Example-Driven Exploratory Analytics over Knowledge Graphs

Matteo Lissandrini
matteo@cs.aau.dk
Aalborg University

Katja Hose
khose@cs.aau.dk
Aalborg University

Torben Bach Pedersen
tbp@cs.aau.dk
Aalborg University

ABSTRACT

Due to their expressive power, Knowledge Graphs (KGs) have received increasing interest not only as means to structure and integrate heterogeneous information but also as a native storage format for large amounts of knowledge and statistical data. Therefore, analytical queries over KG data, typically stored as RDF, have become increasingly important. Yet, formulating such queries represents a difficult task for users that are not familiar with the query language (typically SPARQL) and the structure of the dataset at hand. To overcome this limitation, we propose RE2XOLAP: the first comprehensive interactive approach that allows to reverse-engineer and refine RDF exploratory OLAP queries over KGs containing statistical data. Thus, RE2XOLAP enables to perform KG exploratory analytics without requiring the user to write any query at all. We achieve this goal by first reverse-engineering analytical SPARQL queries from a small set of user-provided examples and then, given the reverse-engineered query, we propose intuitive and explainable exploratory query refinements to iteratively help the user obtain the desired information. Our experiments on real-world large-scale KGs show that RE2XOLAP can efficiently reverse-engineer analytical SPARQL queries solely based on a small set of input examples. Additionally, we demonstrate the expressive power of our interactive refinement methods by showing that RE2XOLAP allows users to navigate hundreds of thousands of different exploration paths with just a few interactions.

1 INTRODUCTION

Knowledge Graphs (KGs) are nowadays already in widespread use because of their ability to represent entities and their relationships in many domains [38, 50]. Given the expressiveness of the model, an increasingly large body of statistical data is also stored and accessed through KGs within companies [47, 50] and as Linked Open Data on the Web [5], e.g., Open Government Data [11] or recent COVID-19 data[1]. In this context, *statistical data* within a KG form a heterogeneous network of numerical observations, entities, and relationships among them (Figure 1). Given the importance of KGs, typically stored in RDF [43], there is a need to support advanced analytics to extract relevant insights [1, 4, 6, 11, 22, 33, 34]. Analytical queries allow analyzing numerical information (observations such as the volume of immigration flows or daily contagions) along well-defined dimensions, e.g., the Time and Place dimension, to provide aggregate statistics, for example, results aggregated by month or region. Yet, expressing such analytical queries requires a significant level of expertise in the data model, the query language, and the content of the KG itself; expertise that is only very rarely unified within a single person. Consider the following example.

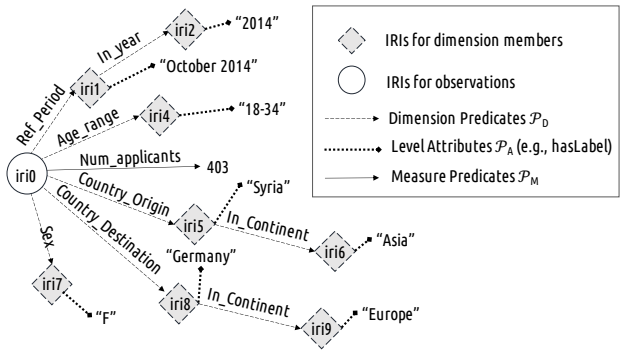

Figure 1: Portion of an RDF graph describing statistical data.

Running example. Alex is a journalist writing an article on immigration flows inspecting "Requests for Asylum" applications published as a statistical KG – a fragment is depicted in Figure 1. First, Alex tries to get an initial overview of the data, e.g., by examining how many people requested asylum to EU countries and from which continent. This requires formulating an initial query providing aggregate values for all combinations of continents of origin and countries of destination (Figure 2). Then, Alex would like to refine the query and investigate the data about Germany more closely and finally to identify countries with similar volumes of asylum requests. This requires adding appropriate filtering conditions to the initial query to obtain new refined queries. Finally, upon inspection of the initial results, Alex would like to look into particular details and aspects along dimensions, such as variations between continents of origin or age groups. This entails refining the query once more by expanding the set of attributes and adapting the aggregation levels.

As we see, a typical workflow requires formulating complex queries, inspecting their results, and providing appropriate refinements to obtain the required insights. These are cumbersome tasks even for expert users. For instance, in the first step, Alex expects the desired result to contain tuples relating to entities like Germany and Asia. Yet, while searching for such entities could be relatively easy, formulating an appropriate analytical query requires deep knowledge about the structure of the data, e.g., which sequence of triples connects the number of applicants to the continent of origin and which attributes can be used to add further filtering conditions. Here, we show how we allow users like Alex to simply provide the entities of interest, e.g., Germany, to bootstrap their analysis and directly obtain suggestions of candidate queries.

A common way to help users explore complex datasets is to develop algorithms and tools that synthesize queries based on a set of (partial) input examples [29, 30, 34], which can then be

[1] https://op.europa.eu/en/web/eudatathon/covid-19-linked-data

```
SELECT ?origin ?dest  SUM(?obsValue) WHERE {
 ?obs <Country_Origin> / <In_Continent> ?origin .
 ?obs <Country_Destination>  ?dest .
 ?obs <Num_Applicants>  ?obsValue .
} GROUP BY ?origin ?dest
```

Figure 2: SPARQL query to obtain aggregate per continent.

interactively refined in a sequence of consecutive steps. In such scenarios, domain experts commonly explore a dataset by searching for entities of interest, let the system identify appropriate queries to retrieve relevant information, and then interactively retrieve more information for those entities and similar ones [30–32, 36, 48]. Yet, entity exploration approaches do not support analytical queries. While the literature proposes approaches for analytical exploration in relational databases [41], none of them is applicable to KGs. The main reason is that KGs are characterized by heterogeneous structures composed of nodes and edges instead of tables with well-defined relational schemas; these characteristics already render reverse engineering of arbitrary (non-analytical) queries computationally hard [8]. The alternative of transforming KGs into the relational model and trying to synthesize analytical queries there is not viable either, because it incurs the extra cost and challenge of mapping and transforming data to the relational model and, more importantly, forfeits the advantages of the graph model, such as a flexible schema, easy discoverability and integration, the possibility to perform reasoning and to understand the context of the data.

Therefore, in this paper, we propose Re2xOLAP, which stands for Reverse-engineering and Refining Exploratory OLAP. This is the first approach to generate complex exploratory analytical queries and query refinements for statistical data within a KG. In summary, this paper makes the following contributions:

- The first formalization of the exemplar query problem for example-driven exploratory analytics over KGs.
- Re2xOLAP, an end-to-end approach comprising a novel algorithm, ReOLAP, to synthesize analytical SPARQL queries from user examples and ExRef a suite of example-driven query refinement methods for exploratory workflows.
- A fully functional Re2xOLAP system that operates on standard SPARQL interfaces (with non-specialized RDF stores) and implements optimizations for core operations.
- A comprehensive experimental evaluation on real-world datasets, which shows that Re2xOLAP efficiently reverse-engineers queries and guides the user through thousands of exploratory paths in a few seconds, even on large-scale statistical KGs.

This paper is structured as follows. First, Section 2 discusses related work. Then Section 3 formalizes the notion of statistical KGs and Section 4 formalizes example-driven analytical exploration in this context. Subsequently, Section 5 presents our ReOLAP method for synthesizing analytical SPARQL queries and Section 6 presents the query refinement methods introduced in Re2xOLAP. Finally, Section 7 presents the experimental evaluation and Section 8 concludes the paper with an outlook to future work.

2 RELATED WORK

In the following, we discuss existing approaches to support data exploration through analytical queries, i.e., On-Line Analytical Processing (OLAP) queries [25], and discuss their limitations in addressing exploratory analytics for statistical KGs.

Analytics for KGs. Supporting advanced analytics for KGs has now received attention for a number of years [1, 4, 6, 7, 55]. This is also motivated by increasing interest from public and private organizations to represent business data in specialized KGs [17, 38, 47, 50]. As a result, there have been proposals for standardized RDF vocabularies that allow describing statistical data [5]. The most complete is the QB4OLAP specification, a vocabulary for defining multi-dimensional (MD) data cubes on RDF [10, 11], which has also been extended for spatial data [19, 20]. Also, approaches exist to automatically enrich KGs with QB4OLAP annotations [56] and infer also dimension hierarchies [14]. These vocabularies allow specifying what is the subject of an observation (the fact), which measures are associated with it, its dimensions, and their hierarchies. The above corpus of work has hence focused on the definition of multi-dimensional data cubes that could be queried with standard SPARQL. Other approaches allow to specify analytical operators on KGs by means of "materialized views" and aggregations [4, 55] and allow OLAP query optimization on RDF data cubes [13]. Such views are themselves expressed in RDF, hence existing methods can be adopted to query these views as graphs. Finally, a different proposal allows for modelling the structure of a social network within a multi-dimensional space [2]. Yet, this model is not designed for KGs and cannot be used to produce OLAP queries over statistical KGs, which is the goal of Re2xOLAP. Therefore, despite the growing need for exploratory analytics on KGs [33, 34], in all the above cases, the user is required to be familiar with the contents of the data and with the (complex) query language. *In this work instead, we are able to lift these requirements by allowing automatic query synthesis and refinement from examples.*

Different works have focused on helping users understand the content of a graph by extracting summaries and interesting aggregates. To this end, such approaches distill recurring structures in order to produce *summary graphs* [28, 54, 58]. For instance, graph summarization supports data understanding by extracting recurrent structures describing how different entities are connected [18, 54]. Other works, instead, select different aggregation queries that describe specific entity types and are judged interesting because they present some peculiar characteristics (e.g., some skew) [6, 7]. These systems are able to extract high-level structural information from the data and to propose some analytical results profiling the numerical contents of an RDF graph. Nonetheless, *these systems do not take into consideration the user's information need* and do not directly support the task of reverse engineering and interactively refining complex analytical queries, as our approach does.

Exploratory Search for RDF Data. Several approaches allow exploring KGs by proposing graphical user interfaces for assisting the user in formulating *simple, non-analytical,* SPARQL queries [8, 9, 12]. These methods are designed to help users with a clear information need in writing (relatively simple) queries about specific entities. Recent advances have proposed systems able to accept keyword-query search over RDF data [9], as well as by-example reverse engineering of SPARQL queries from examples [8, 35]. In general, exemplar queries have proven effective for exploratory search on graphs [30, 31] and SPARQLByE [8] is the state-of-the-art method for reverse engineering SPARQL queries from examples. Nonetheless, we note that these approaches, and approaches for keyword search in general, can only help users

Table 1: Comparison of Existing Related Approaches: our method supports large KGs in RDF, provides queries with aggregations, and supports interactive query reformulation based on the (partial) user input.

	RE2xOLAP	SPARQLByE [8]	Spade [6]	REGAL [51]
RDF	✔	✔	✔	
Large KGs	✔	✔		
Aggregations	✔		✔	✔
Reformulations	✔			
User Input	✔	✔		✔
Partial Input	✔	✔		

find how specific nodes are connected. Thus, *none of the proposed systems is designed to support analytical queries*, nor to provide appropriate query refinements for analytical exploration, unlike our approach.

Exploratory Analytics for Relational Data. Exploratory analytics received much attention in the context of the relational model [23]. Typical methods to perform complex analytics over relational data are data warehouses and OLAP queries [24, 25, 40]. These usually require advanced expertise. Hence, different solutions have been studied to facilitate the user in the exploratory and analytical process [15, 26, 37, 45]. In particular, there exist approaches that help the user in discovering interesting insights by suggesting particular visualizations or filters on the data [57]. Different methods allow for *guided data exploration* by proposing query selectors that identify interesting subsets of the data based on clustering [48, 57]. These also provide specific navigation of the dataset with operations similar to drill-down queries [27], especially to identify outliers [46, 52] or subspaces (views) in which tuples are different from the rest of the database [49]. Other approaches search for outliers and unexpected trends (e.g., rapid rise during a time period) [52]. Additionally, many studies have also focused on exploiting examples to query enterprise data lakes (usually CSV files) [3] and reverse engineer SQL queries [39, 51, 53]. Among those, some approaches allow reverse-engineering SQL queries with aggregation and ranking [39, 51]. Yet, *these approaches require the user to explicitly provide also the result of the aggregation, which is a strongly limiting assumption.*

Furthermore, all aforementioned approaches are designed for the relational model only. *Hence, they are not directly applicable to the graph data model*, where there is no tabular schema. Moreover, as more and more data is being integrated into knowledge graphs [16, 38, 47], *it becomes infeasible to load the data into a relational database system* to explore it using existing relational tools. In practice, transferring KG data into a relational model for exploratory analysis defies the current needs of many companies and organizations to model their data as graphs [38, 47].

Research gap. In conclusion, as summarized in Table 1, approaches for KGs are currently missing for both the task of reverse engineering analytical queries as well as for suggesting appropriate query refinements. Thus, we focus on reverse engineering queries for KGs from partial exemplar answers *avoiding the need for transferring the data to a relational system. In this paper, we target this gap and introduce RE2xOLAP, the first comprehensive approach for supporting exploratory analytics for statistical KGs.*

3 STATISTICAL KNOWLEDGE GRAPHS

KGs are usually stored in RDF [43]. An RDF graph is a set of $\langle \mathbf{s}\ \mathbf{p}\ \mathbf{o} \rangle$ triples, meaning that the subject **s** has the property **p** with object **o** as value. An RDF graph may contain Internationalized Resource Identifiers $\mathcal{I}$ (IRIs), typed or un-typed literals $\mathcal{L}$ (constants), and blank nodes $\mathcal{B}$ (representing placeholders for IRIs or literals).

Definition 3.1 (RDF Graph). An RDF graph is a labeled directed graph $G = \langle \mathcal{N}, \mathcal{E}, \lambda \rangle$ with:

- $\mathcal{N} \subseteq \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ is the set of nodes, let $\mathcal{N}^0$ denote the nodes in $\mathcal{N}$ having no outgoing edges, and let $\mathcal{N}^{>0} = \mathcal{N} \setminus \mathcal{N}^0$;
- $\mathcal{E} \subseteq \mathcal{N}^{>0} \times \mathcal{N}$ is the set of directed edges;
- $\lambda : \mathcal{N} \cup \mathcal{E} \mapsto \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ is a labeling function[2] such that $\lambda_{|\mathcal{N}}$ is injective with $\lambda_{|\mathcal{N}^0} : \mathcal{N}^0 \mapsto \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$, $\lambda_{|\mathcal{N}^{>0}} : \mathcal{N}^{>0} \mapsto \mathcal{I} \cup \mathcal{B}$ and $\lambda_{|\mathcal{E}} : \mathcal{E} \mapsto \mathcal{I}$.

Finally, $\mathcal{P} : \{p \in \mathcal{I} \mid \exists e \in \mathcal{E}. \lambda_{|\mathcal{E}}(e) = p\}$ is the set of predicates for G.

Thus, an RDF graph represents any factual data in the form of linked statements (see Figure 1).

Statistical Knowledge Graphs. Let us discuss first how statistical data is represented within a KG. The multi-dimensional (MD) model is used to describe a set of observational data (called observations) associated with numerical information called measures (e.g., statistical information about the number of applicants) that can be aggregated across a number of common dimensions (e.g., time and location) each characterized by a set of attributes called levels (e.g., month and year for time, and country and continent for location). This model is also called a *Data Cube* [25]. In a KG, MD data can be described with the RDF Data Cube (QB) Vocabulary [5] (a W3C standard) and its extensions [11]. Additionally, multi-dimensional data can be extracted from a KG by specifying an analytical schema [4] over it, which is a set of view definitions over the graph to define observations, measures, and dimensions.

Therefore, a statistical KG is an RDF graph, also called an *RDF cube* [13], containing a set of *observation* nodes $\mathbf{O} \subseteq \mathcal{N}$ to which the statistical data is attached (e.g., iri0 in Figure 1). Following previous work [4–6], *our only assumption on the structure of the KG is that all relevant observations are instances of a predefined RDF class* (e.g., qb:Observation [5]) and we design the system to automatically infer the MD components and structure. We note that it is straightforward to obtain a statistical KG by creating a (materialized) view over an existing KG, as we discuss in the experimental section (Section 7). Our model of a statistical KG is composed of the following. First, a set of *measures* (**M**) with the respective predicates $\mathcal{P}_\mathbf{M} \subseteq \mathcal{P}$ (e.g., Num applicants) that define edges ($\subset \mathcal{E}$) between observations (**O**) and numerical values ($\subset \mathcal{L}$). These values are those to be aggregated in analytical (OLAP) queries (e.g., the value 403).

Then, a set of *dimensions* **D** across which the measure of each observation can be aggregated, which are identified by edges ($\mathcal{E}_\mathbf{D} \subset \mathcal{E}$) between observations (**O**) and other nodes ($\mathcal{N}_\mathbf{D} \subset \mathcal{N}$) called *dimension members* (e.g., iri5 representing Syria). These edges are labeled with appropriate dimension predicates $\mathcal{P}_\mathbf{D} \subseteq \mathcal{P}$ (e.g, Country of Origin). Each dimension member can also be linked to other members at a coarser or finer degree of specificity (e.g., from iri5 Syria to iri6 Asia). Thus, we also distinguish the case when two members reside at different *levels* in the *hierarchy* of a dimension (e.g., country vs. continent of origin). Moreover, level attributes $\mathcal{P}_\mathbf{A} \subseteq \mathcal{P}$ are predicates that assign descriptive properties to each member (e.g., a label or a name).

Given the formalization above, it follows that for each observation $obs \in \mathbf{O}$, there exists a set of dimension predicates $\{p_1, ..., p_n\} \subseteq \mathcal{P}_\mathbf{D}$ (n=|**D**|), such that we can link the observation

[2] Here $f_{|d}$ denotes the restriction of f to its sub-domain d.

node to the nodes of a corresponding member in each dimension with a suitable set of edges, i.e., $\exists d_i \in \mathcal{I}, e{:}\langle obs, d_i\rangle \in \mathcal{E}.\lambda_{|\mathcal{E}}(e) = p_i$ (as in Figure 1).

Consider the sample RDF graph of Figure 1. Sex, Age Range, Ref. Period, Country of Origin and Destination are attributes identifying each a different dimension (from **D**). Then Ref. Period points to a hierarchy with lowest level Month and highest level Year, similarly for the hierarchies for Origin and Destination where the lower level is Country and the highest is Continent. Entities like "Syria" and "October 2014" are dimension members for levels Country of Origin and Ref. Period Month respectively, while Num Applicants is the measure associated with each observation. Finally, a statistical KG modeled as an RDF cube is queried with SPARQL queries describing the connections between observations and dimension levels (as in Figure 2). In practice, a SPARQL query is composed of triples where some of the **s**, **p**, **o** positions are substituted with variables and are called basic graph patterns (BGP).

4 FORMALIZING EXAMPLE-DRIVEN EXPLORATORY ANALYTICS

Our first contribution is the formalization of example-driven analytical exploration of statistical KGs and the identification of its main steps and challenges (illustrated in Figure 3). For this reason, we investigate two problems: query synthesis from examples (Section 4.1) and interactive query refinement (Section 4.2).

In our system, the user provides as input one or more entities (e.g., Germany) and obtains the analytical queries producing output tuples involving those entities (e.g., aggregating volume of requests across countries of destination or countries of origin, as in the first interaction in Figure 3). The user will then select the desired query to be executed and obtains a first set of results (e.g., summing up the number of applications with Germany as the country of destination). This step of *query synthesis* is achieved by *reverse engineering* an OLAP query **Q**. Subsequently, the user asks for *further query refinements*. We propose three alternatives for this phase: (i) producing results at a finer level by introducing additional dimensions (Disaggregate operation), (ii) restricting the output of **Q** to a subset by introducing range filters on the measure values (Subset Results operation), and (iii) identifying tuples with similar measure values (Similarity Search operation). As shown later, the first operation corresponds to a drill-down operator while the second and third operations correspond to a dice operation in the OLAP model. Each operation can be applied multiple times and in any order *producing via simple interactions queries of arbitrary complexity*. Below, we provide the problem formalization for each one.

4.1 Query Synthesis

In *example-driven* paradigms, the user presents examples of the desired results instead of a query describing the conditions to be satisfied by such results. The most common approach to support this is to reverse engineer queries from example answers [30].

Definition 4.1 (REQ). (Reverse Engineering Queries from Examples) Given a set of exemplar answers $\mathcal{A}_E$ derived from a dataset $\mathcal{D}$, the task of reverse engineering queries requires to identify the family of queries $\mathcal{Q}$ such that $\forall \mathbf{Q} \in \mathcal{Q}$, $\mathcal{A}_E \sqsubseteq \mathcal{A} = \mathbf{Q}(\mathcal{D})$, where the symbol $\sqsubseteq$ represents an instance of a containment relation and $\mathcal{A}{=}\mathbf{Q}(\mathcal{D})$ is the set of answer obtained by execution of **Q** over $\mathcal{D}$.

Since the definition of the REQ task is general, the nature of the set of queries $\mathcal{Q}$ to be obtained depends on both the type of answers $\mathcal{A}$ and the adopted containment relation $\sqsubseteq$. Moreover, depending on $\sqsubseteq$, the set $\mathcal{A}_E$ can contain also partially incomplete answers [30]. *In this work, we aim at reverse engineering SPARQL OLAP queries over KGs.* That is, we accept as input entities and we support as output SPARQL queries of the form SELECT...WHERE...GROUP BY with graph patterns (BGPs) specifying observation nodes (facts) and measure values to be aggregated, and connecting them with members of a subset of the dimensions (inferred from the input entities) that also appear in the GROUP BY clause (as in Figure 2).

In accordance with the definition of an analytical schema [4], we define the output of an OLAP query **Q** on an RDF graph G in the form of a set of answer tuples $\mathbf{Q}(G) = \mathcal{T}$. Each tuple $t \in \mathcal{T}$ is of the form $t{:}\langle d_1, ..., d_k, m_1, ..., m_j\rangle$, such that each d_i, for $i \in [1, k]$, is a member of some dimension δ, i.e., $\forall\, i \in [1, k]\ \exists \delta \in \mathbf{D}$, s.t., $d_i \in \mathcal{I}_\delta$. Also, we have $m_1, ..., m_j \in \mathbb{R}$ to be real values representing the results of the aggregations over measures $M_1, ..., M_j \in \mathbf{M}$. Hence, with abuse of notation, we say that $\mathbf{D}(t){=}\{\delta_1, ..., \delta_k\} \subseteq \mathbf{D}$ and $\mathbf{M}(t){=}\{\mu_1, \ ..., \ \mu_j\}$ are the dimensions and the measures of t. Similarly, we define $\mathbf{D}(\mathcal{T})$ and $\mathbf{M}(\mathcal{T})$ also for a set of tuples $\mathcal{T}$ where all members have the same set of dimensions and measures. For instance, an output tuple from Figure 1 could be $t{:}\langle$iri6, iri2, 2500$\rangle$, i.e., involving the node corresponding to "Asia" and the year "2014" and assuming that 2500 is the result of some aggregation across the measure values linked to those two nodes.

We do not expect the user to know the result of a specific aggregation over an unfamiliar dataset, with the exception of special instances (e.g., when the measure is some special number like 0). Therefore, we reduce the amount of information required from the user and accept example tuples without any numerical value, i.e., where each example tuple t has $\mathbf{M}(t){=}\emptyset$, e.g., $t{:}\langle$iri6, iri2$\rangle$. This provides additional flexibility in exploratory settings. Hence, in the following, we focus on reverse engineering queries from exemplar answers where *no measure information is provided by the user*, and we leave the study of more restrictive cases for future work.

We also do not expect the user to specify an example tuple with IRIs (e.g., the IRI for Germany), but instead to refer to some literal value for their attributes (e.g., the label "Asia" or "2014"). Hence, example tuples have the form $t_E{:}\langle a_1, ..., a_k\rangle$ where each $a_i \in \mathcal{L}$ is a literal value connected by a predicate p_i to some dimension member d_i. That is $\exists \delta \in \mathbf{D}$, such that $d_i \in \mathcal{I}_\delta$ and $\exists p_i \in \mathcal{P}_\mathbf{A}$ with $e{:}\langle d_i, a_i\rangle \in \mathcal{E}$, $\lambda_{|\mathcal{E}}(e){=}p_i$. We also say that the tuple $t_E{:}\langle a_1, ..., a_k\rangle$ is mapped to the tuple $t{:}\langle d_1, ..., d_k\rangle$ by $p_1, ..., p_k$ and write $t_E \models t$. For instance, an example input tuple from Figure 1 could be $t{:}\langle$"Asia", "18-34", "2014"$\rangle$ and it can be mapped to the exemplar tuple $t{:}\langle$iri6,iri4,iri2$\rangle$ by p_1, p_2, and p_3 all set to the predicate hasLabel. Hence, each IRI is the corresponding dimension members for Origin, Age, and Ref. Period $\in \mathbf{D}$ respectively.

Given this input format, the system is hence required to map each input value a_i to a matching dimension member d_i. We say that a tuple t is subsumed by a tuple t', and we write $t \sqsubseteq t'$, *iff* $\mathbf{D}(t) \subseteq \mathbf{D}(t')$, $\mathbf{M}(t) \subseteq \mathbf{M}(t')$, and $\forall \delta_i \in \mathbf{D}(t).\delta_i(t){=}\delta_i(t')$. By extension we also say that $t_E \sqsubseteq t'$ *iff* $t_E \models t \wedge t \sqsubseteq t'$. For instance, we have that $t_E{:}\langle$"Asia", "2014"$\rangle \models t{:}\langle$iri6,iri2$\rangle$ and $t \sqsubseteq t'{:}\langle$iri6,iri4,iri2$\rangle$. Similarly, we say that a set of tuples $\mathcal{T}$ is subsumed by a set of tuples $\mathcal{T}'$ if $\forall t \in \mathcal{T}$, $\exists t' \in \mathcal{T}'$, such that $t \sqsubseteq t'$, and we write $\mathcal{T} \sqsubseteq \mathcal{T}'$.

Based on these considerations, we define the following specialization of the REQ problem (Definition 4.1):

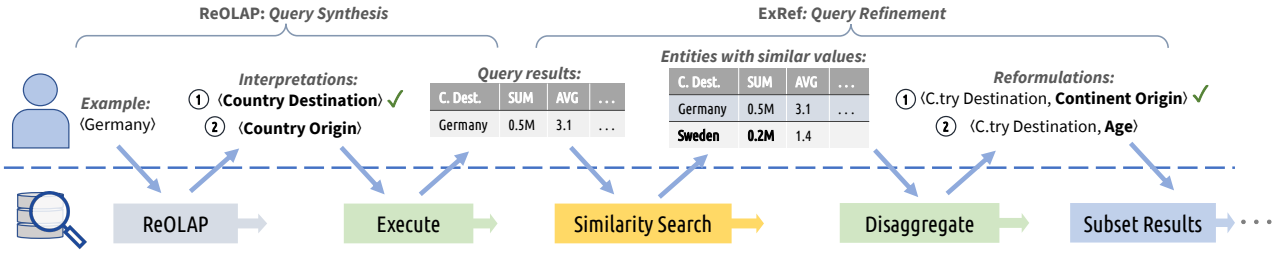

Figure 3: RE2xOLAP: example query synthesis and refinement steps, an interaction step is mapped to a pair of arrows.

PROBLEM 1 (REQ FOR SPARQL OLAP QUERIES). *Given a statistical knowledge graph G with a set of observations $\mathbf{O}$ and a set of exemplar tuples $\mathcal{T}_E$ as input. The problem of reverse engineering SPARQL analytical queries from examples requires to compute the set of valid SPARQL OLAP queries Q over $\mathbf{O}$ such that* $\forall \mathbf{Q} \in Q$, $\mathcal{T}_E \sqsubseteq \mathcal{T}_\mathbf{Q} = \mathbf{Q}(G) \wedge \forall t \in \mathcal{T}_\mathbf{Q}.\, \mathbf{M}(t) \neq \emptyset$.

Since the set Q is usually large, it is common to enforce some sort of minimality criteria on the obtained queries with the goal to not overload the user [41, 53]. In our case, we enforce that for each $\mathbf{Q} \in Q$, it should hold that $\mathbf{D}(\mathbf{Q}(G)) = \mathbf{D}(\mathcal{T}_E)$, so that the reverse engineered queries are limited to contain only dimensions that are matched by some part of the user example.

Thus, in our problem, given the input t:⟨"Asia", "Germany"⟩, the system would output the query in Figure 2, computing the number of asylum requests for every continent of origin and country of destination, as one of the valid output for the data in Figure 1. Then, we present the interpretations to the user so that they select the desired one. This corresponds to the *Query Synthesis* part of the workflow in Figure 3. Thanks to this minimality constraint and the fact that input examples are usually small given their exploratory nature, the number of resulting queries is not very large, usually from 5 to 10 in our experiments. Note that the user is free to expand to other dimensions in a later refinement step (Section 4.2 below). Hence, we focus on retrieving all such queries (ensuring *completeness*) and leave the problem of ranking interpretations to future work.

4.2 Query Refinement

After the user has picked a query $\mathbf{Q}$ to retrieve an initial set of results, the actual interactive exploratory phase starts. In this phase, the user may either require to restrict the current result set to just a few items that exhibit similar behavior, e.g., given the input t:⟨"Asia", "Germany"⟩, identify only those other countries of destination that have a similar amount of asylum requests, or to expand the current results to include new information, e.g., disaggregate the amount of requests across different years. The goal of the system is to generate a set of relevant refinements for the user to inspect and select among them those responding to their needs (as in Figure 3). A refinement is obtained by updating the query with filters to *limit the resultset to a subset* or introducing new dimensions in order to *aggregate* or *disaggregate* measures at different levels of granularity [25]. There are different ways in which data (and the corresponding aggregates) can be projected and filtered, e.g., returning only values where the country of destination is Germany or only countries having an aggregate sum between $0.5M$ and $1.0M$; in the OLAP terminology these filtering operations are called slice and dice respectively [25]. While disaggregating (or aggregating) values, i.e., moving between coarser and finer granularity levels in a hierarchy, are referred to as the drill-down (and roll-up) operations, e.g., drill-down the total sum for an entire year in the separate sums for each month [25].

In the following, we present example-driven methods to automatically perform these different operations and some more advanced ones for the *Query Refinement* phase of the workflow (Figure 3). At its core, example-driven query refinements produce new queries providing a different perspective on the data at hand while still describing the original user example (i.e., if Germany appeared in the results of the initial query, all subsequent refinements should still contain some tuples about Germany).

PROBLEM 2 (EXAMPLE-DRIVEN QUERY REFINEMENT). *Given the example tuples $\mathcal{T}_E$ and the query $\mathbf{Q}$ over the statistical knowledge graph G producing answer tuples $\mathcal{T} = \mathbf{Q}(G)$ s.t. $\mathcal{T}_E \sqsubseteq \mathcal{T}$, provide a set of example-driven refinement queries Q_r, such that each $\mathbf{Q}_r \in Q_r$ produces the resulting tuples $\mathcal{T}_r = \mathbf{Q}_r(G)$, for which it exists a subset $\mathcal{T}' \subset \mathcal{T}$ s.t. $\mathcal{T}_E \sqsubseteq \mathcal{T}' \sqsubseteq \mathcal{T}_r$.*

According to the definition above, a refinement could be a query for the same number of dimensions as the initial query $\mathbf{Q}$ but limited to a subset of tuples, or it can introduce a new dimension. For example, given the query producing the total number of applicants for all the countries of destination for a given year, a refinement query would limit the result to only those countries within the continent of the country appearing in the user example. Since the number of possible refinements is particularly large, we study a set of different refinement strategies that can produce filtering and disaggregation operations common of OLAP workflows while being particularly suited for example-driven exploration. In particular, we provide solutions for 3 refinement operations (depicted in Figure 3). These operations are formalized in Problems 2a-2c below.

PROBLEM 2A (EXAMPLE - DRIVEN DISAGGREGATE). *Given the example tuples $\mathcal{T}_E$ and an OLAP query $\mathbf{Q}$ over the statistical knowledge graph G producing tuples $\mathcal{T} = \mathbf{Q}(G)$ with $\mathcal{T}_E \sqsubseteq \mathcal{T}$, identify a set of refinement queries Q_r, such that each $\mathbf{Q}_r \in Q_r$ produces tuples $\mathcal{T}_r = \mathbf{Q}_r(G)$ with $|\mathbf{D}(\mathcal{T}_r)| = |\mathbf{D}(\mathcal{T})| + 1 \wedge \mathcal{T}_E \sqsubseteq \mathcal{T}_r$.*

PROBLEM 2B (EXAMPLE-DRIVEN SUBSET). *Given the example tuples $\mathcal{T}_E$ and an OLAP query $\mathbf{Q}$ over the statistical knowledge graph G producing tuples $\mathcal{T} = \mathbf{Q}(G)$ with $\mathcal{T}_E \sqsubseteq \mathcal{T}$, provide the set of subset-refining queries Q_r, such that each $\mathbf{Q}_r \in Q_r$ produces tuples $\mathcal{T}_r = \mathbf{Q}_r(G)$ with $\mathbf{D}(\mathcal{T}) = \mathbf{D}(\mathcal{T}_r) \wedge \mathcal{T}_E \sqsubseteq \mathcal{T}_r \wedge |\mathcal{T}_r| < |\mathcal{T}|$.*

PROBLEM 2C (EXAMPLE-DRIVEN SIMILARITY SEARCH). *Given the examples tuple $\mathcal{T}_E$ and an OLAP query $\mathbf{Q}$ over the statistical knowledge graph G producing tuples $\mathcal{T} = \mathbf{Q}(G)$ with $\mathcal{T}_E \sqsubseteq \mathcal{T}$, and given the number k, provide a set of similarity search queries Q_r, such that each $\mathbf{Q}_r \in Q_r$ produces tuples $\mathcal{T}_r = \mathbf{Q}_r(G)$ such that $|\mathbf{D}(\mathcal{T}_r)| = |\mathbf{D}(\mathcal{T})| \wedge \mathcal{T}_E \sqsubseteq \mathcal{T}_r$ and for each tuple $t \in \mathcal{T}_E$ and for some measure $m \in \mathbf{M}(t)$ it holds that the set $\mathcal{T}_r$ contains the k tuples that are most similar to t when compared to the value of $\mathbf{M}(t)$ for some similarity measure $\sigma : \mathcal{T} \times \mathcal{T} \mapsto \mathbb{R}$.*

Table 2: Resultset from the example ⟨"Germany", "2014"⟩, interpreting "Germany" as Country of Destination

Country of Destination	Year	SUM(# Applicants)
Germany	2014	8030
France	2014	5011
Italy	2014	1220
Austria	2014	120
…	…	…

Therefore, Problem 2a corresponds to a Drill-down operation, adding a new dimension to a query obtained from examples that were not mentioning it, while Problems 2b and 2c correspond to Dice operations that return a smaller subset of results based on the aggregate values of the measure.

5 REVERSE ENGINEERING QUERIES

Given the problem formulations presented above, we focus now on the *query synthesis* phase. Following the by-example paradigm, the user input is a tuple of dimension members the user is interested in, e.g., ⟨Germany, 2014⟩[3]. The expected output is a list of queries containing the required graph patterns linking the members to the observations (this requires also traversing the hierarchies) and including the grouping and aggregates appropriate to the obtained aggregation level. Thus, we present ReOLAP, a novel reverse engineering algorithm (Algorithm 1) to obtain SPARQL OLAP queries from examples (Problem 1). During query synthesis, the system needs to access the triplestore (e.g., through a SPARQL endpoint) to search for matches of the user example and how they are connected. Hence, to ensure efficient query answering, our algorithm adopts an in-memory representation of the high-level structure of the graph, called *Virtual Schema Graph*. In the following, we first present an example of the input and output of the algorithm, then we describe the steps of the algorithm (Section 5.1), then we illustrate how the virtual schema graph is built and employed (Section 5.2), and finally, we discuss correctness, completeness, and computational cost of our approach (Section 5.3).

Example. Assume a set of observations (e.g., Figure 1) and the input example ⟨"Germany", "2014"⟩. Algorithm 1 will produce queries that: (1) contain only 2 dimensions, resulting from the combination {Origin, Destination}×{Ref. Period} and (2) aggregate dimensions Origin and Destination at level Country and dimension Ref. Period at level Year. Hence, the system will generate exactly 2 queries: one aggregating across countries of destination and years, and another across countries of origin and years instead. For instance, the results of the former are of the form presented in Table 2.

5.1 The ReOLAP Algorithm

To generate the queries, in Algorithm 1, first each component of the tuple t_E:⟨a_1, ..., a_k⟩ is interpreted as an attribute of an example member of some unspecified dimension (e.g., "Germany" is the attribute label for the dimension member node). Consequently, each item in the input tuple is analyzed to identify to which dimensions it might refer to (Lines 2-3). Note that a single element could appear as a member in distinct dimensions (a country appears both as Country of Destination and Country of Origin). To obtain the match, we search in the database, i.e., we query the triplestore, for any entity matching the provided value (that is by keyword matching over labels). Then we retrieve to which predicates the retrieved entities are connected, and check whether any of those predicates is a dimension predicate $p \in \mathcal{P}_{\mathbf{D}}$. In this way, each component a_i in the example tuple is associated with a set of possible interpretations (Lines 4–5). Then, a query is generated (with the method getQuery) for each combination of the interpretations retrieved in the previous step (Lines 6–9).

Algorithm 1 ReOLAP: OLAP Query Reverse Engineering

Input: Target RDF KG G
Input: Exemplar Tuple t:⟨$a_1, \ldots, a_k$⟩
Output: Candidate Queries Q

```
    // Input is a single exemplar tuple with no measures
1:  Dims ←new Map()
2:  for each a_i ∈ D(t) do
    // Retrieve the dimension members that a_i may describe
3:      D_i ←matches(a_i)
        // Retrieve dimensions corresponding to each member
4:      for each d ∈ D_i do
        // The same entity can be member of multiple dimensions
5:          Dims[a_i] ←Dims[a_i] ∪ {⟨d, δ⟩ | δ∈D∧d∈δ}
6:      end for
7:  end for
8:  Q ← ∅
    // Combine all possible interpretations
9:  for each ⟨⟨d_1, δ_1⟩, ..., ⟨d_k, δ_k⟩⟩∈Dims[a_1] × … × Dims[a_k]
    do
    // Build the corresponding query
10:     Q ← Q ∪ {getQuery(P_M, ⟨⟨d_1, δ_1⟩ … ⟨d_k, δ_k⟩⟩)}
11: end for
12: return Q
```

The getQuery function produces a candidate SPARQL analytical query for a given combination of dimension members. When generating the queries, the dimensions that are not mentioned by the user will not appear in the query, following the minimality criteria of ReOLAP. For measures instead, we will retrieve results for all aggregation functions (*max, min, avg, sum*) over all available measures (in our example only Num Applicants). The resulting query has the form `SELECT...WHERE...GROUP BY`. The core of the query is in the corresponding `WHERE` statement. For each dimension member ($d_i \in \{d_1, \ldots .d_k\}$), the `WHERE` clause will contain a distinct set of Basic Graph Patterns (BGP) to produce the triple pattern of the path from the observation variable (`?obs`) to the desired member d_i in the selected dimension δ_i. The variables corresponding to the level members for each dimension ($vars_\delta$) also appear in the `GROUP BY` clause. These variables can also appear in filter conditions in the `WHERE` clause to restrict the query output.

Presenting Query Interpretations. Once we obtain the candidate queries, we aim to provide some natural language description of them to the user by exploiting existing domain-specific annotations in the database itself. In particular, in RDF, annotations on the schema of the data reside alongside the data itself. Therefore, we retrieve declarations of types and classes as well as labels on predicates and IRIs and use them to annotate the results. For instance, we can find in the graph that the predicate connecting the observation to the node for Germany is labelled as "Country of Destination", while another edge pointing to the

[3]For ease of presentation, the remainder of this section focuses on the case for a single example tuple. We also support the general case of multiple tuples and mixed queries targeting both dimension members and names.

IRI for Syria has a predicate with the label "Country of Origin". This allows us to translate the resulting queries into some descriptive text to facilitate their interpretation. Hence, we present the query to the user by using classic templating techniques like in Sparklis [12], for example, a query can be presented as: Return SUM(Num Applicants) grouped by "Country of Destination" and "Country Of Origin / Continent".

5.2 Virtual Schema Graph optimization

To efficiently support the query reverse engineering task (lines 2-3 in Algorithm 1 and method GETQUERY), we employ an in-memory representation of the organization of hierarchies for each dimension (see Figure 4). This allows us to reduce the need for querying the database. This structure, called the Virtual Schema Graph (also virtual graph for short), is a directed labelled graph that represents how the hierarchies of members are organized within each dimension and it is built automatically during the system bootstrap.

Definition. In practice, a dimension δ (e.g., Ref. Period) can be composed of one or more hierarchies $\mathcal{H}$ of dimension members, each hierarchy $H \in \mathcal{H}$ is composed by one or more levels $\ell \in \mathbf{L}_H$ (e.g., the Month *level*, and the Year *level*) and each level contains the dimension members (e.g., the Year *level* contains 2014 as dimension member, while the lower Month *level* contains, among others, October 2014 as dimension member).

In the virtual graph, we do not represent every single member (e.g., each year) but only the corresponding level (e.g., just the concept of Year). *This design allows the virtual graph to be orders of magnitudes smaller than the underlying graph.* Hence, as shown in Figure 4, the virtual graph has one node v_ℓ for each hierarchy level $\ell \in \mathbf{L}_H$ for each hierarchy $H \in \mathcal{H}_D$ for each dimension $D \in \mathbf{D}$, plus a default node v_o representing the base observation level, i.e., $|V| \approx |\overline{\mathbf{L}}|+1$ where $\overline{\mathbf{L}} = \bigcup_D^{\mathbf{D}} \bigcup_H^{\mathcal{H}_D} \mathbf{L}_H$. In this graph then, given levels ℓ_1 and ℓ_2, we insert an edge $e_{1,2} : v_{\ell_1} \mapsto v_{\ell_2}$ if $\exists H \in \mathcal{H}$ s.t. $\ell_1, \ell_2 \in H$ and the graph contains a predicate $p \in \mathcal{P}_{\mathbf{D}}$ that connects a member of ℓ_1 to a member of ℓ_2, and we assign to e the corresponding label p (e.g., the In Continent predicate). Then all nodes v_{ℓ_i} that have no incoming edges are the base levels and are connected directly to the observation node v_o with an edge $e_{o,i}: v_o \mapsto v_{\ell_i}$ with the predicate that connects instances of the observations to the corresponding base level members (e.g., the Country Origin predicate).

Construction and use. The virtual graph is built iteratively and automatically. The algorithm requires as input only access to the triplestore storing the graph G and the class identifying the set of observation nodes (e.g., qb:Observation, but different classes can also be provided or automatically constructed [7]). The system enumerates predicates linked to a set of observation nodes pointing to other non-literal nodes, these nodes are considered dimension members and the incident predicates are considered dimension predicates. Then, recursively, the process builds the hierarchies by enumerating predicates linking dimension members to further non-literal nodes, which are considered dimension members in higher hierarchy levels. Thus, this requires a bidirectional depth-first traversal (to handle cycles) of the graph starting from observation nodes. *The virtual graph is easily stored in memory since its size (number of edges) is bounded by* $|\overline{\mathbf{L}}|^2$. This allows reducing the number of queries to execute against the triplestore to extract information about the dimensions and their hierarchies. That is, when constructing a SPARQL query, the system produces the BGPs required in the body of the query by depth-first traversals of this graph and by keeping track of the labels of edges and nodes instead of querying the triplestore to identify connected predicates.

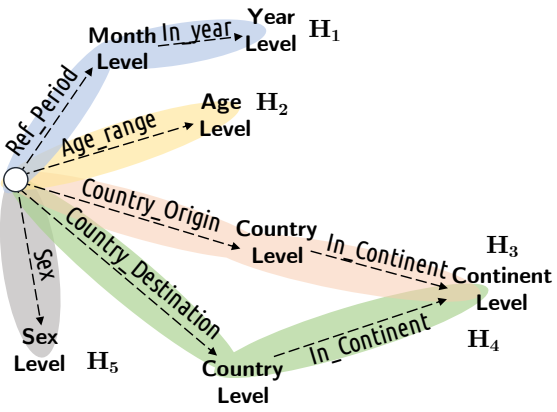

Figure 4: Virtual Graph for Requests of Asylum data

5.3 Analysis of the Approach

Correctness and Completeness. As mentioned earlier, the REOLAP procedure searches for all dimension members matching the examples in the query separately (e.g., first searches for all the interpretations of "Europe", then for all interpretations of "2014"), then builds all the possible combinations of interpretations, *this guarantees the completeness of the process.* Additionally, each interpretation is checked to return at least one observation by running it against the triplestore. *This guarantees the correctness of the results.* Therefore, our algorithm produces all possible queries matching the user input and guarantees that all of them are valid and return a non-empty result set when executed against the given graph.

Computational cost. Previous studies demonstrate that the complexity of reverse engineering SPARQL queries containing simple triple patterns (i.e., without aggregates and grouping) goes from P-time to co-NP [8], that is because for general queries there is no prior information on which predicates can be combined and in which sequence.

In our approach, despite the complexity of the triple patterns and the use of group-by operators we need to employ, *we are able to reduce the search space by exploiting the structure of the virtual graph and, therefore, we drastically reduce the overall computational cost.* In particular, in Algorithm 1, the matches for the examples are combined by navigating the virtual graph and not the complete graph. Hence, for each member of the user example, *running time grows proportionally to the number of dimensions, the number of hierarchies, and the cardinality of their members, but time complexity is independent of the actual number of observations.*

For instance, in the case of the asylum application data in our example, the search algorithm should look for a match among the 373 distinct dimension members (see Table 3 in Section 7) despite the dataset contains 15M observations. Then, while the number interpretations is $|\mathcal{N}_{\mathbf{D}}|^{|\mathbf{D}(t)|}$, the number of queries can be reverse engineered is $|\overline{\mathbf{L}}|^{|\mathbf{D}(t)|}$, i.e., it grows with the existing dimension levels (there are 7 in Figure 4, and actually 9 in the complete asylum application data, since some hierarchies have not been depicted for simplicity). This means that, theoretically, for a given database, the number of queries that can be obtained

Algorithm 2 RE2xOLAP: Reverse-engineering and Refinement

Input: Target RDF KG G
Input: Exemplar tuple $t{:}\langle a_1, \ldots, a_k \rangle$
// Retrieve candidate queries from exemplar tuple
1: $\mathcal{Q} \leftarrow$ReOLAP(G, t)
// Present the queries to the user, the user picks one
2: $\mathbf{Q} \leftarrow$Show$(\mathcal{Q})$
// Query refinement phase
3: ExRef$\leftarrow$ {Dis, TopK, Perc, Sim}
4: **while** true **do**
// Execute the query
5: $\mathcal{T} \leftarrow \mathbf{Q}(G)$
// Present the query results and the refinement methods
6: $\mathbf{R} \leftarrow$Show$(\mathcal{T}, \text{ExRef})$
// The user selects a refinement method $\mathbf{R} \in$ ExRef
7: **if** $\mathbf{R} = \perp$ **then**
// The user can stop at any time
8: break;
9: **end if**
// Apply $\mathbf{R}$ to obtain refinements for $\mathbf{Q}$ given $\mathcal{T}$
10: $\mathcal{Q} \leftarrow \mathbf{R}(\mathbf{Q}, \mathcal{T})$
// Present the refined queries, the user picks one
11: $\mathbf{Q} \leftarrow$Show$(\mathcal{Q})$
12: **end while**

grows exponentially in the number of input examples, e.g., for the input ⟨"Europe", "2014"⟩ this is 373^2 interpretations of dimension members and 10^2 possible queries. Similar proportions usually hold for other datasets as shown in Table 3. Nonetheless, *the actual search space is limited in practice*. First, the number of user-provided examples is usually small (<10). Moreover, in real-world scenarios, the number of matched dimension members is much smaller than $|\mathcal{N}_\mathrm{D}|$, since the attributes of the dimension members are usually well-differentiated, and then each user example usually matches only very few of them.

6 EXAMPLE-DRIVEN QUERY REFINEMENT

In the previous section, we discussed how the user can start from a tuple of entities of interest and obtain a list of interpretations (i.e., queries). After the user has selected a query $\mathbf{Q}$ from the output $\mathcal{Q}$ of the query synthesis phase above and obtained its results $\mathcal{T}$, the *query refinement* phase of RE2xOLAP produces *exploratory query refinements*. Figure 3 and Algorithm 2 show how these two phases are linked. In the interactive refinement phase (Lines 3-12 of Algorithm 2), the system presents to the user the current query $\mathbf{Q}$ and its results $\mathcal{T}$, and the user can select one refinement strategy among those we offer. Thus, to support the *query refinement* phase, we devise ExRef, a suite of methods to address Problem 2a, 2b, and 2c. Thus, we enable three independent refinement operations: *disaggregate*, *subset*, and *find similar* (see example in Figure 3). The user can then decide which operation to execute, then obtains the new (refined) result, and continues the exploration with a new refinement (or backtracks to a previous query to start a different exploration path). Therefore, a user can move from very simple queries to more complex ones without the need to write any query.

Solution criteria. Following the example-driven exploration paradigm [30], we produce refinement queries whose result sets still contain at least some tuple matching the original user example. *Moreover our preliminary study (Section 7.2) lead us to identify two complementary main criteria to guide the design of our solutions: 1) simplicity and 2) explainability*. When exploring unfamiliar data, the system is going to present results for which the user cannot immediately assess the correctness and for which they may have no prior expectation. Therefore, query refinements proposed by the system should introduce the minimal amount of variation possible in the conditions applied (simplicity) and the reasoning behind each new condition that is added (or removed) should be easily explainable to the user (explainability). In the following, we show concrete implementations for each operation based on these criteria.

6.1 Example-Driven Disagreggate

Problem 2a requires to provide refinements that disaggregate the current results over an additional dimension (or at a more detailed level within the current dimension). For this refinement method (Dis), we exploit once more the virtual graph (Section 5) and enumerate all available dimensions and levels not yet included in the query to generate a list of alternative queries. In practice, to produce valid triple patterns connecting observations to level members, we enumerate all paths starting from the root node (v_o), each path identifies a sequence of predicates to reach a specific level in a specific hierarchy for a given dimension (e.g., the sequence Country Origin followed by In Continent to reach the continent level). By checking each path against those already described by the query, the system can discard invalid refinements (i.e., those that are already included or that would aggregate at a higher level instead of disaggregating). Thanks to the virtual graph, this operation does not need to query the triple store and is performed very efficiently, with complexity linear in the number of levels over all dimensions, i.e., $O(|\overline{\mathbf{L}}|)$.

6.2 Example-Driven Subset

During the exploration, it is common that for a given query $\mathbf{Q}$ the cardinality of its results $\mathcal{T}$ is too large for the user to be inspected (e.g., in Figure 3, after disaggregating both for Country of Origin and Year the query returns hundreds of tuples). Therefore, a solution for Problem 2b requires to obtain a query that can produce a smaller set of tuples that the user can focus their attention on. In both the literature [49] and our interviews with users (Section 7), we have found that a common need is to identify tuples for which the resulting measure values are the highest (or lowest), or alternatively form a cluster of similar values. In any case, the identified subsets are guided by the initial user example, i.e., $\forall t' \in \mathcal{T}'. \exists t \in \mathcal{T}_E$ s.t. $t' \sim t_e$, for some similarity relation $\sim$. Hence, we propose two options: *top-k refinement* (TopK) and *percentile-based refinement* (Perc).

Top-K refinement. This refinement method returns a set of top-k tuples $\mathcal{T}_k$ for a measure m_i such that at least one tuple matching the user input appears among them, i.e., $\exists t \in \mathcal{T}_E. t \in \mathcal{T}_k$. The refinement algorithm works as follows. For every measure m_i, it orders the tuples in $\mathcal{T}$ according to the computed value for m_i. Then, it iterates over each tuple $t_i \in \mathcal{T}$ in this ordering until it finds a tuple $t_i \in \mathcal{T}_E$ such that the next tuple t_{i+1} does not match the user example, i.e., $t_{i+1} \notin \mathcal{T}_E$. Then it uses the value of m_i for t_{i+1} to add a filter to the current query so that t_i is included and t_{i+1} is excluded. The system can then explain that the provided query refinement returns values in the top-k with $i{=}k$ for measure m_i. The same is done twice both for ascending and descending order.

Percentile-based refinement. The above strategy returns groups of tuples that are notable because of their extreme values.

C. Dest.	C. Origin	Year	SUM
Germany	**Syria**	2013	0.3M
France	Syria	2013	0.3M
Sweden	Syria	2013	0.2M
Germany	China	2013	0.1M
France	China	2013	0.1M
Sweden	China	2013	0.3M
Germany	**Syria**	2014	0.6M
France	Syria	2014	0.3M
Sweden	Syria	2014	0.4M
Germany	China	2014	0.1M
France	China	2014	0.3M
Sweden	China	2014	0.2M

Itemsets:	*Vectors:*
t_E : ⟨Germany , Syria⟩	⟨0.3 , 0.6⟩
t_1 : ⟨France , Syria⟩	⟨0.3, 0.3⟩
t_2 : ⟨Sweden , Syria⟩	⟨0.2, 0.4⟩
t_3 : ⟨Germany , China⟩	⟨0.1, 0.1⟩
t_4 : ⟨France , China⟩	⟨0.1, 0.3⟩
t_5 : ⟨Sweden , China⟩	⟨0.3, 0.2⟩

Features:	*Top-2 similar (σ):*
⟨Year:2013, Year:2014⟩	t_2 (1) ; t_4 (0.99)

Figure 5: Example of similarity computation.

Yet, sometimes, other interesting tuples are not located in the top-k (or bottom-k). Therefore, we propose an alternative refinement strategy that, given a set of tuples $\mathcal{T}$ and a measure m_i, computes a set of percentile values for m_i, e.g., the 90th percentile, the 75th, and so on, and checks in which intervals (e.g., between 90th and 75th) there exists some tuple matching the user example $t_i \in \mathcal{T}_E$. Then, for all such intervals, a different query is generated using those values as filtering conditions. The system can then inform the user that the provided query refinement returns values within the selected percentile for m_i.

6.3 Example-Driven Similarity Search

The last refinement strategy (Sim), solving Problem 2c, is closely connected to the concept of example-based search through implicit similarity [30]. This strategy aims at limiting the query processing to the few (top-k) members within a dimension (or set of dimensions) that are *most similar* to the member identified by the user example. For instance, given the example ⟨Germany⟩ and a measure m_i (see Figure 3 for the Similarity Search step), we produce a refinement that reports only those k countries that are the most similar to Germany based on the values of the measure m_i at the current aggregation level. Note that the effect of this method and the way in which examples are taken into account are not the same that are used in the context of the subset-based refinement. The similarity-based refinement searches for the k entities most similar to those initially provided, while the subset refinement, presented above, first determines subsets and then exploits the examples only to validate such subsets.

In practice, this method works as follows. Assume a query $\mathbf{Q}$ with dimensions $\mathbf{D}(\mathbf{Q})=\langle \delta_1, ..., \delta_m, ..., \delta_n \rangle$, where dimensions $\{\delta_i | i \in [1, m]\}$ correspond to dimensions matching the example tuple $t_E=\langle d_1, ..., d_m \rangle$, while dimensions $\delta_m + 1, ..., \delta_n$ have been added as a result of some refinement. Then, the set of additional items $\mathcal{T}'$ is composed of all existing combinations of members for the dimensions $\delta_1, ..., \delta_m$. For instance, given the initial example tuple t_E:⟨Germany, Syria⟩ and assuming the user is refining the current query $\mathbf{Q}$:⟨Country Destination, Country Origin, Year⟩, with result $\mathcal{T}$ (see Figure 5), we have that the set of additional items $\mathcal{T}' \sqsubseteq \mathcal{T}$ contains ⟨France, China⟩ as well as ⟨Germany, China⟩ (as well as many others) since only the first two dimensions match the user example, while the third has been added in a refinement step (solving Problem 2a).

Then, we compute the similarity between t_E and the members of $\mathcal{T}'$ to extract the k most similar tuples $\mathcal{T}'_k \subseteq \mathcal{T}'$ according to a target measure m_i. Therefore, the combination of member values for all the remaining dimensions $\delta_{(m+1)}, ..., \delta_n$ is treated as a feature set and the feature value is the value for m_i (or zero if a specific combination does not appear). Therefore, for both the example tuple t_E and the tuple $t \in \mathcal{T}'$, we create a feature vector $\overline{\mathbf{v}}_t$ having size equal to the number of distinct values appearing for $\delta_{(m+1)}, ..., \delta_n$ in the result tuples $\mathcal{T}$ of $\mathbf{Q}$. For instance, in our previous example, this corresponds to having a feature for each distinct member of the Year dimension. Finally, for every pair $\langle t_E, t' \rangle \in \{t_E\} \times \mathcal{T}'$, we compute the similarity between the corresponding vectors $\overline{\mathbf{v}}_{t_E}$ and $\overline{\mathbf{v}}_{t'}$ with some vector similarity (in our case the cosine similarity) and keep the top-k most similar tuples. For instance, in Figure 5, this corresponds to tuple pairs ⟨Sweden, Syria⟩ and ⟨France, China⟩ which observed similar trends in the two years ⟨2013, 2014⟩ that are used as features. Therefore, the query refinement consists of adding that combination of values as filters for the dimensions Country Destination and Country Origin.

7 EXPERIMENTS

Here, we show both the effectiveness and appropriateness of our method with user interviews, a comparison to existing methods, and experiments on real and synthetic datasets. First, we show that despite the REQ problem suffering from high complexity in general, in our case our approach allows fast response time (below 2.5 seconds on average) in practice. Then, we study the performance of our query refinement methods and the effect on it of the performance of the underlying SPARQL endpoint. Furthermore, we study the expressiveness of our framework showing the wide range of exploration paths it can support. Moreover, we verify the efficiency of the system's bootstrapping phase. Finally, we demonstrate that for exploratory analytics, existing SPARQL reverse engineering methods do not produce useful answers. Moreover, interviews with real users validate the for our proposal of example-driven analytics.

7.1 Performance Analysis

Experimental setup. Experiments are run on a virtual machine with 62GB of RAM, and 8 2.3GHz cores. Our system is implemented as a server application, written in Java (to be released as open-source) implementing all the algorithms, data structures, and methods described above. The server sends SPARQL queries to a standard RDF triplestore (Virtuoso v7.2). We note that the server, at startup, is only provided with the address of the SPARQL endpoint, the list of named graphs to query, and the RDF class identifying the observations. *No other information about the dataset is assumed* since all the extra information is obtained by the system by automatic crawling. Finally, the triplestore employs a traditional full-text index to provide a faster response time for the task of resolving keywords to IRIs (Algorithm 1, line 3).

Datasets. We employed three real, publicly available, RDF graphs: *Eurostat* [11], *Production* [17, 21], and a subsample of *DBpedia* (from March 2020) describing entities of type *Creative Work* (see Table 3 and Figure 6). Eurostat describes open data asylum applications and is used in the literature as a typical example of a statistical KG [11]. Production records macro-economic information about materials, energy, and monetary production across 43 countries for more than 160 industries, and 200 products or services. This data is currently in use by domain experts in a framework of open-source environmental assessments [16]. Finally, DBpedia represents an analytical view [7] describing songs categorized by genre, artist, label, instrument, and director, automatically extracted from DBpedia by retaining all IRIs

Table 3: Datasets characteristics

	|D|	|M|	$\|\mathcal{H}\|$	|L|	$\|\mathcal{N}_D\|$	Disk (GB)	VGraph (MB)
Eurostat	4	1	8	9	373	1.1	72
Production	7	1	5	9	6444	2.0	73
DBpedia	5	1	14	23	87160	3.8	79

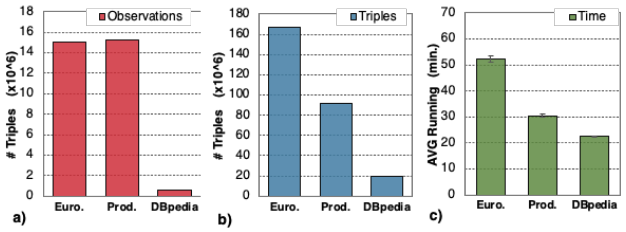

Figure 6: Dataset Size as: (a) # of Observations, and (b) # of Triples; and (c) Time for bootstrapping.

that are instances of the class Creative Work and performing a bi-directional BFS at depth 3.

Eurostat and Production both have ~15M observations, but Eurostat has almost twice as many triples as Production (~160M vs. ~90M; Figure 6a,b), i.e., Eurostat has a richer set of observation attributes. DBpedia, instead, has ~541K observations but ~20M triples, this is mainly due to an extremely complex set of connections interpreted as hierarchies. Interestingly, the DBpedia dataset has a prevalence of hierarchy steps with M-to-N cardinalities (e.g., a song can be associated with multiple genres). Moreover, the three datasets differ greatly in the number of members across all the dimension levels ($|\mathcal{N}_D|$). Overall, the DBpedia dataset, being extracted from an open-domain KG, is the most heterogeneous in structure. *Thus, the performance results on DBpedia can be considered a worst-case scenario, rather than a typical use case.*

ReOLAP Query Synthesis. We generated 4 sets of inputs with a variable number (between 1 and 4) of examples entities (marked as 1 *Ex.* to 4 *Ex.* in the charts). The number of example entities corresponds to the size of the input, e.g., the input ⟨Germany⟩ is of size 1, while ⟨Germany, 2014⟩ is of size 2. In particular, we randomly selected dimension members from each dimension and combined them. We created 10 input queries, i.e., tuples of entities for each size, and we measured the total time required by the system to produce a set of queries derived from that input.

Overall ReOLAP running time: *Results show that time grows proportionally to the size of the input* from 100 − 400ms for size 1 to 2 − 5secs for size 4, depending on the dataset (Figure 7a). Both for Eurostat and Production, we report reverse engineering times on average below or around 1 second with size 1 or 2, and around 2 seconds with size 3 or 4. *Therefore, the system can easily support real-time interactive exploration of real datasets.* Finally, for DBpedia running times start around ~100 msecs with input of size 1 and grows up to ~6 secs for size 4. It is important to note that the schema for the DBpedia dataset contains a high number of dimensions sharing similar values (e.g., the genre of artists and the genre of production labels). Hence, the search step is often comparing large combinations of interpretations. In contrast, this does not happen so often in most real-world statistical KGs. *Nonetheless, comparing the running times to the dataset statistics confirms that the running time for ReOLAP depends on the total number of dimension members (see Table 3), rather than on the absolute size of the dataset (Figure 6a), making it scalable even to large amounts of observations.*

Number of output queries: Figure 7b presents the average number of queries synthesized depending on the size of the input. With input of size 1 or 2, the number of queries is largely below 10 across all datasets. *This ensures that the user will need to inspect only very few alternative interpretations.* With a larger input size, only in some rare cases, we obtain more than 10 reverse-engineered queries. *Comparing these results with Table 3, we see that the number of hierarchies and the number of members shared in different levels (e.g., country of destination and origin) are generally the determining factor.*

Query Refinement We tested each refinement method with the 40 queries synthesized in the previous experiment.

Effect of disaggregation: *The running times for generating the refined queries are below 100ms across all datasets and queries. This shows the efficiency in assisting interactive exploration.* Nonetheless, all the subsequent refinement methods depend on obtaining the results of the refined queries. Hence, we additionally studied the running time for the endpoint to answer the SPARQL query obtained by this step (Figure 8), since this is part of the user exploration workflow. Figure 8a depicts *the running time* of the initial (Orig.) query obtained through ReOLAP and of the refinements obtained by applying 1 (Dis.1) and 2 (Dis.2) Disaggregate steps. The running time for the initial query varies according to the size of the input: with size 1 the triplestore needs to compute a high-level aggregation over one single dimension, while with a larger input size, the resulting query is noticeably more selective, hence faster. With more dimensions (i.e., after Dis.1/Dis.2), the running time increases more prominently for queries generated from input of size 1.

Figure 8b describes the average number of result tuples for each generated query (see Section 5). For Production, with input of size 4, we see that adding more dimensions typically does not increase the number of tuples, these are cases where there is only one (or none) observation for the given combination of entities. *This demonstrates how the disaggregation step can provide access to many facets of the data and allows to verify whether the dataset at hand contains any relevant information in just a few seconds.*

Subset Refinement: In Figure 9a we present running times for both the refinement based on top-k selection (Top-k) and the refinement based on percentile selection (Perc.) as described in Section 6.2. We test them with the queries obtained, after ReOLAP, after applying 1 (Dis.1) and 2 (Dis.2) Disaggregate steps, since they contain larger numbers of tuples. Here we see that the processing time is generally below 1 second and scales linearly to the number of tuples to process. In Figure 9b, we report the number of alternative query refinements produced on average by both Top-k and Percentile. By design, the Top-k method returns two refinements (for ascending/descending order) for each measure and aggregation function. The percentile method (Perc.) instead, produces a variable number of refinements, which is still proportional to the number of measures but by definition depends also on how the query results are clustered. *Therefore, the proposed methods introduce almost no delay in the exploration process while providing a huge advantage to the user that can quickly obtain query refinements to explore distinctive portions of the data.*

Similarity Search: Running time for the similarity search (Sim.) refinement is presented in Figure 9a, where apply the method to the same queries as for the previous two refinements described above. This is the most computationally expensive refinement method and its running time depends on the number of total tuples (not only those matching the examples) returned by the

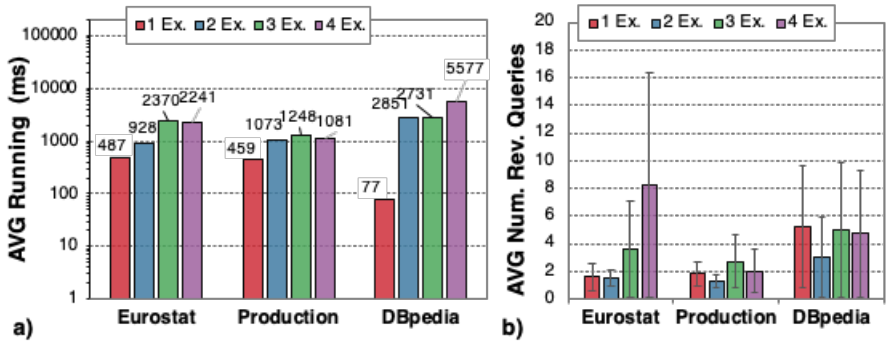

Figure 7: REOLAP: (a) running time; (b) queries retrieved.

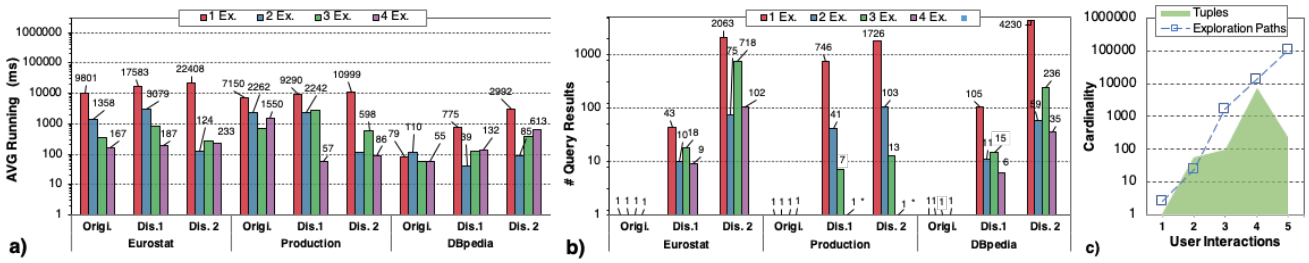

Figure 8: (a) Running time and (b) number of results per query, varying number of input examples. (c) Evolution of exploration workflow. Orig. is the query result from REOLAP, Dis.1 and Dis.2 are the queries after 1 and 2 applications of Disaggregate.

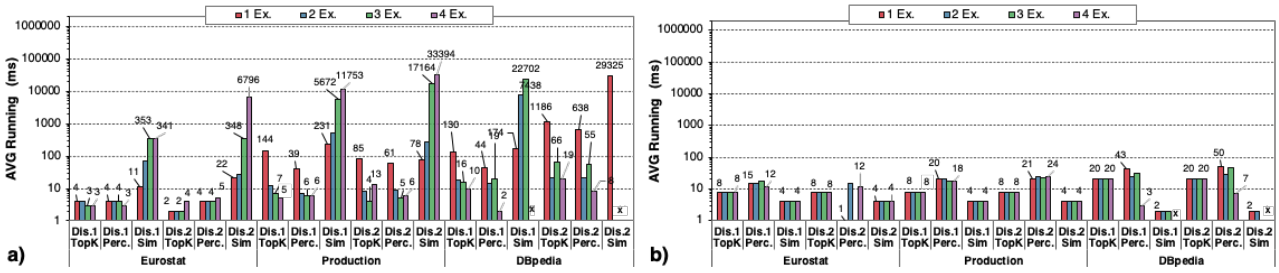

Figure 9: (a) Query running time and (b) number of refinements produced, varying number of input examples. Dis.1 and,Dis.2 are the queries after 1 and 2 applications of Disaggregate.

current selection. Nonetheless, this method runs in less than 1 second with inputs of sizes 1 and 2. For DBpedia, given the M-to-N hierarchies, many of the randomly generated queries were producing extremely large resultsets (e.g., by combining every artist with every musical instrument or with every subgenre), and with inputs of size 3 and 4 members, the SPARQL endpoint reached the timeout (set to 15 minutes). This highlights a challenge for this specific refinement method when analyzing highly unstructured KGs. This also requires optimized query processing at the database level, which is an orthogonal area of research [23, 42, 44]. *However, for a typical KG scenario, the similarity search method provides query refinements with only little (~1 sec) additional processing time.* Finally, Figure 9b shows that this refinement returns a fixed number of reformulations.

Exploration workflow: We consider an example workflow starting with REOLAP, then Disaggregate twice, then Similarity Search and finally TopK refinement. In Figure 8c we see (for Eurostat) that at the first interaction (REOLAP step) given the single input example we offer 4 query interpretations. Each query interpretation gives access to a different view of the data, which we call an exploration path. Applying Disaggregate to any of them (i.e., selecting one path) returns a set of refinements corresponding to new exploration paths (i.e., queries) each giving access to many (usually aggregate) tuples. We report the cumulative number of exploration paths and tuples the system gives access to with just this small sequence of operations, that is, how many different aspects of the data the user can access with just a few iterations. For instance, after 4 interactions the system gives access to 12.000 distinct paths and a total of 8000 tuples, while each TopK reformulation (5th interaction) filters out tuples and each filter corresponds to a new exploration path. *This provides further evidence of the expressiveness of our framework as it provides the user with a wide range of different exploration paths.*

System Bootstrapping. We investigated the time required by the system to bootstrap (Figure 6c). *This operation is done only once offline and its main operation is to build the Virtual Schema Graph. Moreover, if the schema does not change and only new data*

a)
```
SELECT  *  WHERE {
    ?x  olap:memberOf  schema:year .
    ?y olap:memberOf schema:continent . }
```

b)
```
SELECT ?citizen_continent ?refPeriod_year sum(?obsValue)
WHERE {
    ?hq prop:citizen / schema:inContinent ?citizen_continent .
    ?hq dimension:refPeriod / schema:inYear ?refPeriod_year .
    ?hq measure:obsValue ?obsValue .
} GROUP BY ?citizen_continent ?refPeriod_year
```

Figure 10: Queries obtained from the example ⟨"Asia", "2011"⟩ with (a) SPARQLByE [8] and (b REOLAP

is added, all the in-memory data structures are updated efficiently without the need for re-computation. Running time varies from ∼25 minutes for DBpedia up to ∼60 minutes for Eurostat. We note that the bootstrap time is influenced mainly by the complexity of the schema and not by the number of observations. *We conclude that it is both the number of dimension members and attributes that affect this phase, and in all cases, the triplestore performance in serving the data is the determining factor and dominates the bootstrap time.*

7.2 Appropriateness of the Solution

Comparison with SPARQLByE. Here, we show an illustration of the fact that REOLAP and SPARQLByE [8] (the state-of-the-art SPARQL reverse engineering method) solve two different problems. Thus, we compare the queries returned by the two systems when given equivalent inputs. In particular, SPARQLByE and REOLAP accept the same input format, but SPARQLByE's reverse engineering algorithm is designed to return the minimal BGP pattern that covers the user example. For instance, for input ⟨"Asia", "2011"⟩, SPARQLByE recognizes that the two are members of the levels Continent and Year (Figure 10-a) but it does not connect them to IRIs of observations. Instead, providing an example for an observation, SPARQLByE returns an empty resultset, because it does not navigate connections with 2 or more hops. REOLAP *instead, explicitly navigates the complete structure of the dataset to retrieve those connections, and is moreover able to query measure values and instantiate group-by and aggregate operators* (Figure 10-b).

Understanding User Exploration Needs. We recruited 8 volunteers (4 researchers in Computer Science and 4 domain experts in Environmental Science) and asked them to fill out an anonymous questionnaire regarding their data exploration workflow. Then, to investigate user behavior more in-depth, in relation to data-exploration tasks, we invited 4 users (2 researchers from Computer Science and 2 domain expert from Environmental Science) in a 1-to-1 session with a preliminary prototype of our system. The prototype implemented the functionalities described in this work, with a few differences. It provided (i) a data profiling functionality, returning general information and statistics about the dataset (e.g., listing the available dimension and the number of distinct members); (ii) it was missing the subset query refinement based on top-k (see Section 6); and it offered in its place (iii) a clustering-based refinement [48]. As a result, we identified two groups of common information needs: (1) to compute max and min values within distinct groupings of data points, and (2) to compare values for known entities alongside other values that provide some context. Moreover, we highlight how all users commonly refer to some entities of interest from the domain of the dataset as a starting point of their investigations.

From the interviews, we also noticed a striking difference in how people with a CS background (especially familiar with SQL) were more keen on exploring the data-profiling information and recognized the need for top-k query functionalities (as in Section 6). On the other hand, users from Environmental Science preferred to immediately investigate specific entities of interest, e.g., one user reported "I would expect it to contain information about China's electricity production, and I want to see other countries with similar production". Moreover, after some interaction, also the CS users started to ask similar questions, e.g., "I come from this country, I'm really curious to see the sums for my country compared to the other". Finally, we also asked the users to evaluate methods providing more complex filtering conditions based on clustering [49]. Users reported that methods providing complex filtering conditions were not useful since they could not understand the rationale behind them.

This preliminary study supports our problem definitions and the proposed refinement methods based both on simple subset methods and on example-driven search.

8 CONCLUSIONS AND FUTURE WORK

In this work, we analyze for the first time and we describe RE2XOLAP, the first solution for example-driven exploratory analytics over statistical KGs. It consists of REOLAP, a query-by-example algorithm for synthesizing analytical queries in SPARQL and a suite of *intuitive and explainable* query refinement methods for exploratory analytics. This method enables users to perform example-driven analytics of statistical KGs without the need for them to formulate a query in SPARQL. Our experiments demonstrate the applicability and effectiveness of our approach in semi-automatically synthesizing exploratory analytical queries over large statistical KGs. In particular, REOLAP depends only on the complexity of the schema and is thus able to provide response times of a few seconds even for large KGs. Furthermore, our refinement methods allow the user to easily navigate a wide range of exploratory paths identifying distinct facets of the data.

Moving forward, we identify the need to evaluate how to provide user-friendly natural language descriptions of the queries that are produced by the system and to study whether more complex refinement techniques can be added. Moreover, our current approach does not support complex use cases where the user is interested in contrasting the measure values of two different sets of examples or where the user provides instead a set of negative examples. Furthermore, for cases in which many different refinements are produced, we envision the need for a method for ranking the suggested query reformulations to help the user prioritize among them. Finally, we draw attention to the poor performance of existing systems in executing complex analytical queries and identify the need for improving analytical query executions within triplestores.

ACKNOWLEDGMENTS

Matteo Lissanrini is supported by the EU's H2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 838216. Katja Hose was supported by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, and the Poul Due Jensen Foundation.

REFERENCES

[1] Alberto Abelló, Oscar Romero, Torben Bach Pedersen, Rafael Berlanga, Victoria Nebot, Maria Jose Aramburu, and Alkis Simitsis. 2015. Using semantic web technologies for exploratory OLAP: a survey. *TKDE* 27, 2 (2015), 571–588.

[2] Dritan Bleco and Yannis Kotidis. 2019. Using entropy metrics for pruning very large graph cubes. *Information Systems* 81 (2019), 49–62.

[3] R. Castro Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. 2018. Aurum: A Data Discovery System. In *ICDE*. 1001–1012.

[4] Dario Colazzo, François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. 2014. RDF analytics: lenses over semantic graphs. In *WWW*. ACM, 467–478.

[5] Richard Cyganiak, Dave Reynolds, and Jeni Tennison. 2014. The RDF data cube vocabulary. *W3C Recommendation, W3C (Jan. 2014)* (2014).

[6] Yanlei Diao, Paweł Guzewicz, Ioana Manolescu, and Mirjana Mazuran. 2021. Efficient Exploration of Interesting Aggregates in RDF Graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 392–404.

[7] Yanlei Diao, Ioana Manolescu, and Shu Shang. 2017. Dagger: Digging for interesting aggregates in RDF graphs. In *ISWC*.

[8] Gonzalo Diaz, Marcelo Arenas, and Michael Benedikt. 2016. SPARQLByE: Querying RDF data by example. *PVLDB* 9, 13 (2016), 1533–1536.

[9] Ahmed El-Roby, Khaled Ammar, Ashraf Aboulnaga, and Jimmy Lin. 2016. Sapphire: Querying RDF data made simple. *PVLDB* 9, 13 (2016), 1481–1484.

[10] Lorena Etcheverry and Alejandro A Vaisman. 2012. QB4OLAP: a new vocabulary for OLAP cubes on the semantic web. *Proceedings of COLD* (2012).

[11] Lorena Etcheverry and Alejandro A Vaisman. 2017. Efficient Analytical Queries on Semantic Web Data Cubes. *Journal on Data Semantics* 6, 4 (2017), 199–219.

[12] Sébastien Ferré. 2017. Sparklis: an expressive query builder for SPARQL endpoints with guidance in natural language. *Semantic Web* 8, 3 (2017), 405–418.

[13] Luis Galárraga, Kim Ahlstrøm Jakobsen, Katja Hose, and Torben Bach Pedersen. 2018. Answering Provenance-Aware Queries on RDF Data Cubes Under Memory Budgets. In *ISWC 2018*. 547–565.

[14] Enrico Gallinucci, Matteo Golfarelli, Stefano Rizzi, Alberto Abelló, and Oscar Romero. 2018. Interactive Multidimensional Modeling of Linked Data for Exploratory OLAP. *Information Systems* (2018).

[15] Irene Garrigós, Jesús Pardillo, Jose-Norberto Mazón, and Juan Trujillo. 2009. A conceptual modeling approach for OLAP personalization. In *International Conference on Conceptual Modeling*. Springer, 401–414.

[16] Agneta Ghose, Katja Hose, Matteo Lissandrini, and Bo Weidema Pedersen. 2019. An Open Source Dataset and Ontology for Product Footprinting. In *ESWC*.

[17] Agneta Ghose, Matteo Lissandrini, Emil Riis Hansen, and Bo Pedersen Weidema. [n.d.]. A core ontology for modeling life cycle sustainability assessment on the Semantic Web. *Journal of Industrial Ecology* ([n. d.]), 1–16. https://doi.org/10.1111/jiec.13220

[18] François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. 2020. RDF graph summarization for first-sight structure discovery. *The VLDB Journal* 2 (2020).

[19] Nurefsan Gür, Jacob Nielsen, Katja Hose, and Torben Bach Pedersen. 2017. GeoSemOLAP: Geospatial OLAP on the Semantic Web Made Easy. In *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee, 213–217.

[20] Nurefşan Gür, Torben Bach Pedersen, Esteban Zimányi, and Katja Hose. 2018. A foundation for spatial data warehouses on the semantic web. *Semantic Web* 9, 5 (2018), 557–587.

[21] Emil Riis Hansen, Matteo Lissandrini, Agneta Ghose, Søren Løkke, Christian Thomsen, and Katja Hose. 2020. Transparent Integration and Sharing of Life Cycle Sustainability Data with Provenance. In *The Semantic Web – ISWC 2020*. Springer International Publishing, 378–394. https://doi.org/10.1007/978-3-030-62466-8_24

[22] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. 2014. Towards Exploratory OLAP Over Linked Open Data - A Case Study. In *BIRTE*. 114–132.

[23] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. 2015. Overview of data exploration techniques. In *SIGMOD*. 277–281.

[24] Mohsin Iqbal, Matteo Lissandrini, and Torben Bach Pedersen. 2022. A foundation for spatio-textual-temporal cube analytics. *Information Systems* (2022), 102009. https://doi.org/10.1016/j.is.2022.102009

[25] Christian S Jensen, Torben Bach Pedersen, and Christian Thomsen. 2010. Multidimensional databases and data warehousing. *Synthesis Lectures on Data Management* 2, 1 (2010), 1–111.

[26] Mikael R Jensen, Thomas Holmgren, and Torben Bach Pedersen. 2004. Discovering multidimensional structure in relational data. In *DaWaK*. Springer, 138–148.

[27] Manas Joglekar, Hector Garcia-Molina, and Aditya G Parameswaran. 2017. Interactive Data Exploration with Smart Drill-Down (Extended Version). *TKDE* 1 (2017), 1–1.

[28] Shahan Khatchadourian and Mariano P Consens. 2010. ExpLOD: summary-based exploration of interlinking and RDF usage in the linked open data cloud. In *ESWC*. 272–287.

[29] Matteo Lissandrini, Davide Mottin, Themis Palpanas, Dimitra Papadimitriou, and Yannis Velegrakis. 2015. Unleashing the Power of Information Graphs. *ACM SIGMOD Record* 43, 4 (Feb. 2015), 21–26. https://doi.org/10.1145/2737817.2737822

[30] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. 2018. *Data Exploration Using Example-Based Methods*. Synthesis Lectures on Data Management, Vol. 10. Morgan & Claypool Publishers. 1–164 pages.

[31] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. 2018. X2Q: Your Personal Example-based Graph Explorer. In *Proceedings of the Conference in Very Large Databases (PVLDB), 11 (12) 2018* (Rio, Brazil) *(VLDB '18)*. ACM, New York, NY, USA, 2026–2029. https://doi.org/10.14778/3229863.3236251

[32] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. 2020. Graph-Query Suggestions for Knowledge Graph Exploration. In *Proceedings of The Web Conference 2020*. 2549–2555. https://doi.org/10.1145/3366423.3380005

[33] Matteo Lissandrini, Torben Bach Pedersen, Katja Hose, and Davide Mottin. 2020. Knowledge graph exploration: where are we and where are we going? *ACM SIGWEB Newsletter* Summer 2020 (2020), 1–8.

[34] Matteo Lissandrini, Torben Bach Pedersen, Katja Hose, and Davide Mottin. 2022. Knowledge Graph Exploration Systems: are we lost?. In *12th Conference on Innovative Data Systems Research, CIDR 2022*. www.cidrdb.org.

[35] Steffen Metzger, Ralf Schenkel, and Marcin Sydow. 2017. QBEES: query-by-example entity search in semantic knowledge graphs based on maximal aspects, diversity-awareness and relaxation. *Journal of Intelligent Information Systems* 49, 3 (2017), 333–366.

[36] Tova Milo and Amit Somech. 2016. REACT: Context-Sensitive Recommendations for Data Analysis. In *SIGMOD*. 2137–2140.

[37] Tapio Niemi, Jyrki Nummenmaa, and Peter Thanisch. 2001. Constructing OLAP cubes based on queries. In *DOLAP*. ACM, 9–15.

[38] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. Industry-scale knowledge graphs: Lessons and challenges. *ACM Queue* 17, 2 (2019), 48–75.

[39] Kiril Panev and Sebastian Michel. 2016. Reverse Engineering Top-k Database Queries with PALEO. In *EDBT*. 113–124.

[40] Dennis Pedersen, Karsten Riis, and Torben Bach Pedersen. 2002. A powerful and SQL-compatible data model and query language for OLAP. In *Australasian Database Conference*, Vol. 5.

[41] Fotis Psallidas, Bolin Ding, Kaushik Chakrabarti, and Surajit Chaudhuri. 2015. S4: Top-k Spreadsheet-Style Search for Query Discovery. In *SIGMOD*. 2001–2016.

[42] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2021. Optimizing SPARQL Queries using Shape Statistics. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*. OpenProceedings.org, 505–510.

[43] W3C RDF Working Group. 2014. *Resource description framework*. http://www.w3.org/RDF/.

[44] Tomer Sagi, Matteo Lissandrini, Torben Bach Pedersen, and Katja Hose. 2022. A design space for RDF data representations. *The VLDB Journal* (2022), 1–27.

[45] Babak Salimi, Johannes Gehrke, and Dan Suciu. 2018. Bias in OLAP Queries: Detection, Explanation, and Removal. In *ICMD*. ACM, 1021–1035.

[46] Sunita Sarawagi. 2000. User-adaptive exploration of multidimensional data. In *PVLDB*, Vol. 2000. 307–316.

[47] Stefan Schmid, Cory Henson, and Tuan Tran. 2019. Using Knowledge Graphs to Search an Enterprise Data Lake. In *ESWC*.

[48] Thibault Sellam and Martin Kersten. 2016. Cluster-driven navigation of the query space. *TKDE* 28, 5 (2016), 1118–1131.

[49] Thibault Sellam and Martin Kersten. 2016. Ziggy: Characterizing query results for data explorers. *PVLDB* 9, 13 (2016), 1473–1476.

[50] Juan Sequeda and Ora Lassila. 2021. Designing and Building Enterprise Knowledge Graphs. *Synthesis Lectures on Data, Semantics, and Knowledge* 11 (2021).

[51] Wei Chit Tan, Meihui Zhang, Hazem Elmeleegy, and Divesh Srivastava. 2017. Reverse Engineering Aggregation Queries. *PVLDB* 10, 11 (2017), 1394–1405.

[52] Bo Tang, Shi Han, Man Lung Yiu, Rui Ding, and Dongmei Zhang. 2017. Extracting top-k insights from multi-dimensional data. In *SIGMOD*. 1509–1524.

[53] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2014. Query Reverse Engineering. *The VLDB Journal* 23, 5 (2014), 721–746.

[54] Georgia Troullinou, Haridimos Kondylakis, Evangelia Daskalaki, and Dimitris Plexousakis. 2015. RDF digest: Efficient summarization of RDF/S KBs. In *ESWC*. 119–134.

[55] Georgia Troullinou, Haridimos Kondylakis, Matteo Lissandrini, and Davide Mottin. 2021. SOFOS: Demonstrating the Challenges of Materialized View Selection on Knowledge Graphs. In *SIGMOD*. Association for Computing Machinery, New York, NY, USA, 2789–2793.

[56] Jovan Varga, Alejandro A Vaisman, Oscar Romero, Lorena Etcheverry, Torben Bach Pedersen, and Christian Thomsen. 2016. Dimensional enrichment of statistical linked open data. *Web Semantics: Science, Services and Agents on the World Wide Web* 40 (2016), 22–51.

[57] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. 2015. SeeDB: efficient data-driven visualization recommendations to support visual analytics. *PVLDB* 8, 13 (2015), 2182–2193.

[58] Mussab Zneika, Claudio Lucchese, Dan Vodislav, and Dimitris Kotzinos. 2016. Summarizing linked data RDF graphs using approximate graph pattern mining. In *EDBT*. 684–685.