FLOWER: Viewing Data Flow in ER Diagrams

Elijah Mitchell[1], Nabila Berkani[2], Ladjel Bellatreche[3], Carlos Ordonez[1]

[1] University of Houston, USA
[2] Ecole nationale Supérieure d'Informatique, Algiers, Algeria
[3] ISAE-ENSMA, Poitiers, France

Abstract. In data science, data pre-processing and data exploration require various convoluted steps such as creating variables, merging data sets, filtering records, value transformation, value replacement and normalization. By analyzing the source code behind analytic pipelines, it is possible to infer the nature of how data objects are used and related to each other. To the best of our knowledge, there is scarce research on analyzing data science source code to provide a data-centric view. On the other hand, two important diagrams have proven to be essential to manage database and software development projects: (1) Entity-Relationship (ER) diagrams (to understand data structure and data interrelationships) and (2) flow diagrams (to capture the main processing steps). These two diagrams have historically been used separately, complementing each other. In this work, we defend the idea that these two diagrams should be combined in a unified view of data pre-processing and data exploration. Heeding such motivation, we propose a hybrid diagram called FLOWER (FLOW+ER) that combines modern UML notation with data flow symbols, in order to understand complex data pipelines embedded in source code (most commonly Python). The goal of FLOWER is to assist data scientists by providing a reverse-engineered analytic view, with a data-centric angle. We present a preliminary demonstration of the concept of FLOWER, where it is incorporated into a prototype that traces a representative data pipeline and automatically builds a diagram capturing data relationships and data flow.

1 Introduction

Data pre-processing is a time-consuming and difficult step in Big Data Analytics (BDA); However, it remains essential in cleaning and normalizing data in order to make analysis possible. The difficulties of this step arise from both the data and the processes transforming them. It is important to mention that in BDA, data often comes from diverse sources with different structures and formats. The accuracy of the final analysis strongly depends on the processes applied to this data, which include data cleaning, transformation, normalization, and so on. These processes conducted by the data analyst typically generate many intermediate files and tables in an often disorganized manner. By deeply analyzing these processes from an explainability point of view, we realize that easy understanding is limited to whatever documentation exists and the memory of the analyst authoring them. This represents a real obstacle to deobfuscating BDA systems.

In the context of BDA, data entities and transformations must be considered together to facilitate the understanding of the whole pre-processing pipeline. Several closely related works on ER diagrams [1], [2], Flow diagrams [3], and data transformation [4], [5] have been done by other researchers. In this work, we propose a hybrid diagram called FLOWER (FLOW + ER) which combines data structure and the flows of a given BDA project. The goal is to assist big data analysts in data pre-processing by examining existing pipelines for data flow relationships. We illustrate its utility by demonstrating real tools that can automatically generate FLOWER diagrams from existing pipelines. Engineers and analysts leveraging these automated methods will be capable of describing and presenting complex code bases and data pipelines with minimal human input.

2 Our Proposed Hybrid Diagram

Before detailing FLOWER, we give fundamental concepts, definitions, and hypotheses related to the ER model, databases, and data sets.

Let E be a set of entities (objects), linked by relationships (references). We follow modern UML notation, where entities are represented by rectangles and relationships are shown by lines, with crowfeet on the "many" side. There exists an identifying set of attributes for each entity (i.e. a primary key or PK, an object identifier). A small (non-disruptive), but powerful change to ER notation is that relationships will have a direction, representing data flow.

Our diagram considers two primary kinds of entity: *Source* (stateful) entities containing raw data, such as SQL tables, files or other data sources, and *transformation* (flowing) entities representing intermediate steps in the data pipeline, such as scripts or partially merged tables. We assume that the input entities can be a bag, without a PK. However, when interpreting non-tabular information, it is common to vectorize or otherwise transform it into a tabular representation. Therefore, a line number or feature index for a text file may act as the primary key. In the case of images, the image file name is commonly a PK as well.

Our hypothesis: Not all files used for data analysis have PKs or attributes, as in traditional databases. However, we presuppose that data pre-processing programs do produce entity names, keys, attributes, and relationships that can be captured from source code (i.e. file name, feature variables, columns, and other names). We extend the term "primary key" or PK from the relational model to refer more generally to object identifiers, the set of attributes uniquely identifying particular instances of an entity in some data pipeline. Considering these and provided source code, inferred attributes (like the sentiment features of text data, or the formation components extracted from a decomposed image) for FLOWER can be reverse-engineered through automated analysis.

3 Capturing Data Flow with Generated ER Diagrams

In this section, we discuss the specific notation of FLOWER along with considerations to be made when modeling real-world data systems.

3.1 Extending ER diagram notation

We allow the relationship lines of ER to include an arrow indicating data flow direction. This direction can also be interpreted as input and output, going from input entities to output entities. The arrow is shown only for relating transformation entities to others. That is, in the case of "raw" source data sets that can be represented as normalized tables, there is no arrow between entities. The arrow represents (1) a processing dependence between two entities, and (2) data flow direction that indicates one entity is used as input.

This is a minor yet powerful change that enables navigating all data elements in the system under consideration (such as a Data Lake), providing a flow-aware view of big data processing. We emphasize that the entities remain linked by "keys", or identifying attributes for records. A FLOWER diagram retains traditional keys, and also considers keys derived from attributes used to link entities under transformation in order to unify relational and non-relational data entities.

3.2 Entities beyond Databases

An entity in FLOWER is broad in the sense that it can represent any object used in the context of BDA in various formats. An entity may be a file, matrix, relational table, dataframe, or more depending on what the analyst chooses to consider. Entities are broadly classified as:

– Source (raw, stateful) entities, representing raw data, loaded into the Data Lake or other system.
– Transformation (flowing, data pre-processing) entities being the output of some system (e.g., Hadoop), tool (statistics or machine learning), or some programming languages used in data science (e.g., Python, R, SQL).

We focus on representing data transformations for BDA, including machine learning, graphs, and text documents. Our diagram does not yet include the "analytic output" such as the parameters of the ML model, IR metrics like precision/recall, and graph metrics, which would be the subject of further work. FLOWER considers three major categories of data transformations:

1. *Merge*, which splices ("joins" in database terms) multiple entities by one or more attributes, which is a weakly typed relational join operator.
2. *GroupBy*, which partitions and aggregates records based on some key. We emphasize that Data Science languages (Python, R) provide operators or functions highly similar to the SQL GROUP BY clause.
3. *Derived expressions*, which represent derived attributes coming from a combination of functions and value-level operators (e.g. equations, string manipulation, arithmetic expression, nested function calls). These can be grouped together, or be separately categorized as same-type and different-type operations depending on the types of the inputs and outputs.

A FLOWER diagram encompassing these transformations may contain far more than simple table data sources. Many heterogeneous data types used in a pipeline may be considered as entity candidates for the purposes of FLOWER. Information on these entities can be retrieved either by the analyst or through the use of automated tools. We provide an (inexhaustive) list of potential sources:

- Importing ER diagrams available from existing transactional or analytical databases. We assume ER diagrams are available for a relational database DDL or exported from an ER diagram tool as CSV files.
- Automatic entity and attribute identification from metadata embedded in the file itself. We assume CSV files are the typical file format for spreadsheet data, logs and mathematical software. Other popular formats such as JSON can capture non-tabular but nonetheless interesting data.
- For text files like documents or source code, there may be statistics on strings for words, numbers, symbols, and so on, as well as text features computed from NLP techniques. In this case, we assume an IR library or tool preprocesses the file and converts the useful data into tables, matrices or data frames. We propose that the "keys" and attributes of these files be identified by their real-world usage, and so can be discovered by examining how they are used by the pipeline.
- Automatic data set and attribute name identification for data sets built by Python, R, or SQL code, generalizing a previous approach with SQL queries [6]. In our section on validation, we explore applications of this approach with a prototypical diagramming tool.

In the process of diagram generation, we may also define the transformation type and create new transformed entities either manually or as informed by automated observation. Data scientists may perform several transformations discussed above, generating temporary entities which we may also wish to capture. In the case of a "Merge," the entity structure may change, but the attribute values remain the same. A "Group by" may use one or more grouping attributes along with or without aggregations (sum, count, avg). In general, aggregations return numbers, but using only "Group by" will return the attribute values as their types. Mathematical transformations will mostly return derived attributes, though there are some analogs. We discuss in Section 3.3 a number of transformations that may be considered analogous to these operations.

Following these considerations, the new transformed entities are linked to and from the original entities using an arrow. Keys can be preserved or intuited based on the kind of operation performed. This FLOWER diagram may have applications leading to new insights for analysts in navigating source code, reusing functions, and avoiding the creation of redundant data sets.

We show in Figure 1 a FLOWER diagram example enriched with UML notation provided from a hypothetical store data processing pipeline. Arrows show the direction of flow and are compatible with other ER extensions such as those describing attributes and cardinality. We can see the flow of the transformed entities as from the source entities by following the direction of the arrow.

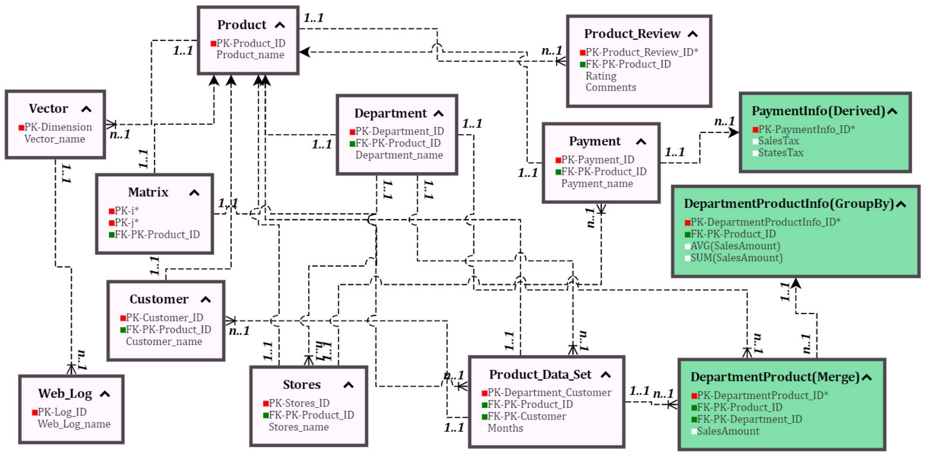

Fig. 1: Example FLOWER diagram: ER diagram enriched with UML notation.

Following a data-oriented approach, the model underlying a FLOWER diagram can be stored as JSON files or in metadata management systems for later updates and analysis.

3.3 Equivalent Operations

The exact relationship between software entities and the data pipeline will be application dependent; However, tools created to support FLOWER can be designed with the most common design practices in mind, and be supplied with extensions to support alternative workflows. The concept of *equivalent operations* greatly simplifies this task, as we can map the behavior of program code to transformations analogous to relational databases in certain contexts.

Because FLOWER is an extension to the concept of ER diagrams, we start from ER entities (SQL tables, generally) and extend the concept of “Entity” to describe any of the source entity types we choose to consider. In this way, we can map many kinds of data-bearing objects into source entities, together with SQL-equivalent operations (Table 1).

The usefulness of FLOWER is somewhat complicated by a well-known obstacle to static analysis: Many popular data analysis languages, in particular Python and R, are dynamically or “weakly” typed, meaning that many operations and attributes do not have known behavior or values until runtime. Static analysis of weakly typed languages is therefore limited to the attributes we can infer from program semantics. By leveraging the notion of equivalent operations, we argue that programs made to analyze data pipelines for FLOWER therefore can assume certain behaviors for a subset of program statements such as documented APIs (pandas, numpy), and adopt general policies for statements with unknown (or inaccessible, at the time of parsing) behavior.

Equivalent operations present a novel perspective in understanding data relationships by broadening of the scope of our understanding of a system as

Table 1: Examples of entity candidates and their SQL-equivalent operations

Entity Type	Constructed From	Transforms	Merges	Group Bys	PK
SQL Table	SQL	SQL operations	MERGE, JOIN	GROUP BY	PK
pandas DataFrame	from_csv, from_sql	vectorized operations	merge(), join()	groupby()	index or column
numpy array/matrix	np.array()	np.sin()	@, np.matmul()	projection to lower dimensional space	row or column index
image	open	color shifting	pasting	aggregate channel statistics	row or column index
text file	text feature extraction	feature biasing	data set combination	word, character, token summarization	label

we evaluate more objects as entities. Because entity operations can be roughly grouped together into equivalences, we can make more consistent inferences while developing our FLOWER diagram, and more easily account for these functions when describing patterns for automated tools to search for.

3.4 Data Flow Analysis

In this section, we introduce a method suitable for data flow analysis of source code. The primary languages for data analysis today tend to be imperative and object-oriented, therefore we focus on these paradigms in describing how the implementation of an analysis tool may be achieved.

Conceptually, this method transforms imperative statements into stateful operations that may be parsed into directed provenance graphs. These graphs can then be easily analyzed and transformed for presentation, which in our case leads to the creation of a FLOWER diagram. We define *Flow* and *State* objects, from which relationships may be observed:

Given: *States*, where a State is an object containing data representing the conceptual state of an entity in the ER model at a specific phase within the pipeline.

$$State := \{name, operation, parents, writes, reads, attributes\}$$

Where *name* is the text of the code statement producing it, *operation* is the name of the operation (such as a function call) that produces it, *parents* is the set of the parent states the State is derived from, $reads := \{R_1, R_2...\}$ is the set of resources (such as file name strings) read in its construction, $writes := \{W_1, W_2...\}$ is the set of resources that it may be written to, and $attributes := \{A_1, A_2, ...\}$ is a list of inferred attributes. A State can only be derived from a read operation or another State, so there will not exist any state that does not have an external resource in its ancestry.

Given: A *Flow*, containing a set of $statements := \{T_1, T_2...\}$ in a given imperative language, a set of $initial := \{S_1, S_2...\}$ States, and initially empty sets of reads and writes

$$Flow := \{initial, statements, reads := \{\}, writes := \{\}\}$$

Where a Flow can be used to describe anywhere a sequence of statements may exist with an optional set of initial states. In Python or R, scopes like modules and function calls are Flows, where a function's arguments form its initial states. For object methods, the owning object itself may be an initial state, as might be (separately) its attributes. The following operations are available on a State in a Flow:

– Read: A State may be constructed from one or more read operations in a single statement, which are put in its list of reads. This is the only type of State which might have no parent States. The operation for constructing this state will be a *read*.
– Update: States cannot be modified directly by operations within a Flow; Instead, each modification to a state results in a new State whose parents are the modified state along with any other states included in the modification operation. The operation for this state will be the operation used in the update. The new State replaces the old State within the controlling Flow.
– Assign: A State may be Assigned to another State label while retaining its own, without modification. Anywhere the original or assigned label is updated, the state referred to by both is updated and replaced according to the update operation above. This includes States belonging to other Flows: If a State in a Flow's initial state is assigned or updated, the State is updated and replaced there as well as in the current Flow. For languages with copy operations, copying is treated as an update, not an assignment.
– Write: A State may be written out to one or more resources. These resources are added to the *write* list in the State.

We also define *opaque* and *transparent* operations. An opaque operation is one that does not have behavior known to the program. A transparent operation is one that does have known behavior, such as behaviors referenced from available source code or otherwise making the operation known to the program ahead of time. Reads and writes are necessarily transparent, as they must be known to the program to be defined as read or write.

A Flow must first have its statements normalized into a list of *assignment expressions* of the form

$$Expr := \{\{S_1, S_2...\}, operation, \{E_1, E_2...\}\}$$

where S is one or more labels of a state to be updated (or created, if the state is not already present in the Flow), *operation* is the name of the operation used in this statement, and E is one or more expressions that may be parent States or non-State values. This process requires flattening each statement into an atomic assignment operation and recursively tracing the execution of nested flows to determine transformation relationships. We can extract information pertaining to data relationships between entities and flow states as follows:

1. Define and collect information on data sources and data transformations (imperative/OO language source code, in this case) under consideration.
2. Define a Flow for each data transformation with the relevant initial States.
3. For each Flow, pre-process its statements into basic assignment expressions.
4. Sequentially analyze each Flow's expressions to form a directed graph of State nodes. Nested Flows like transparent function calls are processed recursively and may modify the containing Flow's States. calls are ignored.
5. For each Flow, either: Treat the Flow as a transformation entity under FLOWER (considered as a single transformation) with its inputs and outputs based on all reads and writes; Or, collapse the State graph into a number of transformation nodes based on ancestry, preserving attributes and PKs. In both cases, reads and in the Flow with no linked write in the graph and vice versa are dropped as extraneous.

Because we only care about general data flow and cannot necessarily determine runtime behavior, control flow structures such as conditionals and loops are treated as inline statements. Assuming code correctness, all branches of the code are anticipated to be reached at some point, so we assume that State transformations in a branch should be captured regardless of whether conditions would cause it to be so. Recursive function calls are also treated as opaque operations. This approach sacrifices some accuracy but allows for linear time complexity O(n) and lets us avoid the halting problem.

3.5 FLOWER Diagram

We formalize the construction of the FLOWER diagram as follows:

Entities $:= \{E_1, E_2, ..., E_n\}$, a list of ER model entity objects $E_i := \{identifier, attributes, entity\_type\}$, where

identifier is a unique identifier such as a string specifying the entity name.

attributes $:= \{A_1, A_2, ..., A_m\}$ is a list of entity attributes. If (through observation or prior knowledge) it is determined that an attribute (or set of attributes) uniquely identifies a given instance of an entity, it is marked as a primary key (PK). More importantly, attributes that make references across different data structures are labeled as foreign keys (FK) and are similarly marked.

entity_type is a type specifier for whether this is a source or transformation/derived entity.

Relationships $:= \{R_1, R_2, ..., R_k\}$, a list of relationship description objects. The relationships among entities are represented by foreign keys. Cardinalities of entities in each relationship are defined: 1:1 relationships can be merged into one entity because they share the same primary key. Relations having cardinalities 1:N or N:1 (1 to many, many to 1) exist between distinct entities. M:N relationships (many to many) connect both entities, taking their respective foreign keys as its primary key. Relations are defined in the form $\{E_i, E_j, Rel_{type}, cardinality\}$, where

E_i **and** E_j are identifiers referencing entities in the ER model.

rel_type is a type specifier as to whether this is a normal or arrow relation. If this is an arrow relation, E_i must be the source and E_j must be the entity pointed to.

cardinality is the specifier for the cardinality of the relation from E_i to E_j: one-to-one (1:1), many-to-one (M:1), one-to-many (1:M), or many-to-many (M:N).

In order to derive the FLOWER diagram, we use the output from the source code parsing/analysis as input for a diagram program reading two complementary files describing entities and relationships. In our example, we output these files as JSON files, from which we generate the model. The following excerpt explains the structure of these files:

entities.json

```
[   // Source entities
    { key: "Product",
      items: [
        { name: "Product_ID",
          iskey: true,
          figure: "Decision",
          color: "red" }, ...],
      colorate: "#fff9ff" },

    // Transformation entities
    { key: "DepartmentProductInfo(GroupBy)",
      items: [
        { name: "DepartmentProductInfo_ID*",
          iskey :true, ... },
        { name: "FK-Product_ID",
          iskey: true, figure: "", ... }, ...],
      colorate: "#82E0AA" }, ...
]
```

relationships.json

```
[   // Source - Source (line)
    { from: "Sale", to: "Product",
      color: "black", dash: [3, 2], arr:"Standard",
      width:"1", text: "1..1", toText: "" },

    // Transformation - Any (arrow)
    { from: "Product_Data_Set",
      to: "DepartmentProduct(Merge)",
      color: "black", dash: [3, 2], arr:"LineFork",
      width:"1", text: "1..1", toText: "n..1" }, ...
]
```

For example, the Product entity given above may have its attributes inferred from corresponding source code like this Python snippet:

```
import pandas as pd
...
Product = pd.read_csv("products.csv")
Sale = pd.read_csv("sales.csv")

# Analysis observes pattern of use suggesting primary key:
ProductSales = Product.merge(Sales, how="left",
    left_on="Product_ID", right_on="Product_ID_fk")
...
```

Given two JSON files following this arrangement, a FLOWER model is derived portraying an ER representation of the observed pipeline. For each JSON object from the entities file, an entity $E_i \in Entities$ is created with its attributes described in the elements of the item. The primary keys are defined. Likewise, for each JSON object from the relations JSON file, a relationship such as $relation =< (E_i, Card_i, Flow_i), .., (E_j, Card_j, Flow_j) >$ is created. Foreign keys are defined from source to target relation. Cardinalities are associated with each entity-relation. The flow direction is also defined, if applicable.

3.6 Strengths and Limitations of our Approach

FLOWER's greatest strength lies in its compatibility with the ER diagram. By adding a single symbol, an arrow, FLOWER unlocks an entire paradigm of data inter-connectivity where previously only static relationships were considered. Importantly, the arrow shows dependence and data flow while preserving ER structure. The simplicity of the arrow does not correspond to its utility: by introducing the concept of flow to ER, the worlds of Data Warehousing and Data Pre-processing can be reconciled with minimal modification. Not only are we able to see where data goes and how it is transformed, we are also able to combine it with many other extensions to ER which is meant to be compatible. In particular, we are able to treat UML diagrams as a specialized form of ER diagram in this manner, as we show in our prototype. While the depiction of flow in a data pipeline is far from a novel concept, we believe the notion of extending ER for the purpose of looking backward at existing or proposed pipelines as opposed to a domain-specific or ad-hoc representation is. Other approaches also exist for the management of existing metadata, but do not yet specify the nature of actual data used ([7]: structured, unstructured, programs, etc.), or analyze the way the information is actually structured and extracted [8].

FLOWER excels particularly as the output of automated reverse engineering. Potentially complex or undocumented data pipelines can be observed and explained in part or whole in moments. In practice, days to weeks of analyst time might be saved when approaching problems such as documenting legacy systems. While ER is traditionally used in describing a relational schema before implementation, it can also be leveraged through use of FLOWER to help explain existing schemas using the same kind of diagram that would be used in their creation. There is an additional advantage in FLOWER being descriptive instead of prescriptive, as it can also enable the user to connect additional attributes to the ER diagram that traditionally do not (and in non-diagnostic cases, should not) be present, such as specific PF/FK relations to entities inside and outside the database.

Implementation of automated FLOWER diagramming will not be without its challenges. Real-world software supporting FLOWER must include some or all of the properties of modularity, nested analysis, some measure of attribute and type inference (as with linters and IDEs), and auxiliary source inspection. [9] gives methods of observing and describing data exchange in heterogenous schemas; We propose extending this knowledge in further work to include observed flowing data entities and describing these schemas with FLOWER. Our prototype only considers a subset of Python and known library transformations, which could be improved with the outlined capabilities to be robust enough for real-world use.

4 Validation

FLOWER is only useful if it is possible to procure data sufficient to generate an accurate diagram. As discussed above, weak typing and other obstacles to static analysis make finding the relationships between nodes in a real-world data pipeline challenging. In this section, we demonstrate the possibility of a sufficient analysis tool by presenting the results of an actual prototype implementation. This prototype performs a subset of the automated reverse engineering techniques described to inspect and explain real-world code.

4.1 Hardware and Software

The specifications of the computer used to implement and test our prototype are as follows: , Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz; Memory: 48.0 GB; Operating System: Linux (Ubuntu 18.04.4 LTS). Our prototype runs on pure CPython 3.10.10, with no other requirements. The primary modules from Python 3.10 used were *ast* for traversing Python's abstract syntax tree, and *JSON* for output. Other modules utilized were *typing, dataclasses, functools, uuid, glob* and *argparse*. Our diagram generator was developed using HTML, Bootstrap, CSS, JavaScript (JS), and the GoJS library. Git was used for versioning this project.[4]

4.2 Input Data

We validated the efficacy of our approach by building a prototype implementation for analyzing Python data pipelines. Our prototype focuses on the read and write patterns of pandas' read_* and to_* functions. All other functions are considered opaque, so no nested flow analysis is performed. The prototype also only inspects string literals when collecting resource information. We listed methods of succeeding these limitations in future work in Section 3.6.

[4] Full code is available on GitHub at `https://github.com/Big-Data-Systems/FLOWERPrototype`.

We provided the software with a path pointing to a collection of scripts used in prior research. These scripts used hard-coded resource path strings, which is what our prototype looks for in defining reads and writes. We consider the source code as fragments, absent their context or files they interact with, to simulate a complicated or disparate system of legacy code.

4.3 Parsed and Inferred Metadata

The default output of our prototype parser is a JSON file describing each Flow node together with its inputs and outputs, along with “interesting” internal State nodes forming the graph inside the given Flow. An interesting node is a State with a read operation, a write operation, or having more than one ancestors or descendant. It includes a list of all operations on non-interesting ancestors leading up to and including the current node. This approach summarizes States such that each line of non-interesting States followed by an interesting State form a single transformation entity within FLOWER.

For the diagram itself, a command flag causes the software to output entity and relationship JSON files suited to our FLOWER diagram generating program. This instructs the program to output the Flows and any files they connect to as Entities, and their connections as relationships. Optionally, it may also output more detailed files containing the interesting node summaries as entities.

4.4 Results: Diagram Output

We developed a basic graph visualization tool in JavaScript for the purpose of demonstrating automated FLOWER diagram generation. The tool reads the formatted data, in the form of JSON files for entities and relations, and generates a visualization of the complete system in FLOWER format. The end result of this process is a diagram very similar to one that might have been used on the reverse side to plan such a system.

In Figure. 1 we displayed a FLOWER diagram generated from source and entity files describing a hypothetical data set. This data set would be a wide range of files and processes dealing in statistics about stores, customers, products, and sales, which would be extracted from external and internal sources relating history sales data, customer information and buyers’ opinions. The green entities represent transformation objects such as data scripts, AWS Lambda, etc. that would be detected or declared by the analyst and added to the model. This fully-featured graph illustrates FLOWER’s capabilities.

The GUI presented to the user allows them to view the FLOWER model displayed as a dynamic graph. The user can navigate between and interact with the entities to examine their relationships. Further, the PK and FK columns are labeled and the cardinalities are displayed above the arrow link. The transformation entities are shown in color green and ER entities in white. The arrows represent the data flow. An analyst working in combination with these tools can fine tune the parameters of the intermediate files and inputs to create a more accurate diagram.

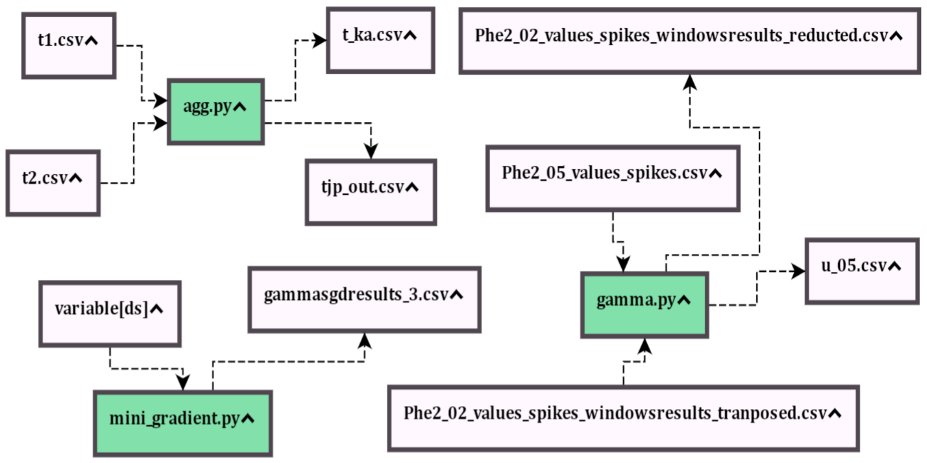
t1.csv
t2.csv
agg.py
t_ka.csv
tjp_out.csv
Phe2_02_values_spikes_windowsresults_reducted.csv
Phe2_05_values_spikes.csv
gamma.py
u_05.csv
Phe2_02_values_spikes_windowsresults_tranposed.csv
variable[ds]
gammasgdresults_3.csv
mini_gradient.py

Fig. 2: Results diagram for real BDA source code

We provided the results of our analysis prototype to our visualizer, producing Fig. 2. As expected, the pipelines do not connect between unrelated data sources, only their related fragments. The Python scripts are expressed as green boxes, showing them as intermediate steps in the process of data transformation.

4.5 Efficiency Considerations

Because we statically analyze source code (text data), the processing time is negligible. All our source code analyses and diagram generation ran in a fraction of a second. Data sets are not loaded into memory: the system only needs to scan the source code. The version of Python analyzed, 3.10, uses a PEG (parsing expression grammar) in parsing the tree. This grammar has an exponential worst-case parsing time complexity, which Python handles by loading the entire program into memory and allowing for arbitrary backtracking and caching. Our analysis leverages Python's existing parser (the *ast* module) to parse code into machine-readable syntax trees, then runs over each statement linearly to inspect behavior. The parsing step is no less efficient than the parsing done when the program is actually run, and our own inspection algorithm runs in O(n) time thereafter.

5 Related Work

In closely related research, [10] proposed an extension of ER diagram by adding "data transformation" entities in order to visualize tasks of data mining projects developed in the context of relational database warehouses. Two types of transformations were considered: denormalization and aggregation. Lanasri et al. [6] proposed a tool, ER4ML, that assists data scientists to visualize and understand the different data transformations in pre-processing data for a Machine Learning system. In a similar manner to [10], the authors focused only on transformations (denormalization and aggregations) that can be handled by SQL queries. This advanced work is state-of-the-art, but does not elaborate on understanding transformations performed by common data science languages like Python, and it does not capture flows beyond databases. In an alternative line of work, [5] solved data normalization with minimal human interaction. A layout algorithm for automatic drawing of the data flow diagram was presented in [3]. This layout algorithm receives an abstract graph specifying connectivity relations between the elements as input and produces a corresponding diagram as output. The authors of [1] presented a method for entity resolution that infers relationships between observed entities, and uses those relationships to aid in mapping identities to underlying entities.

In a similar context and in order to prevent data lakes from being invisible and inaccessible to users, there exist various solutions for data lake management [11], including data lake modeling, metadata management, and data lake governance. Other works focused only on generating a graph-based model to describe intra-object, inter-object, and global metadata [12] or assessing the Data Vault model's suitability for modeling a zoned data lake [13].

There is work on the detection of relationships among different data sets in data lakes [13,12]. However, a data lake contains diverse sources, therefore the description of the process of extracting information (including images, documents, programs; and so on) and identifying relationships between them is required. In addition, these works propose their own models for the representation of metadata. This stands in contrast to our approach, which is based on a common standard for data lake metadata representation, the ER model which is enriched with modern UML notation. Until now, no tools currently exist to explore data pre-processing in data lakes. Our vision for FLOWER seeks to unify data warehousing, database design, and data pre-processing.

6 Conclusions

Data science needs innovative techniques to capture data structure and interrelationships, following and extending proven database design techniques with respect to ER diagrams and relational databases. We believe this diagram proposal is a step in the right direction.

Evidently, there is a lot of work to be done. Our prototype examines top-level code, not processing nested flows. Additionally, it only inspects relationships between objects, not considering inherited attributes or keys. Extending to a more mathematically secure and functionally capable tool that can encompass Python as well as other languages, as well as incorporating previous work in SQL, will be the next big task in developing FLOWER. This includes executing portions of the code on input files and tracking values in order to understand data types, dependencies, and provenance. We also must study how to represent data filters, similar to the relational selection operator, in order to define a complete algebra and capture the whole data flow accurately. A complete platform would furthermore include parsing database query scripts [10] (e.g. SQL, SPARQL, MongoDB) and combining such outputs with existing UML diagram information to get a full data picture. Collecting opinions and observations from real data scientists will also be necessary to evaluate the usefulness, ease of use, and flexibility.

References

1. J. Mugan, R. Chari, L. Hitt, E. McDermid, M. Sowell, Y. Qu, and T. Coffman, "Entity resolution using inferred relationships and behavior," in *IEEE International Conference on Big Data.* IEEE Computer Society, 2014, pp. 555–560.
2. G. Guo, "An active workflow method for entity-oriented data collection," in *Advances in Conceptual Modeling - ER 2018 Workshops Emp-ER, MoBiD, MREBA, QMMQ, SCME, Xi'an, China, October 22-25, 2018, Proceedings.*
3. C. Batini, E. Nardelli, and R. Tamassia, "A layout algorithm for data flow diagrams," *IEEE Trans. Software Eng.*, vol. 12, no. 4, pp. 538–546, 1986.
4. M. Sebrechts, S. Borny, T. Vanhove, G. van Seghbroeck, T. Wauters, B. Volckaert, and F. D. Turck, "Model-driven deployment and management of workflows on analytics frameworks," in *IEEE International Conference on Big Data*, 2016, pp. 2819–2826.
5. M. Pham, C. A. Knoblock, and J. Pujara, "Learning data transformations with minimal user effort," in *IEEE International Conference on Big Data (BigData)*, 2019, pp. 657–664.
6. D. Lanasri, C. Ordonez, L. Bellatreche, and S. Khouri, "ER4ML: an ER modeling tool to represent data transformations in data science," in *Proceedings of the ER Forum and Poster & Demos Session 2019*, vol. 2469, pp. 123–127.
7. R. Eichler, C. Giebler, C. Gröger, H. Schwarz, and B. Mitschang, "Handle-a generic metadata model for data lakes," in *DaWaK.* Springer, 2020, pp. 73–88.
8. C. Quix, R. Hai, and I. Vatov, "Metadata extraction and management in data lakes with gemms," *Complex Systems Informatics and Modeling Quarterly*, no. 9, pp. 67–83, 2016.
9. R. Fagin, L. M. Haas, M. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis, *Clio: Schema Mapping Creation and Data Exchange.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 198–236. [Online]. Available: https://doi.org/10.1007/978-3-642-02463-4_12
10. C. Ordonez, S. Maabout, D. S. Matusevich, and W. Cabrera, "Extending ER models to capture database transformations to build data sets for data mining," *Data & Knowledge Engineering*, 2013.
11. C. Giebler, C. Gröger, E. Hoos, H. Schwarz, and B. Mitschang, "Leveraging the data lake: Current state and challenges," in *DaWaK.* Springer, 2019, pp. 179–188.
12. É. Scholly, P. N. Sawadogo, P. Liu, J. Espinosa-Oviedo, C. Favre, S. Loudcher, J. Darmont, and C. Noûs, "Coining goldmedal: A new contribution to data lake generic metadata modeling," in *DOLAP*, vol. 2840, 2021, pp. 31–40.
13. C. Giebler, C. Gröger, E. Hoos, H. Schwarz, and B. Mitschang, "Modeling data lakes with data vault: practical experiences, assessment, and lessons learned," in *ER 2019.* Springer, 2019, pp. 63–77.