# Data Virtual Machines: Simplifying Data Sharing, Exploration & Querying in Big Data Environments

Damianos Chatziantoniou
*AUEB, Greece*
damianos@aueb.gr

Verena Kantere
*NTUA, Greece*
verena@mail.ntua.gr

Nikos Antoniou
*AUEB, Greece*
nantoniou@aueb.gr

Angeliki Gantzia
*AUEB, Greece*
agantzia@aueb.gr

***Abstract*—Today's analytics environments are characterized by a high degree of heterogeneity in terms of data systems, formats and types of analysis. Many occasions call for rapid, ad hoc, on demand construction of a data model that represents (parts of) the data infrastructure of an organization, including ML tasks. This data model is given to data scientists to play with (express reports, build ML models, explore, etc.) We present a novel graph-based conceptual model, the Data Virtual Machine (DVM) representing data (persistent, transient, derived) of an organization. A DVM can be built quickly and agilely, offering schema flexibility. It is amenable to visual interfaces for schema and query management. Dataframing, a frequent preprocessing task, is usually carried out by experienced data engineers employing Python or R: a procedural approach with all the known drawbacks. Dataframes over DVMs are expressed *declaratively* - and visually, via a simple and intuitive tool. This way, non-IT experts can be involved in dataframing. In addition, query evaluation takes place within an algebraic framework with all the known benefits. I.e. a DVM enables the delegation of data engineering tasks to simpler users. Finally, a DVM offers a formalism that facilitates data sharing, data portability and a *single* view of *any* entity – because a DVM's node is an attribute and an entity at the same time. In this respect, DVMs can excellently serve as a data virtualization technique, an emerging trend in the industry. We argue that DVMs can have a significant practical impact in today's big data environments.**

## I. Introduction

Multiple data systems and analysis platforms need to co-exist, integrated and federated. While data warehousing is the usual approach, it is rigid for a rapidly changing data environment. In addition, [2] discusses the need for techniques that create *late-bound schemas* on data that may be persisted but are processed seldomly, if ever. This need is prevalent when the data infrastructure is dynamic and analysis purposes change agilely. For such environments, producing fully-fledged classical integrated schemas on structured, semi-structured and unstructured data is not only extremely expensive in terms of time and resources, but it may also be impossible to achieve in the time-window of the analysis. An alternative approach followed in many production environments when time to build a model or a report is of essence, is the use of a programming language (e.g. Python) to extract, transform and assemble data (e.g. a dataframe) within a program in an ad-hoc fashion. While agility is the clear benefit of this approach, data exploration for the end-user is non-existent: the creation of a new dataset requires a new program. An interesting thought is whether we can find a sweet spot between the two alternatives. Thus, we state the following high-level research question:

**Research question.** How can a DB expert *agilely and on-demand* create a virtual data layer on top of an organization's data infrastructure that can be easily understood and used by non-DB experts for *easy and intuitive* data sharing, data exploration, query formulation and data exporting?

To address this challenge, we propose the Data Virtual Machine (DVM). A DVM is a graph-based conceptual model based on entities and attributes – concepts that users understand well. The idea behind a DVM is simple yet powerful: given a computation $P$ with an output $(o_1, o_2)$, $o_1 \in A$ and $o_2 \in B$, where $A$ and $B$ are attribute domains, the output of $P$ serves as a mapping between $A$ and $B$. This can be represented in a graph by two nodes, $A$ and $B$, and edges between them representing the mappings as manifested by $P$'s output. $P$ can be an SQL/NoSQL query, a spreadsheet reader, any program.

By providing engineers with an environment where they can easily (visually) define computations over the data infrastructure (relational and non-relational data, streams, standalone programs) the DVM is automatically generated, i.e. the produced schema reflects a collection of computations (based on their output). As a result, a DVM can be *rapidly, agilely* constructed, *on demand*. DVM's approach is different than traditional data integration techniques which focus on settling on a schema and either define appropriate data processing tasks to populate/refresh this schema (data warehousing), or define wrappers to bind data with a virtual schema (mediators) [17], [21], [23] - in both cases, *existing* data is 'fitted' to a *predefined* schema. In a DVM, the mapping processes (i.e. computations) determine the schema.

We argued in [11] that a DVM serves practical, real-world requirements in big data environments, such as: (a) model simplicity and visual manipulations (schema/query), (b) schematic agility and flexibility, (c) consistent data sharing within organizations, (d) intuitive and simple query formulation, especially dataframing, and (e) model polymorphism.

Section II provides intuition for the DVM and formally defines it. Section III defines DVM *dataframe* queries and Section IV proposes an algebraic framework to evaluate them. Section V discusses query evaluation and optimization of DVM dataframe queries. Section VI presents a case study as a proof-of-concept and Section VII surveys related work. We conclude with Section VIII.

## II. Data Virtual Machines

A conceptual model, like the ER, is simple to understand, succinct, and depicts entities at a higher level. A DVM can be thought as an altered ER schema [11]–[13], where (a) all attributes are *derived* (the computation $P$ binds attribute value(s) to an entity via its primary attribute) and *multi-valued* (to accommodate mapping of multiple attribute values via $P$ to an entity), (b) entities are represented by their primary attributes, and (c) binary relationships are represented by associating the primary attributes of the two entities. In a DVM, each node can be considered either as an entity or an attribute, depending the analysis case. However, while ER uses a top-down approach, from a conceptual design to an actual data, DVM uses a bottom-up approach, from existing data to a conceptual model.

A DVM represents a collection of *mappings* (edges) between *attribute domains* (nodes), where mappings are manifested as data processes with a 2-dimensional output (the attribute domains) over existing data.

*Definition 2.1:* **[Key-list Structure]** A key-list structure (KL-structure) $K$ is a set of (key, list) pairs, $K = \{(k, L_k)\}$, where $L_k$ is a list of elements or the empty list and $\forall$ $(k_1, L_{k_1}), (k_2, L_{k_2}) \in K$, $k_1 \neq k_2$. Both keys and elements of the lists are strings. The set of keys of KL-structure $K$ is denoted as $keys(K)$; the list of key $k$ of KL-structure $K$ is denoted as $list(k, K)$. If $k \notin keys(K)$, the value of $list(k, K)$ is null. The schema of a KL-structure $K$, denoted as $K(A, B)$ consists of two labels, $A$ and $B$. $A$ is the role of the key and $B$ is the role of the list in the key-list pairs. ■

A key-list structure is a multi-map, where mapped values to a key are organized as a list.

*Example 2.1:* Figure 1 shows two KL-structures $K_1(custID,\ transID)$ and $K_2(transID,\ custID)$ in tabular format. ■

*Definition 2.2:* **[Data Virtual Machines]** Assume a collection $\mathcal{A}$ of $n$ domains $A_1, A_2, \ldots, A_n$, called *attributes*. Assume a collection $\mathcal{S}$ of $m$ multisets, $S_1, S_2, \ldots, S_m$, where each multiset $S$ has the form: $S = \{(u, v) : u \in A_i, v \in A_j,\ i, j \in \{1, 2, \ldots, n\}\}$, called *data processing tasks*. For each such $S \in \{S_1, S_2, \ldots, S_m\}$ we define two key-list structures, $K^S_{ij}$ and $K^S_{ji}$ as:

$K^S_{ij}$: for each $u$ in the *set* $\{u : (u, v) \in S\}$ we define the list $L_u = \{v : (u, v) \in S\}$ and $(u, L_u)$ is appended to $K^S_{ij}$. $K^S_{ji}$ is similarly defined.

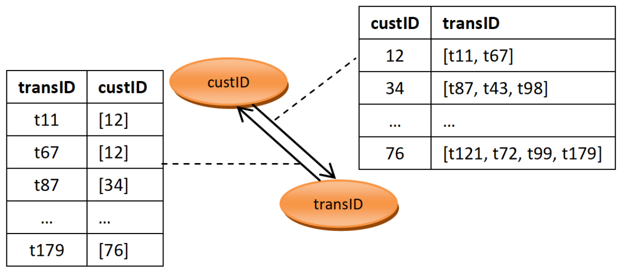
The DVM is a multi-graph $G = \{\mathcal{A}, \mathcal{S}\}$ constructed as:

- each attribute becomes a node in $G$
- for each data processing task $S$ we draw two edges $A_i \rightarrow A_j$ and $A_j \rightarrow A_i$, labeled with $K^S_{ij}$ and $K^S_{ji}$ respectively.

The key-list structure that corresponds to an edge $e : A_i \rightarrow A_j$ is denoted as $KL(e)$, with schema $(A_i, A_j)$. ■

The term data processing task actually refers to its output. The terms attributes and nodes of a DVM will be used interchangeably in the remaining of the paper.

*Example 2.2:* Assume two attributes, `custID` and `transID` and the output of the SQL query `"SELECT custID, transID FROM Customers` that maps transactions to customers and vice versa. The attributes, edges and the respective key-list structures are shown in Figure 1. ■



| transID | custID |
|---|---|
| t11 | [12] |
| t67 | [12] |
| t87 | [34] |
| ... | ... |
| t179 | [76] |

| custID | transID |
|---|---|
| 12 | [t11, t67] |
| 34 | [t87, t43, t98] |
| ... | ... |
| 76 | [t121, t72, t99, t179] |

Fig. 1. Attributes and edges in DVMs

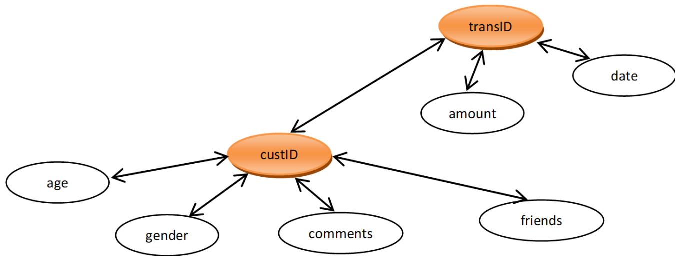


Fig. 2. A simple DVM example

In the rest of the paper we are going to use the DVM example of Figure 2. In this example, we assume the following: `custID` attribute is associated to `age` and `gender` attributes via SQL queries, to `comment` via a program that reads in a csv file and outputs (customer id, comment) lines, to `friends` via a cypher query over a graph database that outputs customer id and friend id rows and to `transID` via a spreadsheet reader that reads in an excel and outputs the transaction id and the customer id. The same excel is used to associate `transID` to `amount` and date. DVM's nodes connected to multiple (>1) other nodes are shown in different color and intuitively represent entities.

## III. Dataframe Queries

What kind of queries can we have on top of DVMs? There is an extended relevant research on query languages and visual query formulation over ERs, dating back in the 80s [4], [16]. However, defining a generic query language over DVM is work in progress. In this paper we focus on what data scientists/statisticians usually do, since this is the target group of this work. They usually form dataframes in Python, R or Spark. A dataframe is a table built incrementally, column-by-column.A column cell may contain an atomic value or a list. A dataframe usually provides a tabular representation of an entity and usually serves as input to ML algorithms. There has been discussion of a similar query class in the past (called multi-feature queries), both in terms of syntax (proposing SQL extensions [9], [14]) and evaluation (proposing a relational operator relying in parallel processing and in-memory evaluation techniques [10]). A dataframe query over a DVM is defined as a tree, consisting of DVM nodes and edges – *not necessarily* a subtree of the DVM. The root node of this tree (any node of the DVM) corresponds to the key column of the dataframe. Each additional column corresponds to a subtree of the root node, evaluated in a specific way.

*Example 3.1:* Consider the DVM of Figure 2. We form a dataframe with key (1st column) the `custID` and additional columns her `age`, `gender`, average sentiment of her `comments` containing the keyword "google", list of her `friends` and total `amount` of her transactions on May 2019.■

Selecting the `age` and `gender` should be straightforward as they are children of `custID`. All attributes are multi-valued, which means that several ages or genders may have been mapped to a `custID`. So we have to apply an aggregate function to the attribute before placing it in the dataframe. This function could be something simple like `any()` which picks a random element from a list of values.

In order to compute the average sentiment of a `custID`'s comments containing the keyword "google", we have to first filter the comments mapped to that `custID`, creating thus a new list of comments; then apply some python/R/Java function for sentiment computation on each comment of that list, creating another list of numbers; finally, aggregate this list using the `average()` function.

For the list of friends of a customer with $custID = x$, we can use the $list(x, KL(custID \rightarrow friends)$.

Finally, we want to compute the total amount of the customer's transactions on May of 2019. `amount` is not an attribute of `custID`, but there is a path to it, $custID \rightarrow transID \rightarrow amount$. So we can *make* it an attribute of `custID`, by replacing the transaction IDs of a customer in $KL(custID \rightarrow transID)$ with the amounts corresponding to each transaction ID in $KL(transID \rightarrow amount)$. At this point, a `custID` is associated to a list of amounts, which can be reduced using the `sum()` aggregate function. Before doing so, we may choose to replace a transaction ID with all the amounts that correspond to it, or first aggregate the list of amounts, for example taking the average of the amounts or choosing a random one using the `any()` aggregate function. One can see that expressing an aggregate of aggregates is simple. In addition, we have to constrain transactions to those of May 2019, i.e. the `date` attribute has to be between '2019-05-01' and '2019-05-31'. This is a *selection* condition on `transID` attribute, involving one or more of its children.

Figure 3 shows the tree of Example 3.1. Nodes used for the dataframe's output are marked. Definition 3.1 specifies a properly constructed dataframe.

*Definition 3.1:* **[Dataframe Queries]** Given a DVM $G = \{\mathcal{A}, \mathcal{S}\}$, a dataframe query is a tree structure $Q$, defined as:

- each node $N$ of $Q$ has a *name* and a *label*: the name is unique within $Q$ and the label is an attribute of $G$; these are denoted as $N.name$ and $N.label$ respectively
- for each edge $N \rightarrow N'$ in $Q$, there exists an edge: label($N$) $\rightarrow$ label($N'$) in $D$
- each edge $e$ of $Q$ is annotated with a list of transformations, called the *transformations string*, denoted as $e.transformations$

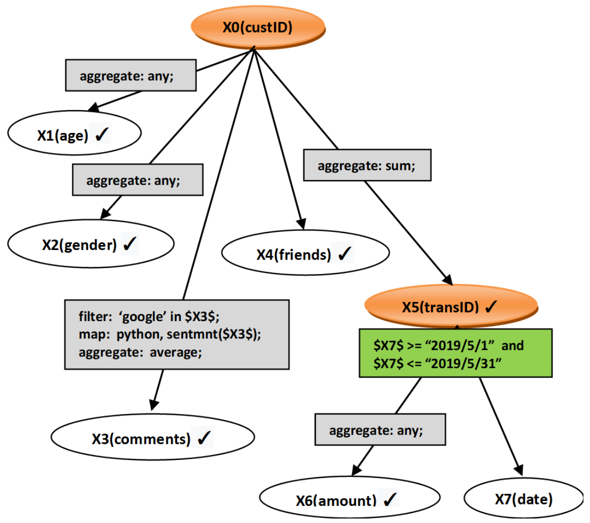


Fig. 3. An example of a dataframe query

- each node $N$ of $Q$ is annotated with the *selection condition*, which is either the special value `'TRUE'` or a python-like logical expression, where $N$ and any of $N$'s children may appear as identifiers within this expression;the selection condition of $N$ is denoted as $N.selection$
- each node, except the root, has an *output label*, which has the value `'true'` or `'false'`; if the output label of a node is `'true'`, then all nodes in the path from the root to that node, except the root, must have a `'true'` output label; this output label denoted as $N.output$ ■

Visual formulation of dataframe queries over DVMs can be an intuitive and simple task, as demonstrated in [12], making it a suitable query formulation technique for non-database experts, such as data scientists/statisticians. One can also express a dataframe using DVM-QL, a textual query language. We omit the detailed description of DVM-QL due to lack of space.

The root of a dataframe query can be any attribute of the DVM. Implicitly, it represents the entity to be studied. It can be `custID` or `transID` (customer entity or transaction entity) but it can also be `age` or `date` attribute. Being able to easily express queries for any entity in a virtual schema is one of the goals of data virtualization. Note that a DVM's node/edge could appear multiple times in a dataframe query, even in the same path (backtracking is allowed).

*Example 3.2:* Consider once again the DVM of Figure 2. For each gender we want to compute (in separate columns): the total number of customers, the average age of the gender's customers, the total amount of transactions corresponding to the gender, and, the average of the number of transactions per customer of the gender. Figure 4 shows the tree representation of this query. Nodes in the dataframe's output are marked.■

## IV. Algebra of Key-List Structures

To evaluate a dataframe query, such as the one in Figure 3, edges rooted on the same node have to be (conditionally) *combined* to a new edge (e.g. combining $X5 \rightarrow X6$ and

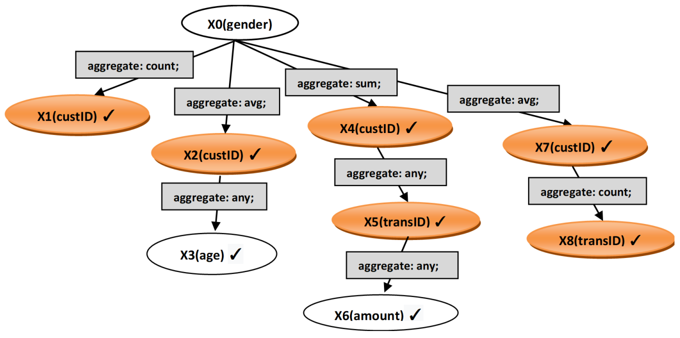
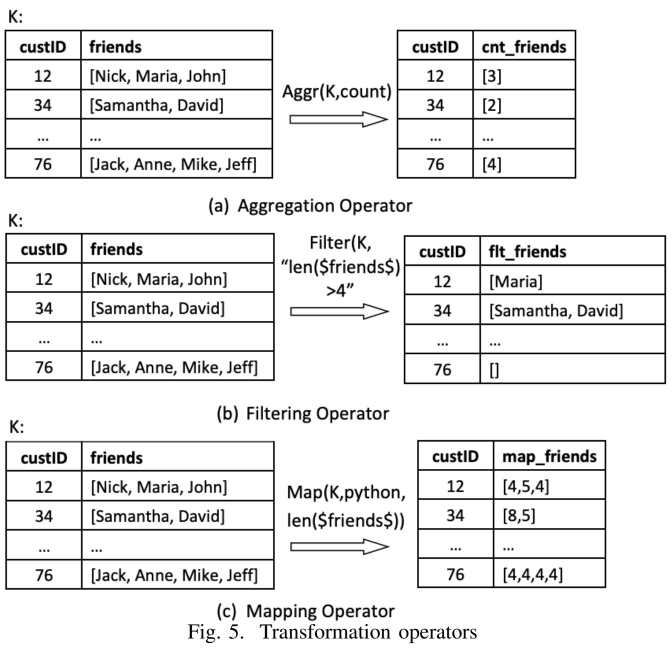


Fig. 4. Another example of a dataframe query



| custID | friends |
|---|---|
| 12 | [Nick, Maria, John] |
| 34 | [Samantha, David] |
| … | … |
| 76 | [Jack, Anne, Mike, Jeff] |

| custID | cnt_friends |
|---|---|
| 12 | [3] |
| 34 | [2] |
| … | … |
| 76 | [4] |

| custID | friends |
|---|---|
| 12 | [Nick, Maria, John] |
| 34 | [Samantha, David] |
| … | … |
| 76 | [Jack, Anne, Mike, Jeff] |

| custID | flt_friends |
|---|---|
| 12 | [Maria] |
| 34 | [Samantha, David] |
| … | … |
| 76 | [] |

| custID | friends |
|---|---|
| 12 | [Nick, Maria, John] |
| 34 | [Samantha, David] |
| … | … |
| 76 | [Jack, Anne, Mike, Jeff] |

| custID | map_friends |
|---|---|
| 12 | [4,5,4] |
| 34 | [8,5] |
| … | … |
| 76 | [4,4,4,4] |

Fig. 5. Transformation operators

$X5 \rightarrow X7$ to a new edge $e$), edges along a path have to be *joined* (e.g. $X0 \rightarrow X5$ and $e$), and edges have to be transformed (e.g. the lists of comments per custID, $X0 \rightarrow X3$, have to be filtered to contain the keyword “google”.) We define a set of algebraic operators that take as input one or more edges (i.e. key-list structures) and have as output an edge (another key-list structure.) As a result, dataframe queries are evaluated and optimized within an algebraic framework (Section V).

## *A. Transformation Operators*

These operators transform the lists of a key-list structure, producing a new key-list structure. For example one can filter the elements (strings) of a list according to a condition, aggregate the elements of a list according to an aggregate function or apply a function (written in any programming language as long as it gets a string and returns a string) on each element of the list, producing a new element. Others (e.g. sorting list elements) can be defined as well.

*Definition 4.1:* **[Aggregation]** We define an operator, called *aggregation*, which gets a key-list structure $K$ and an aggregate function $f$ and returns a new key-list structure $K'$ constructed as follows: $\forall k \in keys(K)$, it adds pair $(k, L'_k)$ to $K'$, where $L'_k = [f(L_k)]$, i.e. a list with a single element, the result of the reduced list $L_k$ according to $f$. We denote this operator as $Aggr(K, f)$. ■

The aggregate function $f$ can be one of the built-in functions (min, max, average, sum, count) or written in any programming language, such as Python or R. An implementation of this operator should handle polyglotism in terms of $f$, for example have implementations in different programming languages. Figure 5(a) shows an example of aggregation.

*Definition 4.2:* **[Filtering]** We define an operator, called *filtering*, which gets a key-list structure $K$ and a condition $\theta$ defined on a single element of a list (a string) and returns a new key-list structure $K'$ constructed as follows: $\forall k \in keys(K)$, it adds to $K'$ a pair $(k, L'_k)$, where $L'_k$ contains all $x \in L_k$ such that $\theta(x)$ is true. We denote this operator as $Filter(K, \theta)$. ■ This operator returns a new key-list structure with filtered lists for each key, according to $\theta$. Figure 5(b) shows an example of filtering. Note that this operator *cannot* be used to select $(key, list)$ pairs in a key-list structure (Section IV-C).

*Definition 4.3:* **[Mapping]** We define an operator, called *mapping*, which gets a key-list structure $K$ and a function $f$ with signature `string f(x:string)` and returns a new key-list structure $K'$ constructed as follows: $\forall\ k \in keys(K)$, it adds to $K'$ a pair $(k, L'_k)$, where $L'_k = [f(x) : x \in L_k]$, i.e. each element $x$ in $L_k$ is replaced by $f(x)$. We denote this operator as $Map(K, f)$. ■

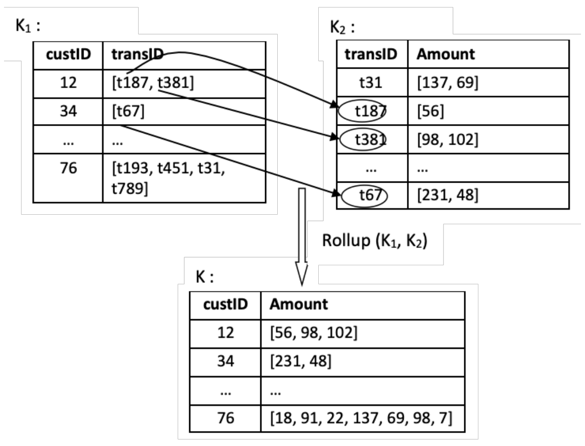
An example is to transform a list of comments to a list of sentiments, by running sentiment analysis on each comment. The mapping function $f$ can be written in any programming language. Figure 5(c) shows an example of mapping.

Aggregation, Mapping and Filtering are unary operators, i.e. they can be used on a single edge of a DVM. One can combine these operators to ask more complex queries involving a single edge of a DVM, such as *“what is the average sentiment of comments containing the keyword ‘google’?”* In this example, one would have a filtering operator to select the comments that contain the keyword *google*, followed by a mapping operator to transform a comment to a sentiment value, followed by an aggregation operator to compute the average of the sentiments.

## *B. rollupJoin Operator*

While one can compute the number of friends per `custID` using transformation operators, s/he cannot compute the total amount of transactions per `custID`. To do so, s/he has to first associate amounts to a `custID`, by replacing each `transID` mapped to a `custID` in the key-list structure $KL(custID \rightarrow transID)$ with the amount(s) mapped to that `transID` in the key-list structure $KL(transID \rightarrow amount)$. Figure 6 depicts the process. This way, a new `amount` attribute for `custID` is created (Figure 7). An aggregation operator (sum) can be applied on this new edge $custID \rightarrow amount'$. Queries like these involve a *path* from an attribute to another one. Thus, a join (“lookup and replace”) operator that gets two key-list structures (representing two consecutive edges in a path in a DVM) and produces a new one is required.

*Definition 4.4:* **[RollupJoin]** We define an operator, called *RollupJoin*, which gets two key-list structures $K_1$ and $K_2$



| custID | transID |
|---|---|
| 12 | [t187, t381] |
| 34 | [t67] |
| ... | ... |
| 76 | [t193, t451, t31, t789] |

| transID | Amount |
|---|---|
| t31 | [137, 69] |
| t187 | [56] |
| t381 | [98, 102] |
| ... | ... |
| t67 | [231, 48] |

| custID | Amount |
|---|---|
| 12 | [56, 98, 102] |
| 34 | [231, 48] |
| ... | ... |
| 76 | [18, 91, 22, 137, 69, 98, 7] |

Fig. 6. Lookup and replace values in a list

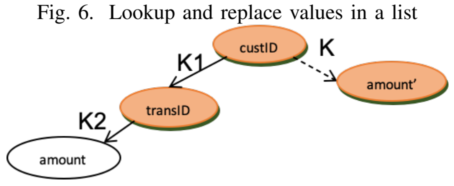


Fig. 7. Joining two consecutive edges to one

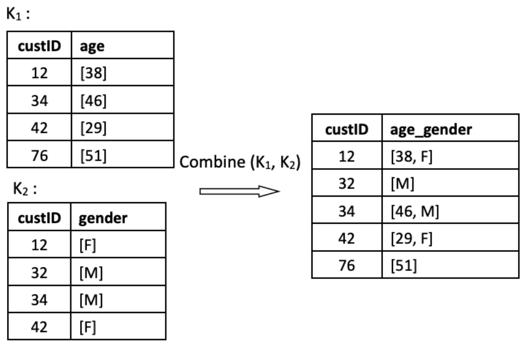
and returns a new key-list structure $K$ constructed as follows: $\forall k \in keys(K_1)$, it adds to $K$ a pair $(k, L_k)$, where $L_k = \oplus_{x \in list(k,K_1)} list(x, K_2)$, $\oplus$ stands for list concatenation. We denote this operator as $rollUpJoin(K_1, K_2)$.

### C. thetaCombine Operator

The last operator, thetaCombine, combines edges rooted on the same node to a new edge, as in Figure 9. Consider the query on the DVM of Figure 2: show for each custID the age and gender if age='25' and gender='F'.

In this example, the first issue has to do with *projection*, i.e. combining multiple "same-domain key" key-list structures to a new one, by concatenating same-key lists. Using edges $custID \to age$ and $custID \to gender$, we would like to create a new key-list structure, where the list of each custID is the combination (concatenation) of the corresponding age and gender lists for that custID. Figure 8 shows the idea. This operator is used to combine child nodes of an attribute, whenever needed (e.g. output of a query), and conveys the idea of projection in relational algebra.

The second issue has to do with *selection*. How can one select the custIDs where age='25' AND gender='F'? This selection condition involves several child nodes of custID (the key-list structures corresponding to these edges, to be precise.) Note that age and gender represent lists of values, so some assumption has to be made on what age='25' means (e.g. *all* or *some* value in the list.) Also note that the result of this query (selecting custIDs) is *not* a key-list structure, i.e. the corresponding operation is not algebraic, which is not desirable. Thus, this selection condition has to constraint a key-list structure where keys represent custIDs, such as $KL(custID \to transID)$. Of course, this selection condition may constrain more than one key-list structures, as long as their keys represent custIDs (in general, same key domains).



| custID | age |
|---|---|
| 12 | [38] |
| 34 | [46] |
| 42 | [29] |
| 76 | [51] |

| custID | gender |
|---|---|
| 12 | [F] |
| 32 | [M] |
| 34 | [M] |
| 42 | [F] |

| custID | age_gender |
|---|---|
| 12 | [38, F] |
| 32 | [M] |
| 34 | [46, M] |
| 42 | [29, F] |
| 76 | [51] |

Fig. 8. Combining age and gender

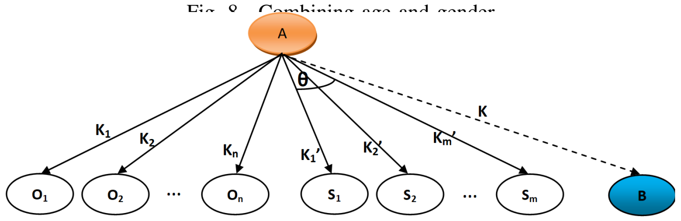


Fig. 9. Intuition for thetaCombine operator

For example, it may also constrain $KL(custID \to friends)$. The key-list structures that a selection condition constrain are called the *output* key-list structures for that condition.

We define a single operator, called *thetaCombine*, to express both ideas mentioned above. Figure 9 shows the big picture. $A$ is the root attribute, $B$ is a new attribute and the key-list structure $K = KL(A \to B)$ is the result of the *thetaCombine* operator. There is a list of "output" attributes $O_1, O_2, \ldots, O_n$, i.e. a list of output key-list structures $K_1, K_2, \ldots, K_n$ that need to be combined, as described above. There is also a list of "selection" attributes $S_1, S_2, \ldots, S_m$, corresponding to key-list structures $K'_1, K'_2, \ldots, K'_m$, which constrains the keys of the result, as described above. These lists of attributes may overlap. Either lists may be empty, but not both.

*Definition 4.5:* **[thetaCombine]** Given:

- a list $O$ of key-list structures $K_1, K_2, \ldots, K_n$, called the *output* list (can be empty)
- a list $S$ of key-list structures $K'_1, K'_2, \ldots, K'_m$, called the *selection* list (can be empty)
- a boolean expression $\theta(k, l_1, l_2, \ldots, l_m)$ involving $k$ (an atomic value), $l_1, l_2, \ldots, l_m$ (lists of values), called the *selection condition*

we define *thetaCombine*, denoted as $thetaCombine(O; S; \theta)$, which returns a key-list structure $K$ constructed as follows:

```
∀k ∈ keys(K_1)∩...∩keys(K_n)∩keys(K'_1)∩...∩ keys(K'_m):
    if (θ(k, list(k, K'_1), ..., list(k, K'_m))) is true:
        add a pair (k, L_k) to K, where:
            if O is empty then L_k = [ ] (the empty list)
            else L_k = ⊕_{i=1,2,...,n} list(k, K_i) ■
```

This definition takes the intersection of the keys of the involved key-list structures as the set of keys of the result (kind of inner join). One can define other versions of this operator, e.g. taking the union of the keys of key-list structures in $O$.

## V. Query Evaluation & Optimization

A DVM is a virtual schema and we assume no materialization of key-list structures. When a user defines a dataframe query $Q$, the data processing tasks of $Q$'s edges are executed and the corresponding key-list structures are populated. Materialized edges can be used, if exist.

## *A. Query Evaluation*

A dataframe query is evaluated bottom-up by combining outgoing edges of a node (except the root) using $thetaCombine$ and joining adjacent edges in a path using $rollupJoin$. An edge can be transformed by a series of transformation operators. Outgoing edges of the root are combined either by creating a dataframe row for each key, in which case the result is a traditional dataframe structure as defined in Python or R (i.e. a cell can be an atomic value or a list), or by using $thetaCombine$, in which case the result is a key-list structure. In the latter, the dataframe can be represented in the DVM as a mapping between the root and a new node, i.e the result is algebraic. Algorithm 1 evaluates a dataframe $Q$.

The function $keyCombine(A, B_1, B_2, \ldots, B_n)$ defines the key set of the new edge by combining the key sets of the involved edges, $A \to B_1$, $A \to B_2$,..., $A \to B_n$, according to the discussion of Section IV-C. The function *applyTransformations(A, B)* applies one more transformation operators on $A \to B$ according to the transformations string of that edge. The function $rollupJoin(A, B, C)$ applies the $rollupJoin$ operator on edges $A \to B$ and $B \to C$. The function *thetaCombine(B,*$C_1$*, ...,*$C_m$*)* implements the $thetaCombine$ operator for edges $B \to C_1$, ..., $B \to C_m$. The output list of the operator includes all edges $B{\to}C$ such that C.output is 'true' (may be empty). The selection condition is that of $B$.

Algorithm 1 produces a dataframe structure, i.e. it is `not` a key-list structure. This approach is chosen because most users prefer the output in this form. However, as mentioned, $thetaCombine$ can be used for the root node as well. This way, the answer can be added as a new attribute of the root.

**Algorithm 1:** Evaluating dataframe queries

```
Input: A dataframe query $Q$ with root node $R$
Output: A dataframe structure $DF$
Execution: evalQuery(R)
evalQuery (node $A$) {
   for each $A$'s child $B_i$ {
      evalChild ($A$, $B_i$);
      applyTransformations ($A$, $B_i$);
   }
   for each $k$ in combineKeys($A, B_1, B_2, \ldots, B_n$)
      if A.selection($A, B_1, B_2, \ldots, B_n$)
         write(DF, row(k,  list($k$, $KL(A \to B_1)$),...,
                           list($k$, $KL(A \to B_n)$))))
}
evalChild (node $A$, node $B$) {
   if $B$ does not have children, return;
   for each $B$'s child $C_i$ {
      evalChild($B$,$C_i$);
      applyTransformations($B$,$C_i$);
   }
   // a new edge $B{\to}C'$ is created
   $B{\to}C'$ := thetaCombine($B$,$C_1$,...,$C_m$);
   rollupJoin($A$,$B$,$C'$);
}
```

## *B. Query Optimization*

A dataframe query is transformed to a key-list algebraic expression, which can then be optimized. Due to lack of space, we only sketch these optimizations.

**Optimizations during edge materialization.** The key-list structure of an edge of the query has to be materialized during evaluation, i.e. the data processing task corresponding to the edge has to be executed. Optimizations include:

- Caching/pre-materializing "important/frequent" edges. For example, certain edges starting from `custID` or `transID`.
- Evaluating the edge's transformations on-the-fly during population of the key-list structure. For instance, consider edge $X0 \to x3$ in Figure 3 of Example 3.1. During the execution of the edge's data processing task, for each `(custID, comment)` pair read, `comment` can be checked whether it contains the keyword "google"; if yes, the python function `sentmnt()` is applied on it and the result is added to a total and a counter is incremented.

**Graph-related optimizations.** These are optimizations related to the analysis of the tree structure and in-parallel evaluation pf paths or identification of common subexpressions. As an example of the former, edges $X0 \to X1$ and $X0 \to X2$ in Figure 4 of Example 3.1 can be evaluated in parallel. For the latter, consider edges $X0 \to X1$, $X0 \to X2$, $X0 \to X4$ and $X0 \to X7$ in Figure 4 of Example 3.2. They all involve the edge $gender \to custID$, which can be materialized once.

**Query rewriting.** These optimizations modify the tree structure by adding/deleting/ combining edges and redefining data processing tasks. For instance, consider edge $X0 \to x1$ in Figure 4 of Example 3.2. This edge is defined by the data processing task "`SELECT custID, gender FROM Customers`". This edge can be replaced by another edge $X0 \to X1'$ defined by "`SELECT gender,count(custID) FROM Customers GROUP BY custID`". As another example, consider the path $X0 \to X5 \to X6$ in Figure 4 of Example 3.1 (assume there is no selection condition on $X5$ and no $X7$ node), defined by "`SELECT custID, transID FROM Transactions`" and "`SELECT transID, amount FROM Transactions`" respectively. This path can be replaced by another edge, $X0 \to X6'$ defined by "`SELECT custID, sum(amount) FROM Transactions GROUP BY custID`".

**Algebraic optimizations.** This set of optimizations include equivalent algebraic expressions and alternative/efficient operators' implementations, as in relational frameworks.

## VI. Case Study and Experiments

We use a publicly available dataset and a novel tool implementing and utilizing DVMs, called DataMingler [12], [13], to demonstrate the potential of DVMs in on-demand and agile modeling as well as simple and efficient querying. We develop a DVM on top of the underlying data infrastructure, which is then used to express a complex dataframe query. We compare this approach vs a Python-based implementation of the same query in terms of performance, simplicity in query formulation and maintenance. Supplementary material can be found in [1].

### *A. DataMingler: A Tool for DVMs*

DataMingler is a prototype GUI tool to (a) define and manage DVMs, (b) express dataframe queries in a visual and intuitive way, and (c) materialize the DVM (or parts of it) in other logical models – currently only JSON is supported.

*Data Canvas* is the module that enables the creation and manipulation of a DVM by mapping data and processes onto the graph and extending it with new nodes and edges. The data source types that DataMingler currently handles are: relational databases, csv files, excel and stand-alone programs (Java and Python). A DVM is kept in a Neo4j graph database.

Dataframe queries can be formulated either textually or visually, using the *Query Builder* module. In both cases, queries are represented in an XML-based intermediate representation and then parsed and transformed to a key-list algebraic expression, which is given to the optimizer and an execution plan is generated. Redis is used as the key-value engine for manipulating key-list structures.

The *JSON Exports* module can be used to instantiate model-specific databases (currently, JSON is supported). The user selects a node and a breadth-first-search tree rooted on this node is defined. Then the system generates a collection of JSON documents corresponding to this tree.

### *B. Data Infrastructure of the Case Study*

We assume a corporate environment that analyses business data. We use a public dataset offered by Yelp [37], which is 10.5GB uncompressed and comprises information on reviews, users, businesses, check-ins, photos and tips in JSON format. Each file is composed of a single object type, one JSON-object per-line. Beyond the Yelp dataset, we use synthetic data to create relational and non-relational data sources. We briefly describe the data sources used:

**Relational data.** We create a relational database by using views that contained each single-line object, and later splitting them into its components (columns/ attributes) using the OPENJSON function [38], and inserting into the following tables: 'business' (160,585 rows), which contains business data including location data, attributes and categories; 'user' (2,189,457 rows), including the friend mapping and all the metadata associated with the user; 'checkin'(18,641,929 rows), regarding the check-ins on a business; and 'tip' (1,162,119), with tips written by a user on a business.

**Non-relational data**. We maintain Yelp 'Review' as a non-relational data source in JSON format, which contains in total 8,635,403 reviews, 6.8 GBs. Furthermore, we create a spreadsheet (Excel) containing synthetic pricing data. We create randomised data in predefined business metrics relevant to Yelp, that amount to an annual revenue per business. We fill out information such as total ads and cost per click, to make a realistic measurement to revenue by ads per business. Then, we utilize Yelp's known revenue streams such as advertising partner program, yelp deals, gift certificates, yelp verified license and full service program. Those are summed to an annual revenue per business, which would presumably highlight the most important clients in Yelp's department.

**Data produced by programs.** We create a program that performs Sentiment Analysis. In order to predict the sentiment of a specific review, and subsequently aggregate it for each business, an NLP model is trained, using Count Vectorizer and Naive Bayes [15], [26]. The data is the review text and the rating, which is transformed from a numerical range ($[1, 5]$ with $0.5$ step) into binary $\{0, 1\}$ as follows: the stars in the range $[1, 3)$ are considered negative, while the stars in the range $[3, 5]$ are considered positive. A script loads the model and predicts the sentiment of each review. The model outputs (businessID, sentimentValue) for each review, where sentimentValue can have the values 'POSITIVE', 'NEGATIVE'.

### *C. Building the DVM*

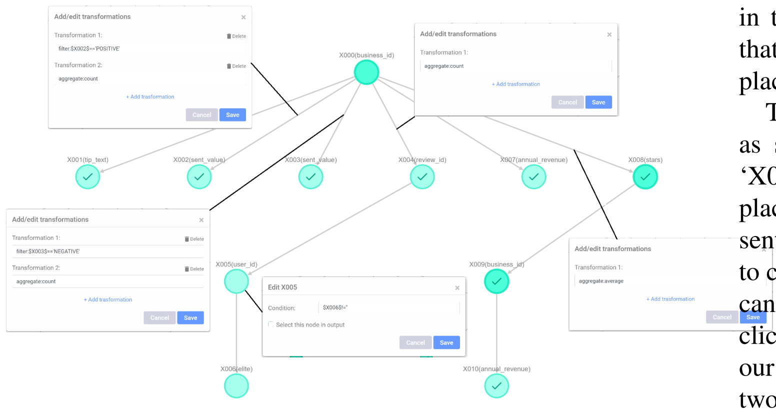
Data Canvas is the module of DataMingler that allows the creation and manipulation of a DVM. It lists on the left pane all available data sources (e.g. relational databases, csv files, spreadsheets, programs) and displays on the right pane the DVM being built. An edge between nodes of the DVM is defined via the use of a data source: the user specifies (a) the column/output position in the data source that corresponds to the head node of the edge (the root attribute) and, (b) the column/output position in the data source that correspond to the tail node of the edge (the child attribute). If these nodes do not already exist in the DVM, they are created. The required data processing task for this edge (an SQL query, a spreadsheet reader, etc.) is automatically derived and attached to the edge between the nodes. The tool supports multiple edge definitions between the same root node and multiple child nodes. Figure 11 shows the DVM built using the data processing tasks shown in the figure. Sources *S1, S2, S3,* and *S4* correspond to the relational database, the Excel spreadsheet, the program that outputs the prediction model, and a JSON reader of the "Review" data, respectively. The DVM was built within minutes and it is well-understood by the non-DB expert.

### *D. A Dataframe Example*

We consider a dataframe query with several types of information per businness_id which can be input to ML algorithms.

*Example 6.1:* For each `business_id` we want to have (a) a *list* of the tips related to the business, (b) the count of positive reviews (labeled as such by the prediction model), (c) the count of negative reviews (labeled as such by the prediction model), (d) the count of reviews made by "elite" users, (e) the `annual_review`, and (f) the average `annual_review` of same-star businesses. ■

Column (a) consists of a list of tips found in the 'Tip' table, described as succinct, witty comments on businesses, usually related to advice fellow Yelp users. This lists can be used for subsequent NLP analysis. Columns (b) and (c) contain aggregate review sentiment metrics for each business, based on



Add/edit transformations
Transformation 1: filter:$X002$=='POSITIVE'
Transformation 2: aggregate:count

Add/edit transformations
Transformation 1: aggregate:count

Add/edit transformations
Transformation 1: filter:$X003$=='NEGATIVE'
Transformation 2: aggregate:count

Edit X005
Condition: $X006$!=''
Select this node in output

Add/edit transformations
Transformation 1: aggregate:average

```
DPT1: SELECT business_id, stars FROM Business using S1
DPT2: SELECT business_id, tip_text FROM Tip using S1
DPT3: SpreadsheetReader(S2,1,12,`business_id',`annual_revenue')
DPT4: ExecEnv(S3,1,2,`business_id',`sent_value')
DPT5: SELECT user_id, elite FROM YelpUser using S1
DPT6: ExecEnv(S4,1,2,`review_id',`user_id')
DPT7: ExecEnv(S4,1,3,`review_id',`business_id')
DPT8: ExecEnv(S4,1,4,`review_id',`r_stars')
DPT9: ExecEnv(S4,1,5,`review_id',`review_text')
```

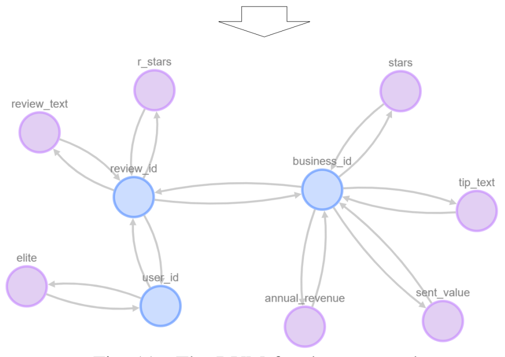


Fig. 11. The DVM for the case study

the pre-trained NLP review sentiment analysis, which is useful in the businesses' quest for marketing dominance. Column (d) concerns the count of reviews by elite users per business. This metric is useful due to the increased weight of an elite user's review in the microcosm of the Yelp website. Finally, columns (e) and (f) examine the annual business revenue per `business_id`, and the average business revenue of its star-category. This enables the analysis of the relationship between the stars and revenue and allows the comparison between the revenue of a business and its star-ranking segment.

We implement this query using DataMingler and Python.

*1) Dataframing using DataMingler:* Query Builder is the module of DataMingler for visual expression of dataframe queries. Recall that a dataframe query is a tree defined using DVM's nodes. The root is the first column (the key) of the dataframe and can be any node of the DVM. Additional columns of the dataframe correspond to nodes reachable by the root. The left pane of Query Builder shows the DVM as available nodes to add in the query, shown in the right pane.

Now consider the query of Example 6.1 (use Figure 10 as reference.) The first node to add is `business_id`, which becomes the root of the query (automatically named as 'X000' by the system). Clicking on 'X000', DataMingler shows all adjacent to `business_id` DVM's nodes as available to add in the dataframe query. Since we want to know the tips for that business, we select the `tip_text` node from the DVM, placed in the query as node 'X001'.

Then we want to know the count of positive reviews (labeled as such by the prediction model.) We once again click on 'X000' node and select the `sent_value` node, which is placed as node 'X002' in the query being built. The list of sentiment values represented by node 'X002' has to be filtered to contain only 'POSITIVE' values and then aggregated. Users can specify one or more transformations on a node, by double-clicking on the edge connecting the node and its parent (in our case clicking on the 'X000 → X002' edge and defining two transformations, ``filter:$X002$=='POSITIVE'`` and ``aggregate:count''.)

Continuing in the same fashion, we construct the dataframe query shown in Figure 10, annotated with all transformations and node selections. An interesting branch is the `X000 → X008 → X009 → X010`, which matches each `business_id` with its star-rating, backtracking then to match this star-rating with all `business_ids` and their `annual_revenue`. This navigational querying formulation style makes expression of a sequence of joins and transformations conceptually simple. We argue that the presence of a DVM and a visual query interface such as QueryBuilder simplifies the expression of complex and conceptually difficult queries and is thus appropriate for DB-illiterate people.

The algebraic expression for this query is shown below:

```
Top-Level-Dataframing (
   KL(X000→X001),
   Aggr(Filter(KL(X000→X002),$X002$=='POSITIVE'), count()),
   Aggr(Filter(KL(X000→X003),$X003$=='NEGATIVE'), count()),
   Aggr(rollupJoin(KL(X000→X004), thetaCombine
            (;rollupJoin(KL(X004→X005), thetaCombine
               (;KL(X005→X006);$X006$!='');true)),count()),
   KL(X000→X007),
   Aggr(rollupJoin(KL(X000→X008),rollupJoin
            (KL(X008→X009),KL(X009→X010))), average()) )
```

*2) Using Python:* In the absence of DVM, the programmer has to connect to the various data sources and treat each one separately. Note that each new Python program will have to replicate this step, which is non-trivial. Writing in Python the dataframe query of Example 6.1 requires the combination of the output of four queries into a single composite dataframe.
**Query 1.** The first query concerns the 'Tip' table. The implementation weighted heavily in SQL optimizations and, as such, it is mostly SQL code, executed through Python and the PYODBC library. The SQL query uses the FOR XML PATH clause to create a semicolon-separated list of tips.
**Query 2.** The data source is the output of the sentiment analysis Python script, with rows equaling the number of reviews and the columns being the `business_id` (business being reviewed) and the sentiment value ('POSITIVE' or 'NEGATIVE') of the specific review. The sentiment analysis script is executed and its output is decoded and split into a string that can be analysed by Pandas. A dataframe that contains every `business_id` and the calculated number of

positive and negative reviews is created.

**Query 3.** The third query counts reviews by elite users per business. The `user_ids` of elite users are fetched from the database and a dataframe that contains all `business_ids` and the respective count of elite reviews is created. The JSON file with reviews is accessed and read in chunks. For each chunk, we filter the reviews by their respective `user_id` and then select the `business_ids` of the filtered reviews. The count of elite reviews for each such `business_id` is computed, which is then added to the final dataframe.

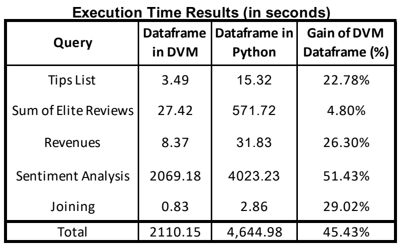
**Query 4.** The fourth query examines the annual business revenue per `business_id`, and the average business revenue of its star-category, with the star categories being 9, ranging from 1 to 5 in half-star increments. This query uses data from the Excel sheet and the database, combined with the existing dataframes in the program. The business_ids and respective revenues are loaded from the Excel file, followed by the business_ids and respective stars, from the database. The average revenue per star category is calculated and merged with the dataframe that contains the business_id, revenue, and star_category, with the latter as a key.

## E. Experimental Results

*1) Effectiveness of DVM in the implementation of dataframes:* Our experience with the described case study and beyond has shown us the benefits of using DVMs as a systematic way of defining agilely and on-demand data models and creating easily and intuitively DVM queries that implement dataframes. Specifically, our observations are:

**The curse of programming.** The Python dataframe code needed substantial design to be broken down in sub-queries. It was imperative to employ expert Python programmers to produce a highly optimized code. The programmers needed to study extensively the source schemas to define the workflow of the dataframe. This required substantial human effort, with the time and cost this comes with. Oppositely, employing DataMingler to create the DVM and the DVM query was extremely fast and easy: The DVM is produced automatically based on the available data processing tasks; for the specific study only a very small number of data processing tasks was necessary to be defined. The DVM query, designed visually on the DVM, while being a tree with only two levels and a few nodes, encapsulated all the complexity of the query workflow. It is apparent that a person that is not a data management expert, or a programmer, can easily define the DVM query.

**The bliss of re-usage and extensibility.** Let us assume that on the same sources we want to issue a new dataframe, similar to the one already implemented. For example, the new dataframe asks for (a) a couple of more columns in the result or, (b) the same information per user_id rather than per buisiness_id. Naturally, the first dataframe in Python is not documented and it is extremely hard to re-use or extend it for programming either (a) or (b), which would need to be designed from scratch. Yet, by employing DataMingler, adding a few more columns necessitates adding the respective data processing tasks and nodes to the DVM and adding these nodes with


Execution Time Results (in seconds)

| Query | Dataframe in DVM | Dataframe in Python | Gain of DVM Dataframe (%) |
|---|---|---|---|
| Tips List | 3.49 | 15.32 | 22.78% |
| Sum of Elite Reviews | 27.42 | 571.72 | 4.80% |
| Revenues | 8.37 | 31.83 | 26.30% |
| Sentiment Analysis | 2069.18 | 4023.23 | 51.43% |
| Joining | 0.83 | 2.86 | 29.02% |
| Total | 2110.15 | 4,644.98 | 45.43% |

Fig. 12. Execution times

the appropriate transformations in the DVM query. Moreover, for the (b) dataframe it suffices to express almost the same simple DVM query using a different node as the root.

*2) Performance results: DVM's algebraic implementation:* A Java program was written based on DataMingler's evaluation engine using the algebraic expression of Section VI-D1, with three hand written optimizations: (a) the external Redis key-value store was replaced with a native Java key-value store client, (b) evaluation of X000$\rightarrow$X002 and X000$\rightarrow$X003 edges were coalesced in one pass, with filtering and aggregation executed on the fly (while reading the program's output), and (c) the evaluation of X008$\rightarrow$X009 edge was "decorrelated", i.e. a list of `business_ids` is computed for each *distinct* value in $X008$ nodes (i.e. for each star) and not for all instances. Parallel evaluation was not employed.

*Python implementation:* the Python code contains multiple optimizations to minimize execution time as much as possible, by utilizing the optimizations of libraries (mainly using Pandas functions that reduce complexity by multiple factors.) The code contains as few 'for' loops as possible.

Figure 12 shows the execution time for the dataframe, in total and broken down for each sub-query. As mentioned, the dataframe is programmed in Python (which is the language used widely for the development of dataframes) and is also implemented via DataMingler in Java[1]. We observe that the execution time of the DVM query is considerably shorter that the execution time of Python dataframe, for each sub-query and in total, reaching an overall gain of 50%. This result is due to two factors: (a) DataMingler performs systematic optimizations for the execution of operators and, (b) DataMingler benefits from the fact that Java is a semi-compiled language whereas Python is an interpreted one.

## VII. Related Work

Classical data integration deals with the definition of a mediated, global, schema on top of heterogeneous relational data sources using a mapping between the global and the local schemata [17]. Answering queries using views [21] aim to retrieving data and optimizing the query. Data exchange [23] aims to transform data from a source to a target schema based on mappings with a focuson the existence of a solution. In both problems query containment and equivalence are of great importance and investigated with respect to the open and closed world assumption. Our attention is not on the original

[1]The experiments run on a PC with Processor AMD Ryzen 5 2500U - 2GHz- 8 logical Cores, Ram 8GB, Disk SSD LITEON CV8-8E128-HP.

queries, neither with respect to definition, nor evaluation. Finding alternative queries to extract data, determining the query equivalence, assessing data quality or performing optimization, is also out the scope of building a DVM.

Virtual Knowledge Graphs [27] focus on mapping definition [8] with a balanced trade-off of expressiveness and complexity. The W3C proposed OWL 2.0 [28] based on DL-Lite [6], [7]. Many techniques perform semantic reasoning and inference [25] using an ontology, or create the ontologies incrementally [3], [24], [36]. Tools that serve this purpose are Protégé [29] RacerPro [30], FaCT++ [18] and HermiT [22]. Oppositely to OWL 2.0 and semantic reasoners, DVM is not meant for expression of complex conceptual relations or reasoning. Thus, we do not focus in defining sophisticated mappings and balancing expressiveness with complexity.

RDFS [31] offers classes employing RDF. Works like [5], [20] create structured summaries or query RDF sources. Creating new querying techniques or expressive data summarizations is out of the scope of the DVM. Other works, like [35], [32], construct bottom-up an RDF graph from a relational schema. The work in [19] designs an ER model based on RDFS. Such works are orthogonal to ours, as they assume as starting point a given relational schema and goal the schema translation into RDF (via the employment of ontologies).

A system that supports query processing across heterogeneous data models by federating data stores [33] can be classified as a federated database, a polyglot, a multistore or a polystore [34]. Our work is motivated by similar concerns and could be considered a multistore.

## VIII. Conclusions

DVM is a novel graph-based model that depicts data and processes of an organization at a higher level. The DVM offers an intuitive way for conceptual representation of data and is amenable to visual manipulations. It is built agilely, on-demand and bottom-up and can be reoriented around any attribute that becomes the focus of study. Non-DB experts can express complex dataframe queries, which are evaluated within an algebraic framework. We use DataMingler, which implements the DVM, enables visual and textual definition of dataframe queries, and materializes re-orientations of the DVM in JSON, in a case-study showing the benefits of DVMs.

## References

[1] Supplementary material for the case study: Data schemas, Python query workflow, Python and Java code - Github repository. https://anonymous.4open.science/r/DVM_DataMingler-9F77/.

[2] D. Abadi and more. The beckman report on database research. In *Communications of the ACM*, volume 59, pages 692–99, February 2016.

[3] E. Alkhammash. Merging approach to support the incremental design of ontology. *Applied Mathematics Information Sciences*, pages 1–11, 01 2019.

[4] M. Angelaccio, T. Catarci, and G. Santucci. Qbd*: A Graphical Query Language with Recursion. *IEEE Trans. Software Eng.*, 16(10):1150–1163, 1990.

[5] M. Arenas and M. Ugarte. Designing a query language for rdf: Marrying open and closed worlds. 42(4), 2017.

[6] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The dl-lite family and relations. *J. Artif. Int. Res.*, 36(1):1–69, Sept. 2009.

[7] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Dl-lite: Tractable description logics for ontologies. volume 2, pages 602–607, 01 2005.

[8] D. Calvanese, A. Gal, D. Lanti, M. Montali, A. Mosca, and R. Shraga. Mapping patterns for virtual knowledge graphs, 12 2020.

[9] D. Chatziantoniou. The PanQ tool and EMF SQL for complex data management. In *Proceedings of ACM SIGKDD, 1999*, pages 420–424, 1999.

[10] D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. The MD-Join: An Operator for Complex OLAP. In *IEEE International Conference on Data Engineering*, pages 524–533, 2001.

[11] D. Chatziantoniou and V. Kantere. Data virtual machines: Enabling data virtualization. In E. K. Rezig, V. Gadepally, T. G. Mattson, M. Stonebraker, T. Kraska, F. Wang, G. Luo, J. Kong, and A. Dubovitskaya, editors, *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB Workshops, Poly 2021 and DMAH 2021, Virtual Event, August 20, 2021, Revised Selected Papers*, volume 12921 of *Lecture Notes in Computer Science*, pages 3–13. Springer, 2021.

[12] D. Chatziantoniou and V. Kantere. Datamingler: A novel approach to data virtualization. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2681–2685. ACM, 2021.

[13] D. Chatziantoniou and V. Kantere. Just-in-time modeling with datamingler. In R. Lukyanenko, B. M. Samuel, and A. Sturm, editors, *Proceedings of the ER Demos and Posters 2021 co-located with 40th International Conference on Conceptual Modeling (ER 2021), St. John's, NL, Canada, October 18-21, 2021*, volume 2958 of *CEUR Workshop Proceedings*, pages 43–48. CEUR-WS.org, 2021.

[14] D. Chatziantoniou and K. Ross. Querying Multiple Features of Groups in Relational Databases. In *22nd International Conference on Very Large Databases (VLDB)*, pages 295–306, 1996.

[15] Count Vectorizer. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html, 2021.

[16] B. D. Czejdo, R. Elmasri, M. Rusinkiewicz, and D. W. Embley. A Graphical Data Manipulation Language for an Extended Entity-Relationship Model. *Computer*, 23(3):26–36, 1990.

[17] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[18] Fact++. https://fact-project.org/FACT++.

[19] X. Gao, D. Ouyang, and Y. Ye. Designment of e-r model based on rdf(s). In *IWOST-2*, 2015.

[20] F. Goasdoué, P. Guzewicz, and I. Manolescu. Incremental structural summarization of rdf graphs. In *EDBT*, 2019.

[21] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, Dec. 2001.

[22] Hermit. http://www.hermit-reasoner.com.

[23] P. G. Kolaitis, J. Panttaja, and W.-C. Tan. The complexity of data exchange. In *PODS*. ACM, 2006.

[24] A. Menolli, H. Pinto, S. Reinehr, and A. Malucelli. An incremental and iterative process for ontology building. In *ONTOBRAS*, 2013.

[25] M. Mountantonakis and Y. Tzitzikas. Large-scale semantic integration of linked data: A survey. *ACM Comput. Surv.*, 52(5), Sept. 2019.

[26] Naive Bayes. https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html, 2021.

[27] D. I. Nsl. Virtual knowledge graphs: An overview of systems and use cases. 1:201–223, 11 2019.

[28] OWL 2.0. https://www.w3.org/TR/owl2-overview.

[29] Protégé. https://protege.stanford.edu.

[30] Racer. https://franz.com/agraph/racer.

[31] RDFS. https://www.w3.org/TR/rdf-schema.

[32] J. F. Sequeda, M. Arenas, and D. P. Miranker. On Directly Mapping Relational Databases to RDF and OWL. In *WWW*, pages 649–658, 2012.

[33] M. Stonebraker. The Case for Polystores. ACM SIGMOD Blog, July 13 2015.

[34] R. Tan, R. Chirkova, V. Gadepally, and T. G. Mattson. Enabling Query Processing Across Heterogeneous Data Models: A Survey. In *IEEE BigData*, pages 3211–3220, 2017.

[35] V. Vidal and et al. Incremental Maintenance of RDF Views of Relational Data. In *On the Move to Meaningful Internet Systems*, 2013.

[36] R. Volz, S. Staab, and B. Motik. Incrementally maintaining materializations of ontologies stored in logic databases. *J. Data Semantics*, 2:1–34, 01 2005.

[37] Yelp Dataset. https://www.yelp.com/dataset, 2021.

[38] Yelp Dataset SQL. https://github.com/arvindshmicrosoft/YelpDatasetSQL, 2021.