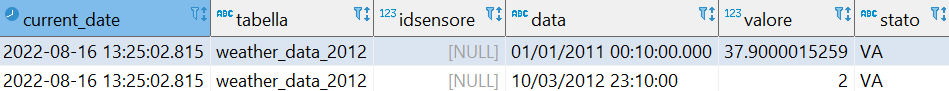
*Figure 5*: Screenshot of table logs

3.2.5 Table for rejected rows

A table *rejected* was created to handle eventual rows that could not be parsed during ingestion operations in order to monitor their fields and statutes. The schema of such table is as follows:

* **Current\_date** (TIMESTAMP): date and time the record was inserted into the table.
* **Tabella** (VARCHAR): which table the unparsable record comes from.
* **Idsensore** (INT8): *idsensore* value of the unparsable record.
* **Data** (VARCHAR): *data* value of the unparsable record.
* **Valore** (FLOAT8): *valore* value of the unparsable record.
* **Stato** (VARCHAR): *stato* value of the unparsable record.
* **Idoperatore** (INT8): *idoperatore* value of the unparsable record.
* **Errormessage** (VARCHAR): text of the error fired by Talend while parsing the record.

Figure 6 shows a screenshot of how the data appears in the database.



Graphical user interface, text

Description automatically generated

*Figure 6*: Screenshot of table rejects

3.3 Data ingestion

After having arranged the data storage, the next logical step was to design and implement the ingestion stage of the data integration system. This was achieved by exploiting the functionalities of Talend Open Studio, to create an ETL pipeline while fulfilling the requirements of the contractor. This stage possibly corresponds to the most important part of the whole data integration system, as it is when the data storage system is populated, but also when data quality and data cleaning operations are performed.

In general, ETL processes refer to all those Extraction, Transformation, and Loading procedures aiming to: 1) identify and retrieve data from different sources; 2) clean and manipulate those data into a desired format, in order to obtain some relevant information for future analysis; 3) finally load the transformed data into a data storage system (usually a data warehouse) [16], [17]. Given the complexity and the cost of developing and maintaining ETL procedures, a proper modelling of such processes is necessary. Nevertheless, no conceptual framework has been indicated as best practice yet [16], in particular when it comes to how the ETL processes should be structured in order to grant qualities such as data integrity and data quality. This possibly because ETL pipelines normally work with non-homogeneous data formats, ambiguous semantics, and data structures that may change over time, which contribute to delay processes related to data quality and integrity. This poses some additional challenges to the already complex design of integration pipelines [18].

Issues related to data quality and integrity were encountered also while designing the ETL pipeline for this work. They are briefly discussed in the first two following subsections, presenting respectively data quality concerns and the problem of Slowly Changing Dimension for data integrity. The last subsection provides instead some insights on the logic behind the implemented ETL pipeline.

3.3.1 Data cleaning and data quality

In order to summarize the different data quality issues encountered, the schematization presented in Table 1 in [18] will be taken as proxy. They delineate a categorization of quality issues according to two factors: the different stages of the ETL pipeline they can be found, and whether they relate to the schema or the instance of the data. Table 2 below, recall the structure of the proposed Table, while adapting it to the use case of this work.

|  |  |  |
| --- | --- | --- |
| **Stage** | **Problem** | **Description** |
| Extract | Embedded values | Values of different fields aggregated in one.  *Example*: lat, lng, utm\_nord and utm\_est values of monitoring stations were stored together as a string ‘coordinates’ in the data source. The resolution was to split the entries while ingesting. |
| Duplicate records | Repeated entries.  *Example*: duplicates were found in pollutants and weather conditions source CSVs. Some of them were due to the Daylight-Saving Time (DST) change in hour and could be handled, other remained unresolved and had to be discarded. |
| Missing values | Fields appearing as NULL or empty.  *Example*: missing values in numeric fields in source CSVs are set to -9999 by the data provider. If not handled, these could be considered as valid numbers during the analysis, strongly biasing results. The resolution was to keep them in the Database, and to exclude them later from the analysis using filters. |
| Transform | Variety of data types | Fields having different data types in the data source with respect to the schema specified in the storage system.  *Example*: data ingested from APIs was saved as String type, independently of the kind of data. The resolution was to manually cast data to the correct data type during ingestion. |
| Structural conflicts | Data type (usually DateTime) has different syntax in the source and in the storage system.  *Example*: DateTime fields in source CSVs had format dd/MM/yyyy HH:mm:ss, while DBeaver default Timestamp format is yyyy-MM-dd HH:mm:ss.SSS. The resolution was to convert the Date field from the CSV to the correct DB format while ingesting. |
| Load | Wrong implementation of the SCD | History of the data can be incomplete a/o conflicted.  *Example*: in the history implemented by the monitoring stations APIs some records were missing the StartDate, while some entries were closed without EndDate. The resolution was to perform some rectifying operations at ingestion time, as explained in the following section. |

*Table 2*: Encountered data quality issues

3.3.2 Slowly Changing Dimension (SCD)

While for some table (usually fact tables) loading processes may only include insert operations of newly acquired data, for other tables (generally dimensions tables) records can be updated, either sporadically or in a systematic way [19]. How to handle those updates is also part of the design decisions concerning ETL processes.

Dimensions that are expected to change over time are referred to as Slowly Changing Dimensions (SCDs). This term was first introduced by [20], who also proposed several options to deal with such matter. They all exploit the fact that only few dimensions may change, hence allowing to maintain an independent dimensional structure where only some minor modifications are needed [20]. At a higher level, two alternative solutions can be pursued: either register the change auditing for past values or replacing old values losing the history of changes [19]. At a lower level, four main types of SCDs can be delineated: Type 0, 1, 2, 3, and 4, each employing a different history storage solution [16]:

* SCD of Type 0 does not account for changes nor for history keeping: data is stored as it was first extracted and is never modified, even though source data may have changed [20].
* SCD of Type 1 accounts for changes in the data by overwriting existing values with the most recent ones [20]. This strategy has the advantage of not occupying additional storage and not needing any significant design. However, it does not allow to implement an history, as old data are merely overwritten [19].
* SCD of Type 2 accounts for both data updates and history keeping. This strategy foresees the insertion of a new row each time an update happens [19]. However, by simply adding the new row, there would be no way of knowing which one contains the updated data, as both the new and old record will share the same key. Hence, to implement this strategy three more attributes should be added to the table: StartDate, storing the date and time when the row was first inserted into the table; EndDate, storing date and time when some changes were registered for any of the attributes of the record (hence when a new row with the same key is added to the table), and a flag IsCurrent being True if the record corresponds to the most recent version of the data, or False otherwise [16]. Therefore, any current record will have a StartDate, EndDate to NULL, and IsCurrent to True. Once a new record with the same key is added to the table, first the existing record will have its EndDate set to the insertion date and its IsCurrent flag set to False, indicating that the record is now part of the history. Then the new record will be inserted into the table, having StartDate as the current date, EndDate to NULL, and flag IsCurrent to True. It should however be mentioned that besides being one of the most used types of SCD, this strategy may not be scalable if dimensions are changing rapidly, as the number of rows will increase consequently [21].
* SCD of Type 3 also accounts for changes in the data and history keeping. However, unlike Type 2, a register is kept only for the most recent update [21]. History is implemented by adding a new column each time a dimension is changed, storing the past value for that dimension. For instance, let’s assume that Employee 1 changed their mansion from Development to Testing; then the corresponding record 1 in table Employees undergoes a change in the field *Mansion*, where the value “Development” is overwritten to “Testing”. At the same time, a new column *PreviousMansion* will be created, being NULL for any record but record 1, where it will assume value “Development”. If instead another employee changed their marital status, then another column *PreviousMaritalStatus* will be created, and the procedure will be repeated. Possibly a *ChangeDate* dimension could be added as well. Besides being a possible solution, it is evident how this strategy may not be scalable if several changing dimensions are present [21]: the number of columns will increase accordingly, generating a sparse table, which is quite costly to store given its several blank fields.
* SCD of Type 4 was introduced to solve the scalability issue of SCD Type 2 [21]. According to this solution, a new “Mini-Dimension” table is created for each highly changing dimension, where such dimension and its history are stored as in Type 2 [19], while records in the main table can simply be overwritten. This reduces the volume of the main table. Mini-Dimensions tables are linked to the main table through keys, which ensures the possibility to perform queries and analysis also exploiting historical data [21].

Cases of SCDs were encountered while implementing the commissioned data integration system, specifically of Type 1 and 2. A Type 1 strategy was adopted for data extracted from APIs concerning the current year data for pollutants concentration and the current month data for weather conditions. Indeed, these APIs not only store newly daily acquired data, but may also contain modified versions of previous days data that have been validated (i.e., modified) by the ARPA data provider. Since data from APIs is ingested daily, then these modifications (usually on attributes *Valore* and *Stato*) were handled as a case of SCD. In particular, as data from APIs is ingested in delta and updated a/o deleted records are merely overwritten, the adopted logic was a case of SCD of Type 1.

On the other hand, the APIs storing dimensional data on monitoring stations already implement a SCD strategy of Type 2, where attribute StartDate is named Datastart, attribute EndDate is named Datastop, and flag IsCurrent is named Storico and acquires values N (True) or S (False). Moreover, for current valid records, the EndDate field is set to the fictitious date 9999-12-31 00:00:00 instead of NULL, because of some compatibility issues in parsing NULL values between Talend Open Studio and DBeaver. Finally, even though the strategy was already implemented in the APIs, some tests were run to verify the integrity of the history and indeed some inconstancies were discovered: some records were missing the StartDate, while some others had the IsCurrent flag set to False, but EndDate still equal to NULL. Such discrepancies were handled at ingestion time, through a series of iterations and queries that will be better explained in the next section.

3.3.3 ETL implementation in Talend Open Studio

As previously stated, the ETL part of the implemented data integration system was developed using the open-source tool Talend Open Studio. The chosen approach can be possibly reconnected to a warehousing integration approach, as data is extracted from sources and loaded in the database, which functions as a materialized static view of the source data [3].

The pipeline is orchestrated into 10 different Talend jobs that can be categorized as:

* 3 utilities jobs: splitCSV, CombinedSplitCsv, FinalBulk;
* 2 jobs for ingesting monitoring stations data from APIs: sensorsStations\_SQLQuery, weatherStations\_SQLQuery;
* 2 jobs for ingesting historical data from CSV files for pollutants concentration and weather conditions: bulkInsertS, bulkInsertW;
* 2 jobs for ingesting current month/year weather conditions data from API: ingestion2022MonthW, ingestion2022W
* 1 job for ingesting current year pollutants data from API: ingestion2022S.

The logic and design of each of these jobs is described in detail in the following subsections.

*splitCSV*

This job is an auxiliary subjob for ingesting historical data from source CSVs. It takes as inputs a directory path where the source CSVs are stored and a file mask corresponding to a regex of the names of the files to read, with the purpose of splitting the original CSV in smaller-sized CSVs. It also filters out unparsable records and logs information on such CSV and eventual rejects. The operation of partitioning source files was added to the original pipeline to avoid heap space errors at parsing time. Indeed, in a first version of the pipeline, source CSVs were ingested without partitioning. However, given the size of those CVSs and considering how Talend loads such files (row-by-row) 8GB of RAM had to be allocated on the machine for this process only. Since it occurred that not every machine may have 16+ GB available, it was decided to split files before applying any transformation a/o insertion operations. Hence, the necessity of this job.

The logic behind the job is illustrated in Figure N, while a snapshot of the Talend job is provided in Appendix B.

Essentially, the job starts by initializing a global variable RejectsCount to zero. This variable stores the number of unparsable rows found in the process and is updated every time a reject is encountered. This information is needed to construct the record to be inserted into the *log* table at the end of the job. Then, the connection to the DB instance managing the data storage system is established. After, all files in the indicated directory path and corresponding to the specified file mask are parsed and their fields are casted to the required data types. If a row cannot be parsed or a field cannot be casted, an entry with the row fields plus some information on the parsing error is generated and an insert operation is performed on the *rejects* table; also, the RejectsCount variable is incremented by 1. Rows not constituting any issue are instead inserted into an output CSV with row limit set to 15.000.000. Each time this limit is reached, a new CSV file is generated, creating a partition of the original CSV. The design limit was established after some trials, as it was the maximum number of rows that could fully exploit Talend loading capacity without overloading its RAM usage. Once all CSVs have been parsed and partitioned, if the variable RejectsCount is greater than zero, a record is generated whose values are retrieved from job specifications and from the RejectsCount variable. The obtained record is then inserted into table *log* on the database, the operation is committed, and the DB connection is closed. If instead, the RejectsCount variable has still value zero (hence no rejects information should be logged), the connection is simply closed without performing any logging operation.

Diagram, schematic

Description automatically generated

*Figure* *N*: SplitCSV job logic

*CombinedSplitCsv*

This job is another auxiliary subjob for ingesting historical data from source CSVs. It takes as inputs a directory path where the partitioned CSVs are stored and a file mask corresponding to a regex of the names of the files to read. Its purpose is basically to revert the work of job splitCSV, i.e., to generate a single file combining all partitioned CSVs of a given source CSV. However, the job does not work with the partitioned CSV produced by the splitCSV job, but with the modified version of those files produced during the process of jobs bulkInsertS and bulkInsertW. This reverting operation was necessary in order to perform later integrity checks between the data stored into the database and the data stored in source CSVs. Indeed, integrity checks were designed to randomly sample a subset of rows from the database on which to run the checks. However, since these rows could potentially be stored in any of the partitioned CSVs and loading each of these CSVs into the environment one by one was found to be extremely costly, it was decided to recombine together the modified versions of the partitions to load a single file into the environment, this being a much less expensive operation to perform.

The logic behind this job is illustrated in Figure N, while a snapshot of the Talend job is provided in Appendix B.

In practice, once the job starts all files in the indicated directory path and corresponding to the specified file mask are parsed and appended to a single CSV file with a user-defined naming. Finally, once all modified partitioned CSVs have been appended, all modified partitions are deleted from the file system, to avoid maintaining unnecessary heavy files.

Diagram

Description automatically generated

*Figure* *N*: ComineSplitCSV job logic

*FinalBulk*

This the last auxiliary subjob for ingesting historical data from source CSVs. It is run at the end of the ingestion process of one source CSV with the purpose of generating the relative log record and cleaning unnecessary files generated during the ingestion.

Figure N below shows the logic behind the job execution, while a snapshot of the Talend job is provided in Appendix B.

At the beginning of the execution, the connection to the database instance managing the storage system is established. Then the usual record with logging fields is generated. Since this is a child job, values for these fields are not extracted from the jobs specification but are inherited from the parent job. The obtained record is then inserted into table *log* on the database. After this operation is concluded, a second operation starts, looping over the folder where the partitioned CSVs generated in job splitCSV are stored. This loop deletes all CSVs corresponding to the specified file mask. Once the folder is emptied, all operations are committed to the database and the connection is closed.

Diagram

Description automatically generated

*Figure* *N*: FinalBulk job logic

*SensorsStations\_SQLQuery and weatherStations\_SQLQuery*

These jobs process the ingestion of information on monitoring stations for pollutants concentration and weather conditions. Each job retrieves the data in JSON format from the respective ARPA api, extracts and casts the fields, finally inserting the records in the correct database table (either *stations\_sens* or *stations\_weather*). This ingestion is performed in delta, as only very few records are expected to change, making an ingestion in full not efficient. Also, as anticipated in Section N, the APIs already implement a SCD strategy of Type 2, which however, was found to have some inconsistencies. These inconsistencies are handled in these jobs.

First a get request is send to the API to retrieve the data stored as JSON. Since data are returned as a single string, fields have to be first extracted from the main string in order to be casted. In the meanwhile, data from tables *stations\_check\_datastart*, *stations\_check\_datastop*, and *stations\_sens* (or *stations\_weather*) are loaded into the environment. Once these operations are concluded a left outer join is performed, having as left table the data extracted from the API and as right tables the two *stations\_checks*. This join is convenient as the *stations\_checks* tables store inconsistent records present in the API that were already handled in previous runs of the job, with their correct fields values: to address only new discrepancies in the history and avoid cleaning the same records at each job run, records in the API whose key is present in the *stations\_checks* tables are updated with their consistent values, before initializing the delta ingestion process.

The ingestion in delta is designed as an inner join having as left table the data from the previous steps and as right table data from table *stations\_sens* (or *stations\_weather*). Since the API implements a SCD of Type 2, no delete operation should be present, hence only update and insert are handled by the job. Updated records are indicated as those resulting from the join operation with some differences in non-key fields between API data and database table data. New records are instead the rejects of the inner join, as this implies their key is present in the API, but not in the database. Update and insert operations are performed on the *stations\_sens* (or *stations\_weather*) table, while two counters are updated, respectively keeping track of how many records with inconsistencies in the datastart and datastop fields are present.

If one of these two counters has values greater than zero, then the connected cleaning procedure is initiated. Let’s consider the case of discrepancies in the datastart field. First the table *stations\_check\_datastart* is populated with any record from the result of the lookup exhibiting this kind of inconsistency (i.e., datastart values set to NULL). Then a loop over all the tables with a given mask name (such as ‘sens\_data\_%’ or ‘weather\_data\_%’) is initiated. In the loop, query in Figure N (A) is executed on each table. This query simply selects for each sensor ID in table *stations\_check\_datastart* the least recent date registered for each year of data. The result of the query is inserted in table *stations\_check\_datastart*. Then query in Figure N (B) is executed to return for each sensor ID the least recent date among those present in *stations\_check\_datastart*. This returns the absolute least recent date among all years for each record key, which is indeed the correct datastart value for that record. These values are finally updated in table *stations\_sens* (or *stations\_weather*).

Text

Description automatically generatedText

Description automatically generated

1. (B)

*Figure N*: datastart cleaning query

To correct the issues of the datastop field the same procedure is applied, with the difference that table *stations\_check\_datastop* is used, and that the most recent date is selected each time (so max() instead of min() is executed in the query). Finally, a log row is generated with all relevant information on the job to be inserted into table *log*.

The logic behind this job is illustrated in Figure N, while a snapshot of the Talend job is provided in Appendix B.

Diagram

Description automatically generated

*Figure* *N (A)*: Stations\_SQLQuery job logic

Diagram

Description automatically generated

*Figure* *N (B)*: Stations\_SQLQuery job logic

*BulkInsertS and bulkInsertW*

These two jobs are responsible for the ingestion of historical data stored in CSV files of pollutants concentration data and weather conditions data. They accept as input the location in which the CSVs are stored, and they aim to insert the data into the designed database tables, after having parsed and casted the data, as well as removed duplicates. They also solve an issue related to the Daylight-Saving Time (DST) change: even though the data did present some actual duplicated values (possibly because of double registration by sensors), the majority of the encountered duplicates was found to be related to how DBeaver handles the DTS change day. Indeed, for each record whose date was the day of DST change for a given year and whose time was 2:00 a.m., some intrinsic process in DBeaver will automatically change the time of that record to 3:00 a.m., thus generating a duplicate (as a record with the same date and 3:00 a.m. time already existed). To avoid triggering said functionality of DBeaver, the solution was to set the time of the interested records to 1:59 a.m.. To make this process as autonomous as possible, a function had to be implemented to calculate the DST change day starting from the year. The body of this function can be found in Appendix C. Finally, it is also worth to mention that historical data were ingested in full using a bulk insert instead of a regular insert operation, as given the amount of data, using a bulk insert provided much faster performances.

In practice, the jobs initialize a first loop iterating over all source CSVs corresponding to the provided file mask in the designated folder. Each of these original files is first passed as input to the child job *splitCSV* to be partitioned in smaller-sized CSVs. Then, a connection to the database instance is established, while another loop is prompted. This second loop first parses each of the partitions, removing duplicates and handling DST records issues. After, it modifies the obtained data in such a way to be suitable for bulk insertion, generating a new modified version of the partitions. Then, this is final version is ingested and changes are committed. As a last step of this inner loop, the job *combinedSplitCsv* is called, to merge together the modified versions of the partitions for later integrity checks purposes. Once all partitions for a given source CSV have been ingested, the outer loop closes the connection to the database, calculates the execution time for logging, and run the child job *finalBulk* to delete the already parsed partitions and insert the record corresponding to the current operation in the log table. Once all the original CSVs have been inserted, the outer loop ends and so do the jobs.

The two jobs *bulkInsertS* and *bulkInsertW* share the same logic, with the only difference being the source directory from which to load the CSVs and the database tables where data are stored (either *sens\_data\_####* or *weather\_data\_####*). One last difference is that in general, no operation is applied to the database tables before inserting the data, as tables are expected to be empty since data are being ingested for the first time. However, if the job *bulkInsertW* is run as a child job of *ingestion2022W* to ingest current year data, then the interested table is first truncated, and then data is inserted. This is because in that case, the table will be already full as the job is not performing an ingestion of historical records, but rather an update of current year data.

The logic behind both jobs is illustrated in Figure N, while a snapshot of the two Talend jobs is provided in Appendix B.

Diagram, schematic

Description automatically generated

\*weather\_data\_###

Also, if the job bulkInsertW is called as child  job of  ingestion2022W,

then before the insert operation the table is truncated.

In any other case, no action on the table is executed.

*Figure* *N*: ingestion historical data jobs logic

*Ingestion2022S and ingestion2022MonthW*

These jobs handle the ingestion in delta of current year data for pollutants concentration and weather conditions. These data are stored on two different APIs with identical structure which are updated daily with new records registered during the day. The only difference is that the API for pollutants concentration data stores current year records from the 1st of January, while the one for weather conditions data only stores current month records and it is emptied every 1st day of the month. The remaining records on weather conditions for the current year are instead stored in a CSV updated every 45 days on the ARPA website. This difference in the management of the APIs is probably due to the considerably higher volume of meteorological data.

The purpose of these jobs is to implement an ingestion in delta of data present on the APIs, using the APIs as master sources and the database tables *sens\_data\_2022* or *weather\_data\_2022* as targets. Data are ingested in delta, because the APIs not only provide new entries for the current day, but also update a/o delete past records that have been validated by the data provider. However, since only a relatively small portion of the data is expected to change, ingesting in full was regarded as too expensive. Nevertheless, some preliminary tests were run on the pollutants concentration API to evaluate the possibility of retrieving only data from the most recent months from the API, instead of all records starting from the 1st of January. This to lighten the burden on the ETL pipeline. Unfortunately, the tests did not provide any strong evidence that by doing so some updates or delete operations on least recent records would not be lost. Indeed, a test run during the first week of July proved that records related to January data were still being modified after six months. Hence, all data present on the pollutants API had to be loaded.

The jobs start by calculating the DST day change for the current year with the function provided in Appendix C. Then a temporary file is generated to store data retrieved from the API. This because these data have to be loaded twice into the environment and it is substantially less expensive to load them from a file rather than extract them from the API. So, data are retrieved from the API, fields are extracted from the JSON row and casted to the correct data type. The issue related to DST is also addressed. The modified data are stored into the temporary file and from there are loaded one first time into the environment to together with records in database table (either *sens\_data\_2022* or *weather\_data\_2022*). Then an inner join is executed, having as left the temporary file and as right the data from the database table. This join allows to catch inserted and updated records. Updates are indicated as those entries resulting from the join whose value in the API data and database data differ for at least one non-key field. Inserts are instead the rejects of the inner join. Both insert and update operations of the identified rows are executed on the database table.

After this first operation has ended, data from the temporary file and the database table (either *sens\_data\_2022* or *weather\_data\_2022*) are loaded once again into the environment to construct a second inner join, this time having the database table as left and the data from the temporary file (i.e., from the API) as right. In this way, delete operations are detected as the result of the join and the database table is leveled to the source API. Once this operation has ended, entries are constructed with all relevant logging information and are inserted into table *log*. Finally, all operations are committed, and the job ends.

Both jobs *ingestion2022S* and *ingestion2022MonthW* are scheduled to run daily.

The logic shared by jobs *Ingestion2022S* and *ingestion2022MonthW* is illustrated in Figure N, while a snapshot of the two Talend pipelines is provided in Appendix B.

Diagram

Description automatically generated

*Figure* *N*: ingestion current year data jobs logic

*Ingestion2022W*

This last job is responsible for orchestrating the proper insertion and update of current year meteorological data. Indeed, because of how the data sources are structured, more attention had to be made to not lose any change in the weather condition data. As already mentioned, current year weather data are updated in two modalities: current month data are uploaded and updated on the API, while data from months prior to the current one is stored into a CSV which is updated every 45 days with a new validated version of the data. While API data are scheduled to be ingested daily, it was decided to execute the more extensive ingestion of the CSV data on monthly basis, precisely on the 15th of each month. Hence, the purpose of this job is to manage the execution of these two concurrent ingestion processes.

The execution starts by calculating the current day of the month. In case it is the half of the month, a flow is initiated to retrieve the most updated versions of current year data of prior months: first, the newest version of the current year CSV is downloaded from the ARPA website; then the file is unzipped and passed as input to job *bulkInsertW*. Once the child job has ended, the zip archive is deleted from the folder to avoid storing unnecessary files. If instead, it is not the half of the month, job *ingestion2022MonthW* is run as a child job, to ingest current month data from the API.

The job logic is illustrated in Figure N, while a snapshot of its rendering in Talend is provided in Appendix B.

Chart

Description automatically generated with low confidence

*Figure* *N*: ingestion2022W jobs logic

3.3.4 Data reliability over time

One last relevant aspect that was investigated while implementing the data integration pipeline was the reliability of ingested data. Data reliability is indeed a fundamental quality to ensure as it allows to trust the data for developing analysis and any other business operation [22].

The challenge that was encountered in the specific case of this work was how to provide data reliability not only right after ingestion, but also over time: even though data updates for the current year are managed by the ETL pipeline, past years data could also be updated sometime later, because of how the data validation process works by the ARPA agency. Hence, there was the need to implement a simple procedure to be run yearly to check for changes in the data source and to eventually update the database tables. According to the procedure, in case some changes are detected, the interested database table is truncated, and data are re-ingested in full using jobs *bulkInsertS* and *bulkInsertW*. Even though this may appear inconvenient as most records probably will not undergo any changes, given the specifics of Talend Open Studio and the mole of data to parse, after some tests it was decided that the most performing solution was indeed to execute an ingestion in full of the updated historical CSVs.

The procedure was implemented using some straightforward Python functions which can be divided into three main categories: PostgreSQL related functions, pickles generation functions, and integrity functions. Some more details are provided in Table N.

|  |  |  |
| --- | --- | --- |
| Category | Function Name | Description |
| PostgreSQL functions | postgresConnection | Inputs: None.  Opens connection to Postgres instance managing the data storage. |
| closeConnection | Inputs: connection, cursor.  Closes connection to Postgres instance managing the data storage. |
| getTable | Inputs: year, category, cursor.  Executes a “SELECT \* FROM ...” query on database table storing data of category (either pollutants or weather) of the specified year. |
| Pickle generation functions | generateDf | Inputs: database table.  Generates a dataframe of the provided DB table. |
| generatePickle | Inputs: dataframe, year, category.  Generates and saves a pickle file of the provided dataframe. |
| generatePickles | Inputs: year [opt], category.  Generates pickle files for data in Postgres DB tables from year 1995 to present (or for one specific table if parameter ‘year’ is specified). |
| Integrity check functions | getSortedCsv | Inputs: year, category.  Returns the source CSV corresponding to the specified year sorted by fields 'dataora' and 'idsensore'. |
| getSortedDbTable | Inputs: year, category.  Returns the DB table corresponding to the specified year sorted by fields 'dataora' and 'idsensore'. |
| checkSize | Inputs: sorted sourceCsvFile, sorted DbTable.  Checks whether the source CSV file and the mirrored DB table have equal number of records. |
| checkConsistency | Inputs: sorted sourceCsvFile, sorted DbTable.  Takes 100.000 random records from the source csv file and DB table and compare their fields values to assert their consistency. |
| checkIntegrity | Inputs: year, category.  Runs integrity checks between the source CSV file and mirrored DB table by calling functions checkSize and checkRecordConsistency. |

*Table N*: Python integrity functions overview

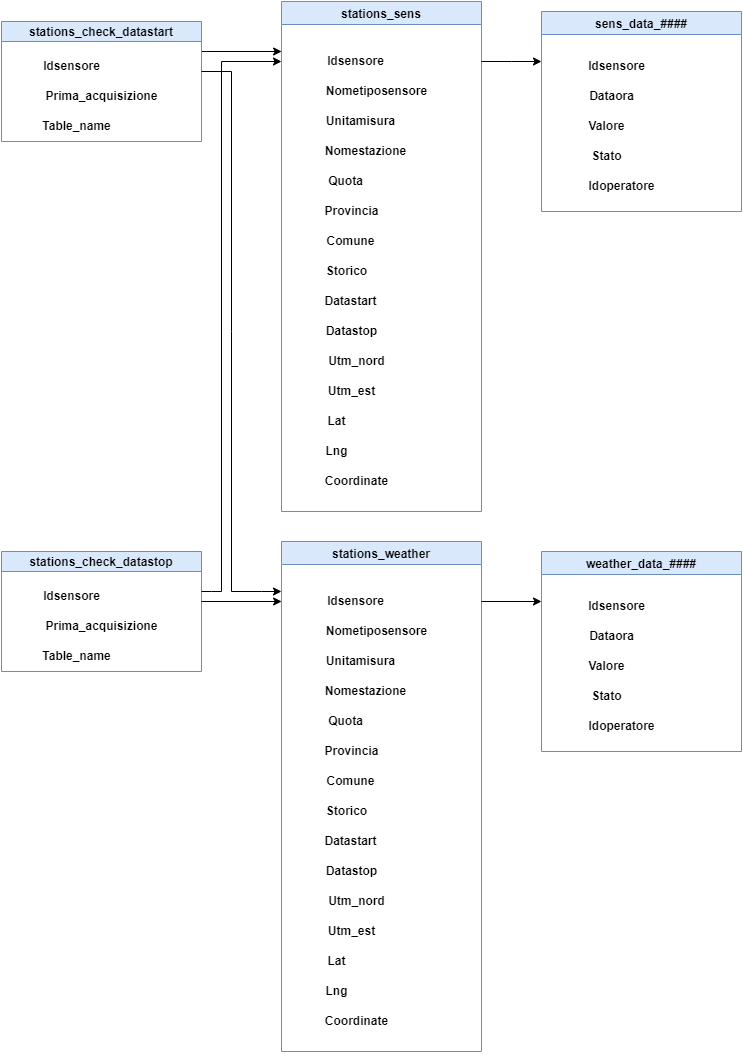
4. Data Analysis and Visualization

5. Conclusions and future work

* Use nosql db with tables bc it is more scalable
* Possible split tables to be one per year
* Put scheduled jobs on a server and not on a pc
* Connect Dashboard directly to the database for data
* Metti file check integrity su talend per automatizzare il tutto

Appendix A

**Database Overview**

****

*Figure 1*: Database tables overview and relations

|  |  |  |
| --- | --- | --- |
| Name | Number of rows | Size |
| logs\* | 63 | 48 KB |
| rejected\* | 3 | 16 KB |
| sens\_data\_1995 | 11,489,984 | 660 MB |
| sens\_data\_2000 | 14,492,734 | 832 MB |
| sens\_data\_2004 | 15,113,853 | 868 MB |
| sens\_data\_2007 | 13,277,946 | 762 MB |
| sens\_data\_2010 | 14,775,586 | 849 MB |
| sens\_data\_2011 | 4,972,158 | 285 MB |
| sens\_data\_2012 | 4,920,972 | 282 MB |
| sens\_data\_2013 | 4,913,671 | 282 MB |
| sens\_data\_2014 | 4,282,894 | 246 MB |
| sens\_data\_2015 | 4,270,563 | 245 MB |
| sens\_data\_2016 | 4,270,696 | 245 MB |
| sens\_data\_2017 | 2,595,465 | 149 MB |
| sens\_data\_2018 | 2,582,803 | 148 MB |
| sens\_data\_2019 | 2,574,002 | 147 MB |
| sens\_data\_2020 | 2,620,280 | 150 MB |
| sens\_data\_2021 | 2680340 | 154 MB |
| sens\_data\_2022\* | 1,714,296 | 112 MB |
| stations\_check\_datastart | 14 | 16 KB |
| stations\_check\_datastop | 27 | 56 KB |
| stations\_sens | 970 | 304 KB |
| stations\_weather | 1,262 | 344 KB |
| weather\_data\_2000 | 13,317,906 | 756 MB |
| weather\_data\_2005 | 39,455,657 | 2.2 GB |
| weather\_data\_2008 | 43,987,469 | 2.5 GB |
| weather\_data\_2010 | 977,107 | 56 MB |
| weather\_data\_2012 | 58,229,801 | 3.3 GB |
| weather\_data\_2013 | 42,497,020 | 2.4 GB |
| weather\_data\_2014 | 49,195,352 | 2.8 GB |
| weather\_data\_2015 | 57,877,143 | 3.2 GB |
| weather\_data\_2016 | 55,107,307 | 3.1 GB |
| weather\_data\_2017 | 56,682,542 | 3.2 GB |
| weather\_data\_2018 | 59,920,834 | 3.4 GB |
| weather\_data\_2019 | 60,126,516 | 3.4 GB |
| weather\_data\_2020 | 61,503,272 | 3.5 GB |
| weather\_data\_2021 | 59,929,021 | 3.4 GB |
| weather\_data\_2022\* | 36,922,546 | 2.1 GB |

\*Number of rows and size may increase over time as tables are updated on daily basis

*Table 1*: Information on database tables

Appendix B

**Talend Jobs**

Diagram

Description automatically generated

*Figure 1:* splitCSV job



*Figure 2:* combinedSplitCsv job

Graphical user interface, text, application, chat or text message

Description automatically generated

*Figure 3:* finalBulk job

A picture containing diagram

Description automatically generated

*Figure 4:* sensorsStations\_SQLQuery job

A picture containing chart

Description automatically generated

*Figure 5:* weatherStations\_SQLQuery job

Background pattern

Description automatically generated

*Figure 6:* bulkInsertS job

Chart

Description automatically generated with medium confidence

*Figure 7:* bulkInsertW job

Diagram

Description automatically generated with low confidence *Figure 8:* ingestion2022W job

Chart

Description automatically generated A picture containing chart

Description automatically generated

*Figure 9:* ingestion2022MonthW job

Chart

Description automatically generated with medium confidence Chart

Description automatically generated with medium confidence

*Figure 10:* ingestion2022S job

References

|  |  |
| --- | --- |
| [1] | M. Lenzerini, “Data Integration: A Theoretical Perspective,” in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2002. |
| [2] | A. Halevy, A. Rajaraman and J. Ordille, “Data Integration: The Teenage Years,” in *Proceedings of the 32nd international conference on Very large databases*, 2006. |
| [3] | S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset and P. Senellart, “Data Integration,” in *Web Data Management*, Cambridge University Press, 2011, pp. 193-222. |
| [4] | DBeaver, “About: DBeaver Community,” [Online]. Available: https://dbeaver.io/about/. |
| [5] | “What is DBeaver,” 15 March 2018. [Online]. Available: https://database.guide/what-is-dbeaver/. |
| [6] | “About: PostgreSQL,” [Online]. Available: https://www.postgresql.org/about/. |
| [7] | O. Mbaabu, “Introduction to Data Integration in Talend Open Studio,” Section, 2020. [Online]. Available: https://www.section.io/engineering-education/introduction-to-data-integration-in-tos/#:~:text=Talend%20Open%20Studio%20is%20an%20innovative%20software%20application%20that%20provides,effective%20and%20easy%20to%20use.. [Accessed 15 08 2022]. |
| [8] | Talend Inc., “Talend Components Reference Guide 7.3.1M6,” 17 01 2020. [Online]. Available: http://download-mirror1.talend.com/tosbd/user-guide-download/V731M6/TalendOpenStudio\_Components\_RG\_EN\_7.3.1M6.pdf. [Accessed 03 07 2022]. |
| [9] | R. Tiwari, “Data Storage vs Database,” Linkedin, 13 10 2021. [Online]. Available: https://www.linkedin.com/pulse/storage-vs-database-rupesh-tiwari?trk=articles\_directory. [Accessed 10 07 2022]. |
| [10] | Hwelett Packard Enterprise Development LP, “What is Data Storage?,” [Online]. Available: https://www.hpe.com/us/en/what-is/data-storage.html#:~:text=Data%20storage%20is%20the%20retention,using%20computers%20or%20other%20devices. [Accessed 10 07 2022]. |
| [11] | IBM, “What is data storage?,” [Online]. Available: https://www.ibm.com/topics/data-storage. [Accessed 10 07 2022]. |
| [12] | J. RubyDinakar and S. Vagdevi, “A Study on Storage Mechanisms for Heterogeneous Sensors data on Big data Paradigm,” in *2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*, 2017. |
| [13] | S. Palanisamy and P. SuvithaVani, “A survey on RDBMS and NoSQL Databases MySQL vs MongoDB,” in *2020 International Conference on Computer Communication and Informatics (ICCCI), 2020*, 2020. |
| [14] | ARPA Regione Lombardia, “Dati sensori aria,” 2022. [Online]. Available: https://www.dati.lombardia.it/Ambiente/Dati-sensori-aria/nicp-bhqi. [Accessed 11 07 2022]. |
| [15] | ARPA Regione Lombardia, “Stazioni Meteorlogiche,” 2022. [Online]. Available: https://www.dati.lombardia.it/Ambiente/Stazioni-Meteorologiche/nf78-nj6b. [Accessed 10 07 2022]. |
| [16] | J. Awiti, A. A. Vaisman and E. Zimányi, “Design and implementation of ETL processes using BPMN and relational algebra,” *Data & Knowledge Engineering,* vol. 129, no. 101837, 2020. |
| [17] | R. Wijaya and P. B., “An overview and implementation of extraction-transformation-loading (ETL) process in data warehouse (Case study: Department of agriculture),” in *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, 2015. |
| [18] | M. Souibguia, F. Atiguib, S. Zammalia, S. Cherfib and S. Ben Yahiaa, “Data quality in ETL process: A preliminary study,” in *23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems*, 2019. |
| [19] | A. U. Sepulveda, R. C. Loyola and M. W. Hernandez, “Performance comparison slowly changing dimensions using model relational and object-relational,” in *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, 2015. |
| [20] | R. Kimball and M. Ross, The Data Warehouse Toolkit Second Edition, New York: John Wiley and Sons, Inc., 2002. |
| [21] | D. Asanka, “Implementing Slowly Changing Dimensions (SCDs) in Data Warehouses,” 3 September 2021. [Online]. Available: https://www.sqlshack.com/implementing-slowly-changing-dimensions-scds-in-data-warehouses/. [Accessed 25 07 2022]. |
| [22] | Talend, “What is Data Integrity and Why Is It Important?,” Talend, [Online]. Available: https://www.talend.com/resources/what-is-data-integrity/. [Accessed 05 08 2022]. |

J. Sreemathy, S. Priyadharshini, K. Radha, K. Sangeerna and G. Nivetha, "Data Validation in ETL Using TALEND," 2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS), 201hgh9, pp. 1183-1186, doi: 10.1109/ICACCS.2019.8728420.