The VLDB Journal (2022) 31:851–876
https://doi.org/10.1007/s00778-021-00704-2

SPECIAL ISSUE PAPER

ABSTAT-HD: a scalable tool for profiling very large knowledge graphs

Renzo Arturo Alva Principe[1] · Andrea Maurino[1] · Matteo Palmonari[1] · Michele Ciavotta[1] · Blerina Spahiu[1]

Received: 21 February 2021 / Revised: 15 July 2021 / Accepted: 7 September 2021 / Published online: 29 September 2021
© The Author(s) 2021

Abstract
Processing large-scale and highly interconnected Knowledge Graphs (KG) is becoming crucial for many applications such as recommender systems, question answering, etc. Profiling approaches have been proposed to summarize large KGs with the aim to produce concise and meaningful representation so that they can be easily managed. However, constructing profiles and calculating several statistics such as cardinality descriptors or inferences are resource expensive. In this paper, we present ABSTAT-HD, a highly distributed profiling tool that supports users in profiling and understanding big and complex knowledge graphs. We demonstrate the impact of the new architecture of ABSTAT-HD by presenting a set of experiments that show its scalability with respect to three dimensions of the data to be processed: size, complexity and workload. The experimentation shows that our profiling framework provides informative and concise profiles, and can process and manage very large KGs.

Keywords Knowledge graph · Data management · Data quality · Data profiling · Distributed processing engine

1 Introduction

Knowledge Graphs (KGs) are used to represent relationships between different entities such as people (e.g., Tom Hanks), places (e.g., Rome), events (Pitchfork Festival), and so on [24]. They organize knowledge in graph structures where the meaning of the data is encoded alongside the data in the graph. RDF is a data model for representing KGs that come with an ecosystem of languages and protocols to foster interoperable data management. In RDF, graph nodes represent *entities*, identified by URIs, or *literals* (e.g., strings, numbers, etc.); edges represent *relations* between entities or between entities and literals, which are identified by RDF *properties*. Entities and literals are associated with types (classes, e.g., `dbo:City` or datatypes, e.g., `xmls:integer`). The sets of possible types and properties are organized into *ontologies*, which specify the meaning of the used types and properties through logical axioms. Intuitively, ontologies provide the schema of the KG, but, remarkably, data and schema are loosely coupled in KGs, with potential mismatches and diverging evolution along time. In addition, KGs are often very large and evolve along time. The classical example of this evolution is the Linked Data Cloud[1,2] which has evolved with roughly 1255 datasets as of February 2021.

KGs support several data-intensive tasks related to data management, information integration, natural language processing, and inference in research and industry [31]. Eventually, they feed, often in combination with machine learning methods, an increasing number of downstream applications such as recommender systems [21] and question answering interfaces [12]. Existing edges can also be used to train soft inference models, e.g., based on *knowledge graph embeddings*, so as to predict missing or probable arcs based on latent features. These models have been used for biomedical applications [29,30], e.g., to predict drug–target interactions. For downstream applications, and, especially, for applications that combine KGs and machine learning, it is important to support domain experts with a clear picture of the content, structure and quality of the input KG. Low-quality or mis-

✉ Blerina Spahiu
blerina.spahiu@unimib.it

Renzo Arturo Alva Principe
renzo.alvaprincipe@unimib.it

Andrea Maurino
andrea.maurino@unimib.it

Matteo Palmonari
matteo.palmnonari@unimib.it

Michele Ciavotta
michele.ciavotta@unimib.it

1 Department of Informatics, Systems and Communication, University of Milano-Bicocca, Viale Sarca, 336, Milan, Italy

1 https://www.w3.org/RDF/
2 https://lod-cloud.net/

interpreted input data may lead to unreliable output models, as expressed by the well-known colloquial motto "garbage in–garbage out."

ABSTAT[3] is a data profiling approach [46] and tool [35] introduced to let users explore the content and structure of large KGs and also inspect potential quality issues. ABSTAT takes an RDF dataset, and (optionally) an ontology (used in the dataset) as input, and computes a semantic profile. The semantic profile consists of a summary, which provides an abstract, but complete description of the dataset content, and some statistics.

The informative units of ABSTAT's summaries are Abstract Knowledge Patterns (AKPs), named simply *patterns* in the following, which have the form (`subjectType, pred, objectType`).

Patterns represent the occurrence of triples < `sub, pred, obj`> in the data, such that `subjectType` is the most specific type of the subject and `objectType` is the most specific type of the object [46].

For example, the pattern (`dbo:SoccerPlayer, dbo:team, dbo:SoccerClub`) represents the occurrence of triples that represent entities of type `dbo:SoccerPlayer` linked with entities of type `dbo:SoccerClub` through the property `dbo:team`. The types `dbo:SoccerPlayer` and `dbo:SoccerClub` are the most specific types for the respective entities, which may have also more generic types such as `dbo:Athlete` or `dbo:Person` and `dbo:SportsClub` or `dbo:Organisation`.

The most specific type is computed with the help of the ontology. Such a choice allows ABSTAT to have a compact but complete summary, by excluding several more generic redundant patterns using the ontology.

ABSTAT profiles can be explored by users through its web interface or by machines using APIs. Rich profiles as the ones computed in ABSTAT support automatic feature selection for semantic recommender systems [15,36], vocabulary suggestions for data annotation, as in [40], and help in the detection of quality problems [45,46].

When processing large KGs, as it is often the case for downstream machine learning tasks, it is critical to reduce the latency between processing the graph and the availability of the results. Such latency is often a result of platform startup costs (e.g., MapReduce [34]) or the complexity of graph processing algorithms.

Sometimes the time needed to compute the results may reach several hours or even days, up to the eventual failure of the computation. From a recent survey on the challenges of large graph processing [39], scalability is the most pressing challenge faced by all participants, who reported problems in processing very large graphs efficiently. The reported limitations include inefficiencies in loading, updating, and performing computations on large graphs. Processing real-world graphs often surpasses the capability of single computers.

A solution to these challenges is switching to the distributed computing paradigm and deploying large graph processing algorithms on a collection of computing nodes, whose configuration can fit storage and computing resources according to the end users' requirements. However, it has been reported that many algorithms at the core of the existing techniques are not ready to be implemented on top of today's graph processing infrastructures, which rely on horizontal scalability [37], with many algorithms being inherently sequential and difficult to parallelize.

Several approaches have been proposed to adapt models and frameworks for graph processing to the distributed computing paradigm [2,32,33,50].

In the domain of KG management and profiling, Sansa is the most notable example of a natively distributed solution. It provides a unified framework for several downstream tasks such as link prediction, knowledge base completion, querying, reasoning and, also, profiling [25]. Similarly to ABSTAT, it has a modular architecture and provides to the end user 32 RDF statistics (such as the number of triples, RDF terms, properties per entity, and usage of vocabularies across datasets), and apply quality assessment in a distributed manner. However, Sansa profiling is not based on a summarization model; ABSTAT profiling digs deeper into semantic features of the KG and makes use of ontologies associated with the data.

Among the several statistics calculated by ABSTAT, one of the most unique and hardest to compute are cardinality descriptors. Cardinality descriptors provide information about the relationships between subjects and objects of the triples at the *pattern level*,which reveal valuable information about the data. For example, cardinality statistics for the pattern (`dbo:Company, dbo:keyPerson, owl:Thing`) in DBpedia 2015-10 reveal that 5263 different entities of type `dbo:company` are connected with a unique entity of type `owl:Thing` through the property `dbo:keyPerson` [45].

We could identify that such entity is `dbr:Chief_executive_officer` which in DBpedia does not have a type, thus defying purely syntactic vocabulary-based statistics. This generic entity is used as placeholder for 5263 companies with unspecified CEO, including `dbr:Kodak, dbr:Telefónica, dbr:Allianz`, and many others. In semantic-aware ABSTAT profiles, such an entity is associated with the default upper type `owl:Thing` and featured as outlier in the cardinality estimated for the pattern, leading to spotting the anomaly. What would have happened if we had trained a link predictor using these triples as positive samples? For large KGs, cardinality descriptors, which have also been proved useful in recommender systems [15,36], could

[3] http://abstat.disco.unimib.it/

be computed only with extremely large execution times or could not be computed at all.

In this paper, we present ABSTAT-HD, that is, *Highly Distributed ABSTAT*. The framework supports the distributed computation of ontology-based summaries and profiles of very large KGs, thus overcoming scalability problems of our previous solution. To the best of our knowledge, this is the first KG profiling approach that supports summarization on a distributed computing infrastructure. The framework supports full ABSTAT profiles that include novel features such as cardinality descriptors, which were used in previous work [15] but never properly defined within our model. In addition, the framework supports *property-based minimalization*, *pattern inference*, and *instance count statistics*, new features that complete the strategy adopted in ABSTAT to remove redundant patterns and better count the data represented by the patterns. In conclusion, we can summarize the main contributions of this paper with respect to the previous work as follows:

- A formal and complete definition of the summarization model which is the backbone of ABSTAT tool.
- A new algorithm based on the relational model for calculating the summary model.
- ABSTAT-HD, a highly distributed and scalable tool for processing and producing profiles for very large RDF graphs.
- A set of experiments that show the scalability of ABSTAT-HD with respect to the previous version of ABSTAT.
- A report about quality issues found in the very large Microsoft Academic Knowledge Graph, to provide more qualitative insights into the informativeness of our profiles.

This paper is organized as follows: Sect. 2 formally introduces the ABSTAT summarization model, while in Sect. 3 we present the process of profile construction and in Sect. 4 the architecture of ABSTAT-HD. A large set of experiments over ABSTAT-HD to evaluate the scalability using existing large KGs under different controlled hardware configurations are discussed in Sect. 5. Section 6 discusses the related work of existing profiling tools for KGs and tools for graph processing while in Sect. 7 we draw conclusions and future work.

2 Profiling model

We first introduce some preliminary definitions needed to explain ABSTAT profiles, then present the summarization and profiling models used in the paper.

2.1 Preliminaries: datasets, assertions, and terminologies

ABSTAT is developed to profile RDF data, which natively represent KGs. In the rest of the paper we define and use the term "dataset" to be equivalent to "KG." In fact, our profiling model is formalized to be applicable to any KG that can be interpreted as a set of triples $< subject, predicate, object >$, and where entities (individuals) and literals are associated with types. Ontologies, which formally specify the terminology used to describe the entities, are leveraged to make profiles more compact. Ontologies for RDF data are usually specified with axioms of the RDFS and OWL2 languages,[4] which are interpreted as Description Logics (DLs) axioms [47].

We define a dataset (equivalently, a KG) by borrowing the definition of Knowledge Base in DLs, i.e., as consisting of a *terminology* (TBox in DLs—intuitively, the schema) and a set of *assertions* about individuals (ABox in DLs— intuitively, the actual data).

Definition 1 *Dataset:* A dataset $\Delta = (\mathcal{T}, \mathcal{A})$ is a pair, where $\mathcal{T}$ is a set of terminological axioms, and $\mathcal{A}$ is a set of assertions.

We define more in details $\mathcal{A}$ first, being the actual data the focus of our profiles, and $\mathcal{T}$ afterward, which supports the profiling process. Since DLs are tractable fragments of the well-known First-Order Logics (FOL), we find more convenient to present our model using a FOL notation for axioms in $\mathcal{A}$ and $\mathcal{T}$.

We use symbols like C, to denote types (unary predicates in FOL), symbols like P, Q to denote properties (binary predicates in FOL), and symbols a,b to denote named individuals or literals (constants in FOL).

Assertions in $\mathcal{A}$ are of two kinds: *typing assertions* having the form $C(a)$, and relational assertions having the form $P(a, b)$, where a is a named individual (or, simply, *individual*) and b is an individual or a literal. We denote the sets of typing and relational assertions by $\mathcal{A}^C$ and $\mathcal{A}^P$, respectively. We consider $C(a) \in \mathcal{A}$ whenever we find RDF triples of the form $< a, \texttt{rdf:type}, C >$, where a and C are URIs, or $< a, P, b\text{^^}C >$, where b is a literal and C its datatype. In addition, we assign to each untyped individual or literal occurring in $\mathcal{A}$ a default type, that is, respectively, `owl:Thing` or `rdfs:Literal`. A literal occurring in a triple can have at most one type (because typing is implicitly encoded in triples like $< a, P, b\text{^^}C >$). Conversely, an individual can have many types. A *relational assertion* $P(a, b)$ is any triple $< a, P, b >$ such that $P \notin M^P$, where M^P is a set of predicates that are reserved for modeling purposes. In this set we include `rdf:type` and all the predicates used to model the

[4] https://www.w3.org/TR/owl2-syntax/

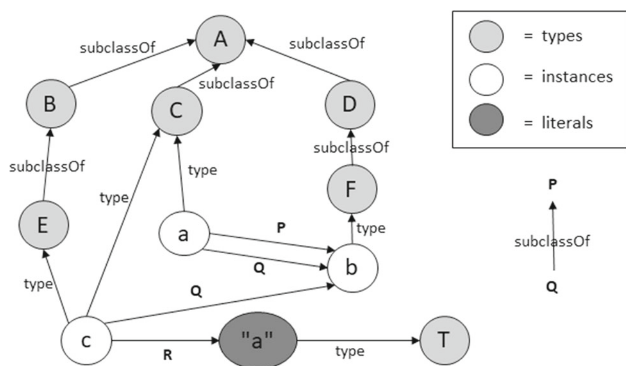

Fig. 1 A small graph representing a dataset

terminology (e.g., `rdfs:subClassOf`, `rdfs:domain`, any other predicate that is not considered relevant for the profile).

The terminology $\mathcal{T}$ may contain an arbitrary set of axioms, but our profiling model uses only axioms specifying that C is subtype of D (*subtype axioms*) and P is subproperty of Q (*subproperty axioms*), which can be expressed by the FOL formulas $\forall x(C(x) \rightarrow D(x))$ and $\forall x, y(P(x, y) \rightarrow Q(x, y))$, respectively. We apply a *completion* of $\mathcal{T}$ inspired by OWL2 semantics. We inject the types and properties that occur in $\mathcal{A}$ into $\mathcal{T}$ and specify their upper types and properties: all named classes and datatypes occurring in $\mathcal{A}$ are subtype, respectively, of `owl:Thing` and `rdfs:Literal`; all the properties that have some object that is an individual are subproperties of `owl:TopObjectProperty` and all the properties that have some object that is a literal are subproperties of `owl:TopDataProperty`. Therefore, if an ontology is not associated with a dataset to be profiled, we can still consider default $\mathcal{T}$ built from $\mathcal{A}$.

With $V^{\mathcal{T}}$ we refer to the terminology-level vocabulary of a dataset, which consists of a set $\mathtt{N}^C$ of *types* (which always include `owl:Thing` and `rdfs:Literal`) and a set $\mathtt{N}^P$ of properties (which always include `owl:TopObjectProperty` and `owl:TopDataProperty`).

2.2 Ontology-based summarization

Abstract Knowledge Patterns (AKPs), equivalently referred to as *patterns* in the rest of the paper, represent schema-level patterns used to model assertions about individuals in a given domain. In particular, we consider patterns that model the existence of entities with certain properties and can be formalized by existentially quantified formulas in FOL.

Definition 2 *Patterns:* A pattern is a triple (C, P, D), such that C and D are types and P is a property, which is interpreted by the FOL formula $\exists x \exists y(C(x) \wedge D(y) \wedge P(x, y))$.

Intuitively, an existential pattern, states that there are individuals of type C that are linked to individuals or literals of a type D by a predicate P.

Our goal is to summarize a dataset, and, more specifically, the assertions $\mathcal{A}$, by defining a set of patterns that represent the full content of $\mathcal{A}$ in a compact way. Profiles will add statistics about the assertions represented by each pattern to the summaries.

Definition 3 *Patterns and represented assertions*: A pattern (C, P, D) *represents* a relational assertion $P(a, b) \in \mathcal{A}$ iff there exist a set $\{C(a), D(b), P(a, b)\} \subseteq \mathcal{A}$.

We denote with $\Pi^{\mathcal{A}}$ the set of patterns that represent all the relational assertions in $\mathcal{A}$. To make summaries compact, we observe that many of the patterns that represent a relational assertion can be inferred from a small subset of more specific patterns if we consider constraints between types and properties specified in the terminology $\mathcal{T}$. Consider the example graph depicted in Fig. 1 where typing assertions and subclass/subproperty constraints are depicted as arcs between nodes, and upper level types and properties are omitted (namely, `owl:Thing`, `rdfs:Literal`, `owl:TopObjectProperty` and `owl:TopDataProperty`). The assertion $P(a, b)$ is represented by many different patterns, where $\exists x \exists y(C(x) \wedge F(y) \wedge P(x, y))$ and $\forall x(C(x) \rightarrow A(x))$ obviously imply $\exists x \exists y(A(x) \wedge F(y) \wedge P(x, y))$, as well as $\exists x \exists y(A(x) \wedge D(y) \wedge P(x, y))$ and so forth via subtype axioms. Based on this principle, the model used in basic ABSTAT summaries and presented in previous work [46] would consider (C, P, F) as the one most specific representative pattern for $P(a, b)$.

Here we complete the model by solving the unbalanced treatment of types and properties, where also properties can have dependencies specified by subproperty axioms. In the example, the pattern (C, Q, F) is even more specific than (C, P, F) because $\forall x(Q(x, y) \rightarrow P(x, y))$, which also let us infer $P(a, b)$ from $Q(a, b)$, i.e., $P(a, b)$ is redundant in the set of assertions based on the terminology. Therefore, we need to generalize our previous model by defining a *subpattern relation* over the patterns to represent that in a pair of patterns one is more specific than the other.

For simplicity, we introduce a *terminology graph* $G^{\mathcal{T}}$ as a proxy that represents relations among types/properties specified by axioms in $\mathcal{T}$. By posing $|\sim$ as a relation between a terminology and subtype/subproperty relations derived from it, we define a terminology graph as follows.

Definition 4 *Terminology Graph*: A terminology graph is the disjoint sum [48] of two posets: a type poset $(\mathtt{N}^C, \preceq^{G^{\mathcal{T}}})$ such that $\mathtt{N}^C$ is a set of types and for all $C, D \in \mathtt{N}^C$, $C \preceq^{G^{\mathcal{T}}} D$ iff $\mathcal{T}| \sim C \preceq^{G^{\mathcal{T}}} D$; a property poset $(\mathtt{N}^P, \preceq^{G^{\mathcal{T}}})$ such that $\mathtt{N}^P$ is a set of properties and for all $P, Q \in \mathtt{N}^P$, $P \preceq^{G^{\mathcal{T}}} Q$ iff $\mathcal{T}| \sim P \preceq^{G^{\mathcal{T}}} Q$.

To specify the relation $|\sim$ we can rely either on explicit or on inferred axioms in $\mathcal{T}$. We prefer the first strategy because of practical reasons: some web ontologies may have unintended inferences that can mess up the intended type and property hierarchies. Similar reasons also suggest us ignoring equivalence relations between named classes and properties, which frequently introduce counter-intuitive inferences (e.g., node collapse), like further discussed in previous work [46].[5]

Now we can transfer the order relation $\preceq^{G^{\mathcal{T}}}$ from the terminology graph to the patterns, by defining a product partial order $(\mathtt{N}^C \times \mathtt{N}^P \times \mathtt{N}^C, \preceq^{G^{\mathcal{T}}})$ that can be interpreted as a *subpattern relation* as defined below.

Definition 5 *Subpatterns*: A pattern (C, P, D) is a *subpattern of* a pattern (C', Q, D') wrt. a terminology graph $G^{\mathcal{T}}$, denoted by $(C, P, D) \preceq^{G^{\mathcal{T}}} (C', Q, D')$ iff $C' \preceq^{G^{\mathcal{T}}} C$, $D' \preceq^{G^{\mathcal{T}}} D$ and $Q \preceq^{G^{\mathcal{T}}} P$.

Observe that this poset has, by definition, two upper level patterns: (`owl:Thing`, `owl:TopObjectProperty`, `owl:Thing`) and (`owl:Thing`, `owl:TopDataProperty`, `rdfs:Literal`). The subpattern relation is eventually used to select, for some input relational assertion, those patterns that are more specific, i.e., minimal in the subpattern poset, among the patterns that represent it.

We observe that some relational assertions can be inferred from other relational assertions and the property poset $(\mathtt{N}^P, \preceq^{G^{\mathcal{T}}})$, e.g., $P(a, b)$ can be inferred from $Q(a, b)$ whenever $Q \preceq^{G^{\mathcal{T}}} P$.

Let us consider the strict order relations $\prec^{G^{\mathcal{T}}}$ that are the irreflexive counterparts of the posets induced by $\preceq^{G^{\mathcal{T}}}$, where $X \prec^{G^{\mathcal{T}}} Y$ imply that $X \neq Y$ whatever X and Y are (types, properties or patterns). We say that a relational assertion $P(a, b) \in \mathcal{A}$ is *redundant* (based on $G^{\mathcal{T}}$) if and only if there exist some relational assertion $Q(a, b) \in \mathcal{A}$ such that $Q \prec^{G^{\mathcal{T}}} P$. Since the property poset is finite, there are minimal properties that ensure that, given a redundant assertion $P(a, b)$, we can always define a set of relational assertions from which $P(a, b)$ can be inferred.[6] We refer to this set of non-redundant assertions from which $P(a, b)$ can be inferred as the $G^{\mathcal{T}}$-inference base of $P(a, b)$.

Definition 6 *Minimal Patterns*: A pattern π is a *minimal pattern* for a relational assertion $P(a, b) \in \mathcal{A}$ and a terminology graph $G^{\mathcal{T}}$ iff one of the two following conditions applies: 1—$P(a, b)$ is not redundant and π represents $P(a, b)$ and there does not exist a pattern π' that represents $P(a, b)$ such that $\pi' \prec^{G^{\mathcal{T}}} \pi$; 2—$P(a, b)$ is redundant and π represents some assertion $Q(a, b)$ such that $Q(a, b)$ is in the $G^{\mathcal{T}}$-inference base of $P(a, b)$ and there does not exist a pattern π' that represents $Q(a, b)$ such that $\pi' \prec^{G^{\mathcal{T}}} \pi$.

Observe that in the first case a minimal pattern will have the form (C, P, D), while in the second case it will have the form (C, Q, D) with $Q \prec^{G^{\mathcal{T}}} P$

In the rest of the paper we use the following expressions: a pattern π *minimally represents* a relational assertion $P(a, b) \in \mathcal{A}$ (*under* a terminology graph $G^{\mathcal{T}}$), iff π is a *minimal pattern* for $P(a, b)$ and $G^{\mathcal{T}}$. Conversely, we say that $P(a, b)$ *is minimally represented* (under a terminology graph $G^{\mathcal{T}}$) by all patterns that minimally represent it. By applying pattern minimalization to a set of relational assertions (with an input terminology), we obtain the set of patterns that minimally represent all of them, also referred to as its *Minimal Pattern Base (MPB)*. A summary consist in a terminology graph and an MPB for an input dataset $\Delta = (\mathcal{T}, \mathcal{A})$.

Definition 7 *Minimal Pattern Base*: A *minimal pattern base* for a set of assertions $\mathcal{A}$ under a terminology graph $G^{\mathcal{T}}$ is a set of patterns $\Pi^{\mathcal{A},\mathcal{T}}$ such that $\pi \in \Pi^{\mathcal{A},\mathcal{T}}$ iff π minimally represents some $\phi \in \mathcal{A}^{\mathcal{P}}$ under $G^{\mathcal{T}}$.

Definition 8 *Summary*: A *summary* of a dataset $\Delta = (\mathcal{A}, \mathcal{T})$ is a pair $\Sigma = (G^{\mathcal{T}}, \Pi^{\mathcal{A},\mathcal{T}})$ such that: $G^{\mathcal{T}}$ is a *terminology graph* derived from $\mathcal{T}$, $\Pi^{\mathcal{A},\mathcal{T}}$ is a *minimal pattern base* for $\mathcal{A}$ under $G^{\mathcal{T}}$.

Observe that different patterns can be extracted for an assertion $P(a, b)$ if a and/or b have more than one minimal type. However, minimalization is capable to exclude many patterns that can be entailed following the $\preceq^{G^{\mathcal{T}}}$ relation and that do not minimally represent any $P(a, b)$.

For example, the MPB for the dataset in Fig. 1, includes the patterns (E, Q, F), (E, R, T), (C, Q, F), (C, R, T), that is, only four of the twenty-four patterns that represent the assertions (in this count we excluded patterns including upper types and properties—omitted in the figure). The MPB excludes patterns like (B, Q, D) and (C, Q, D), but also (C, P, F), as a result of considering properties in the minimalization process and extending the relation $\preceq^{G^{\mathcal{T}}}$ over N^P.

Although very few ontologies make use of subproperty relations intensively, we believe that minimalization wrt both type and property hierarchy is important to generalize the model and to provide a more robust summarization mechanism for future scenarios. However, to provide users with flexible configuration choices, minimalization over properties is optional and can be disabled keeping only type-based minimalization.

Definition 7 extends the definition of minimal patterns that considers type-based minimalization [46]. Observe that

[5] It is easy to verify that the model can be easily adapted to work with equivalence classes of types and properties, rather than individual types and properties.

[6] It is unlikely but not impossible that an assertion $P(a, b)$ can be inferred from more than one assertions; as an example consider the case where $Q(a, b) \in \mathcal{A}$,$R(a, b) \in \mathcal{A}$, $Q \prec^{G^{\mathcal{T}}} P$,$R \prec^{G^{\mathcal{T}}} P$ with P and R disconnected in $G^{\mathcal{T}}$.

when we minimalize over properties, redundant relational assertions become irrelevant for including patterns in the MPB: the patterns that minimally represent redundant relational assertions are patterns that minimally represent also some not redundant assertions. A similar approach based on the identification of redundant assertions can be applied also to typing assertions, some of which can be inferred from $\mathcal{T}$ and $G^{\mathcal{T}}$ and thus considered redundant. We define redundant typing assertions similarly as we did for redundant relational assertions. Observe that also for a redundant typing assertion $C(a)$ it is always possible to track the set of non-redundant assertions $C(a)$ is inferred from based on $G^{\mathcal{T}}$. The computation of the minimal pattern base will use this intuition and prune redundant relational and typing assertions from $\mathcal{A}$ so as to compute the patterns that represent the non-redundant assertions (we remind that, based on Definition 3, a pattern (C, P, D) represents a relational assertion $P(a, b)$ in an assertion set $\mathcal{A}$ if and only if $\{C(a), P(a, b), D(b)\} \subseteq \mathcal{A}$).

Let $\mathcal{A}^-$ be $\mathcal{A} - \{\phi \mid \phi \in \mathcal{A}$ and is redundant based on $G^{\mathcal{T}}\}$; we refer to $\mathcal{A}^-$ as to the non redundant counterpart of $\mathcal{A}$. Then the following equivalence can be proved (see the Appendix for proof).

Theorem 1 *An MPB $\Pi^{\mathcal{A},\mathcal{T}}$ for a set of assertions $\mathcal{A}$ under a terminology graph $G^{\mathcal{T}}$ is equivalent to the set $\Pi^{\mathcal{A}^-}$ of patterns that represent every relational assertions in $\mathcal{A}^-$.*

The above theorem is also useful to better explain how ABSTAT-HD can be adapted to compute summaries incrementally and deal with changes in the ABox by updating profiles and statistics locally. The key idea is that if changes in the ABox concern redundant assertions profiles and statistics do not change; if changes affect new or non-redundant assertions, the profiles are updated after tracking the assertions that can be inferred from the ones affected by changes (e.g., some assertions may change their status from redundant to non-redundant or vice versa). However, in this paper we focus on the implementation and validation of batch profiling, leaving some additional details about incremental profiling in the Appendix.

2.3 Profiles and statistics

A profile extends a summary of a dataset by associating statistics with its patterns in its vocabulary, referred to as V^{Σ}.

Definition 9 *Profile*: A *profile* of a dataset $\Delta = (\mathcal{A}, \mathcal{T})$ is a pair $(\Sigma^{\mathcal{A},\mathcal{T}}, S)$ such that $\Sigma^{\mathcal{A},\mathcal{T}}$ is a summary with a minimal pattern base $\Pi^{\mathcal{A},\mathcal{T}}$, and S is a set of functions $s : \Pi^{\mathcal{A},\mathcal{T}} \cup V^{\Sigma} \rightarrow \mathbb{R}$.

Pattern statistics are computed on the patterns that are in the summary (i.e., which minimally represent some relational assertion in the dataset) by considering the assertions that they *minimally* represent or represent. While some basic statistics like pattern frequency [46] can be computed by processing one assertion at a time (yet, with scalability problems for very large datasets), new statistics like cardinality descriptors—defined below—require methods to group assertions by their representative patterns. As a consequence, they can be hardly computed without the distributed profiling solution described in this paper also for datasets of relatively smaller size.

Pattern frequency count. The frequency of a pattern π is defined as the number of non-redundant relational assertions it *minimally represents*. Note that the frequency of a pattern is always a value between one and the number of non-redundant relational assertions it represents.

Pattern instances count. Let us first define what we mean with *instances of a pattern* π:

$$inst(\pi) = \begin{cases} \{P(a,b) \in \mathcal{A}^- | \pi \text{ represents } P(a,b)\} & \text{if } sub(\pi) = \{\pi\} \\ \bigcup\limits_{\forall \rho \in sub(\pi)} inst(\rho) & \text{otherwise} \end{cases}$$

where $sub(\pi) = \{\rho \in MPB | \rho \preceq^{G^{\mathcal{T}}} \pi\}$ is the set of subpatterns of π. We therefore define the number of instances for a pattern π as the number of relational assertions in $inst(\pi)$, that is, the number of non-redundant relational assertions represented by π or its subpatterns.

Values for this statistic are always positive and also $\rho \in sub(\pi)$ implies that the number of instances for π will be greater than or equal the number of instances for ρ. Please note that $inst$ is defined $\forall \pi \in MPB$ but it can be easily extended to $\forall \pi \in \{N^C \times N^P \times N^C\}$ i.e., the set of every possible pattern in $\mathcal{T}$. Such extension would admit zero values as there may exist some pattern ψ whose $inst(\psi)$ is empty as may be too specific for the dataset. Moreover, extending this statistic would enable further analysis at many levels of abstraction (e.g., patterns external to the MPB).

Type occurrence count. The number of occurrences for a type C is the number of entities a such that $C(a) \in \mathcal{A}^{\mathcal{C}}$.

Property occurrence count. The number of occurrences for a concept P is the number of relational assertions $P(a, b) \in \mathcal{A}^{\mathcal{P}}$.

Pattern cardinality descriptors. Cardinality descriptors are divided into direct cardinality descriptors and inverse cardinality descriptors. Given a pattern (C, P, D) the maximum (minimum, average) direct cardinality is the maximum (minimum, average) number of distinct entities of type C (in subject position) linked to a single entity of type D through the predicate P. Similarly, the maximum (minimum, average) inverse cardinality is the maximum (minimum, average) number of distinct entities of type D (in object position) linked to a single entity of type C through the predicate P. Intuitively it tells us how the assertions represented by a pattern $\pi = (C, P, D)$ are balanced in terms of links between

individuals in subject position and individuals/literals in object position for π in both directions through P.

3 Profiling process

In this section, we describe the profiling process. First, we present the workflow to construct the summary and the respective statistics for each dataset. Second, we provide the profile creation using the relational model.

```
Input: A^C typing assert., A^P relational assert, G^T terminology graph
Output: the Inf_Patterns
1  Inf_Patterns = ∅;
2  for P(x, y) ∈ A^P do
3  |  Inf_x = allInferredTypes(A^C_x, G^T);
4  |  Inf_y = allInferredTypes(A^C_y, G^T);
5  |  Inf_P = allInferredProperties(P, G^T);
6  |  for C' ∈ Inf_x do
7  |  |  for D' ∈ Inf_y do
8  |  |  |  for Q ∈ Inf_P do
9  |  |  |  |  Inf_Patterns = Inf_Patterns ∪ {(C', Q, D')};
10 return Inf_Patterns;
```

Algorithm 1: Computation of the pattern inference

3.1 Profile creation

The profiling workflow of ABSTAT is depicted in Fig. 2. In a first preprocessing step, the assertion set is extracted, the set $\mathcal{A}^C$ of typing assertions is singled out from the set of relational assertions $\mathcal{A}^\mathcal{P}$, and the terminology graph is created using the input terminology. We then perform three operations: type minimalization (over $\mathcal{A}^C$) and property minimalization (over $\mathcal{A}^\mathcal{P}$), to compute a minimal type set for each entity and remove property redundancy, and type inference to infer all the entity types. We extract minimal patterns and statistics and compute cardinality descriptors. We also use $\mathcal{A}^\mathcal{P}$ and inferred types to infer patterns along the subpattern relation and compute statistics that require inference.

Core-profiling consists in the preprocessing, type minimalization, and pattern calculation steps, *full-profiling* includes also all the other steps.

Preprocessing. Preprocessing is explained with an example in Step 1 of Fig. 3.

Observe that relational assertions are finer-grained classified based on the type of the object in the assertion. Assertions with a named entity in the object are called *object relational assertions* (e.g., < `Cher genre Disco`>) while assertions with a literal in the object are called *datatype relational assertions* (e.g., < `Cher alias "Cher Bono"`>). The terminology graph $G^\mathcal{T}$ is built starting from the `rdfs:subClassOf` and `rdfs:subPropertyOf` relations specified in the terminology and managed with a library for managing OWL2 ontologies.

The graph will be then completed with *external* types, that is, types asserted in $\mathcal{A}^C$ and not included in $\mathcal{T}$, when computing the minimal types.

Type minimalization and property minimalization. For each individual x, we compute the set M_x of minimal types with respect to the terminology graph $G^\mathcal{T}$ as exemplified in Fig. 3. Given x, we select all the typing assertions $C(x) \in \mathcal{A}^C$ and form the set $\mathcal{A}^C_x$ of typing assertions about x. Please refer to our previous paper in [46] for more details about the algorithm on type minimalization.

In Step 2, ABSTAT performs type minimalization. As `Cher` has two types: `MusicalArtist` and `Artist`, and `MusicalArtist` is the subtype of `Artist`, only the former type is included in the patterns. If minimalization on properties is enabled, we remove redundancies from $\mathcal{A}^P$.

Consider Step 3 in Fig. 3, since `alias` $\preceq^{G^\mathcal{T}}$ `alternativeName`, the triple < `Cher alternative Name "Cher Bono"`> is considered redundant as < `Cher alias "Cher Bono"`> is also present in $\mathcal{A}^P$, therefore it is removed.

Minimal pattern base. We then iterate over each relational assertion $P(x, y) \in \mathcal{A}^P$ and get the minimal types sets M_x and M_y. Finally, $\forall C, D \in M_x, M_y$ a pattern (C, P, D) is added to the minimal types pattern base. Step 4 in Fig. 3 takes minimal types and relational assertions as input and computes the patterns. The MPB for the example in Fig. 3 is reported in the bottom box.

Pattern inference. ABSTAT computes the subpattern relation by inferring the patterns that are more generic of the patterns included in the MPB.

Algorithm 1 presents the pseudocode for computing pattern inference. We start initializing $Inf_{Pattenrs}$ to $\emptyset$ (line 1), then for each relational assertion $P(x, y)$ we calculate every inferable type for x and y and every inferable property for P (line 3–5). Notice that at this point Inf_x includes $\mathcal{A}^C_x$, Inf_y includes $\mathcal{A}^C_y$ and Inf_p includes P. Finally, $\forall C', D', Q \in Inf_x, Inf_y, Inf_p$ a pattern (C', Q, D') is added to $Inf_{Pattenrs}$ (lines 6–13). We keep trace of the times that a pattern is added to the $Inf_Patterns$ set to obtain the number of instances.

In Fig. 4, Step 5 shows each entity with its inferred types until `Thing` (or `Literal` for literals) is reached. For example `Funk` passes through `Genre` and `TopicalConcept` before it reaches `Thing`. Note that, unlike as in type minimalization, here we want to extract all the possible types from each entity with the support of $G^\mathcal{T}$. In Step 6, for each assertion in $\mathcal{A}^P$ we get the entities' inferred types and extract the superproperties for each property using $G^\mathcal{T}$ to finally generate the inferred pattern set along with the number of instances. For example <`Cher genre Disco`> generates 5*4*2 patterns (5 types for Cher, 4 types for Disco and 2 properties). Despite the huge number of generated patterns through pattern inference, we still keep only minimal patterns

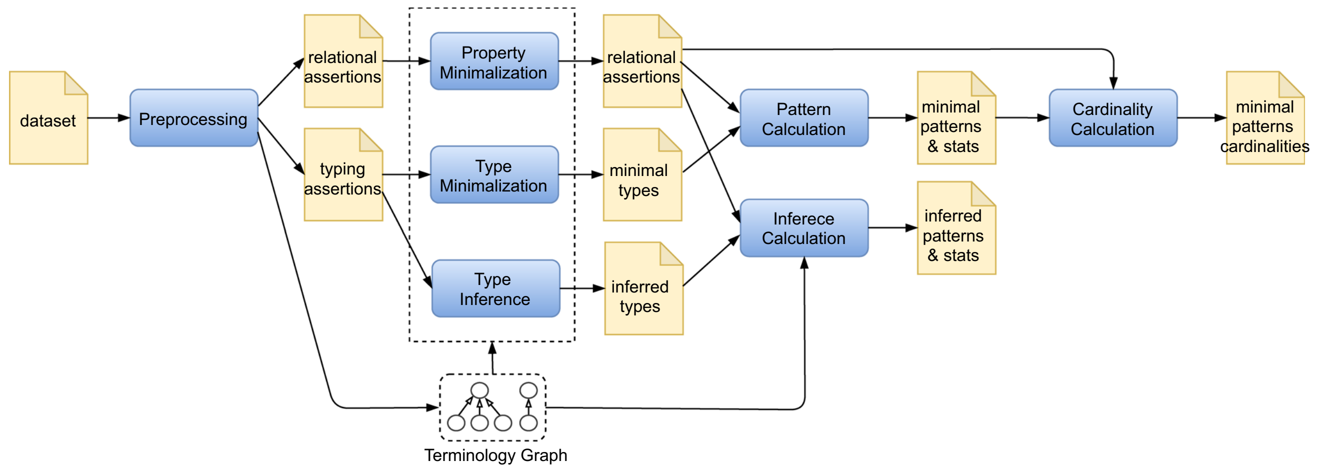

Fig. 2 Profiling workflow

in our summaries but enriching statistics with the number of instances calculated in this phase.

Cardinality descriptors. Algorithm 2 takes as argument the set triples_AKP$_i$, which contains the relational assertions that have AKP$_i$ as minimal pattern. We start by creating a map for the subjects and for the objects that will contain the counts for each subject and object, respectively (lines 1–2). Then for each assertion $P(x, y)$ we count subjects and objects and keep this information on subjects and objects (lines 3–10). Each entry in subjects tells us the number of distinct objects associated with the respective key. Similarly for objects. For direct cardinality descriptors, we calculate the maximum, minimum, and average values for the values of the objects map (11–13). For inverse cardinality descriptors, maximum, minimum and average are calculated for the values of the objects map (lines 14–16). We can think of cardinality descriptors as grouping $\mathcal{A}^P$ assertions by their minimal patterns as depicted in Fig. 4 (Step 7). For each pattern, we can now extract statistics (Step 8) on subjects and objects and thus obtain the cardinality descriptors.

Input: $triples_AKP_i$
Output: the $cardinalities$ for $triples_AKP_i$

```
1  subjects = map();
2  objects = map();
3  for P(x, y) ∈ triples_AKP_i do
4      if not subjects.contains(x) then
5          subjects.put(x, 0);
6      subjects.put(x, subjects.get(x) + 1);
7      if not objects.contains(y) then
8          objects.put(y, 0);
9      objects.put(x, objects.get(y) + 1);
10 cardinalities = list();
11 cardinalities.add(max(objects.values()));
12 cardinalities.add(min(objects.values()));
13 cardinalities.add(avg(objects.values()));
14 cardinalities.add(max(subjects.values()));
15 cardinalities.add(min(subjects.values()));
16 cardinalities.add(avg(subjects.values()));
17 return cardinalities;
```

Algorithm 2: Computation of the pattern cardinality descriptors

3.2 Profile creation via relational model

Algorithms for profile creation shown in the previous section have a linear complexity that makes nearly impossible the definition of a profile for a very large dataset. For this reason, we adopt a relational model approach. By using a relational model approach, it is then possible to implement an algorithm by means of a high scalable engine such as Spark SQL [4].

Let $D(t, s, p, o, d)$ be the original dataset where each triple is enriched with two attributes; attribute t that specifies the type of assertion (typing_assertion, object relational_assertion, datatype relation_ assertion) and attribute d that specifies the datatype of a given literal. In the preprocessing phase, we first create three new relations T_a, O_a, D_a where

$$
\begin{aligned}
T_a &= \sigma_{D.t=\text{"typing"}}(\Pi_{e,t}(D)) \\
O_a &= \sigma_{D.t=\text{"object"}}(\Pi_{s,p,o}(D)) \\
D_a &= \sigma_{D.t=\text{"datatype"}}(\Pi_{s,p,o,d}(D))
\end{aligned}
\quad (1)
$$

where e, t, s, p, o, d represent entity, type, subject, property, object and datatype, respectively.

Then, we apply the property minimalization UDF function PM^{UDF} to both O_a and D_a obtaining two new relations O_a^m, D_a^m. The PM^{UDF} function removes redundant assertions as $P(x, y)$ if exists an assertion $Q(x, y)$ with $Q \preceq^P P$.

The minimize udf M^{UDF} is another UDF function, that takes as input an entity e along with its types and calculates the minimal types mt, and generates a new relation $T_a^m(e, mt)$. The explode UDF function E^{UDF} creates a new row for each type in the type attribute (see Fig. 5).

Minimal patterns are then calculated. For the sake of brevity, we describe the relational queries for object-relational

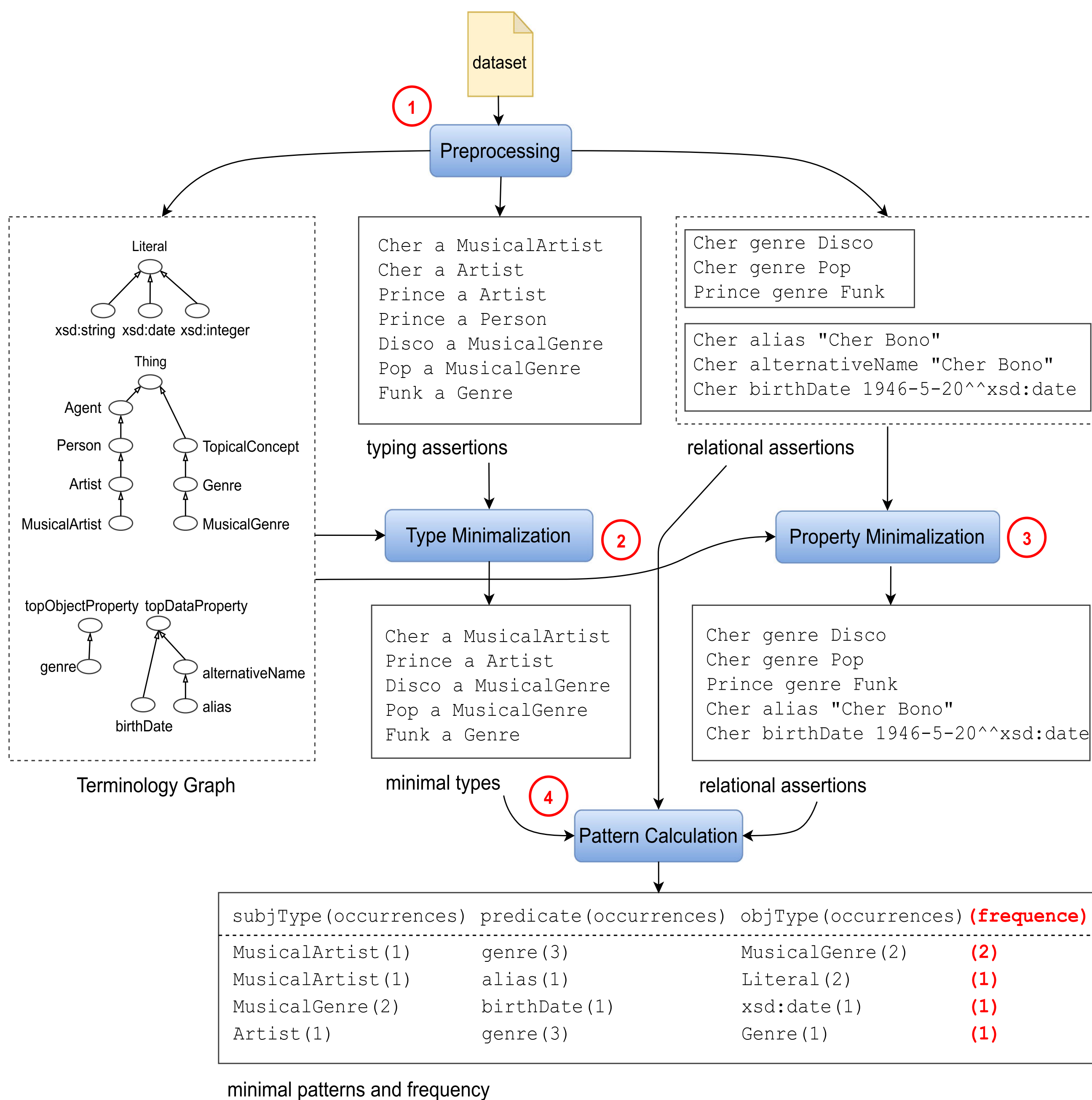

Fig. 3 From preprocessing to pattern calculation

assertions only, as for datatype relational assertions the process is similar. The relational query that calculates the minimal patterns MP relation for the object-relational assertions is defined by the queries in 2 and 3.

$$q1 = \rho_{st \leftarrow mt}((O_a \rtimes_{s=e} T_a^m)$$
$$q2 = \rho_{ot \leftarrow mt}(q1 \rtimes_{o=e} T_a^m))) \qquad (2)$$

In Query 2 the left outer join ($\rtimes$) is applied on the object relational assertions O_a (subject attribute) with the minimal types table T_a^m (e attribute). This Cartesian product generates the new st attribute.

A similar procedure is applied for object attribute. While we join data, projection is applied by removing the subject and object attributes. Following, we rename the minimalType in subjectType as st, and the minimalType in objectType as ot. This produces a relation where each tuple represents a pattern with subject type st, predicate p and object type ot. Certainly this relation will contain duplicates. Observe that this relation contains already the minimal patterns. Hence,

typing assertions

relational assertions

minimal patterns

5 Type Inference

7 Grouping

```
Cher a MusicalArtist
Cher a Artist
Cher a Person
Cher a Agent
Cher a Thing
Prince a Artist
Prince a Person
Prince a Agent
Prince a Thing
Disco a MusicalGenre
Disco a Genre
Disco a TopicalConcept
Disco a Thing
Pop a MusicalGenre
Pop a Genre
Pop a TopicalConcept
Pop a Thing
Funk a Genre
Funk a TopicalConcept
Funk a Thing
```

inferred types

Literal; xsd:string xsd:date xsd:integer; Thing; Agent; Person; Artist; MusicalArtist; TopicalConcept; Genre; MusicalGenre; topObjectProperty topDataProperty; genre; alternativeName; alias; birthDate

Terminology Graph

```
Cher genre Pop
Cher genre Disco
```

MusicalArtist genre MusicalGenre

```
Cher alias "Cher Bono"
```

MusicalArtist alias Literal

```
Cher birthDate
1946-5-20^^xsd:date
```

MusicalGenre birthDate xsd:date

```
Prince genre Funk
```

Artist genre Genre

minimal patterns

6 Inference Calculation

8 Cardinality Calculation

subjType	predicate	objType	(instanceCount)
MusicalArtist	genre	MusicalGenre	(2)
MusicalArtist	alias	Literal	(1)
MusicalArtist	alternativeName	Literal	(1)
MusicalGenre	birthDate	xsd:date	(1)
Artist	genre	Genre	(3)
...	...	...	
Thing	topObjectProperty	Thing	(2)
Thing	topDataProperty	Literal	(3)

inferred patterns and number of instances

subjType	predicate	objType	(dmin-davg-dmax imin-iavg-imax)
MusicalArtist	genre	MusicalGenre	(1-1-1 2-2-2)
MusicalArtist	alias	Literal	(1-1-1 1-1-1)
MusicalGenre	birthDate	xsd:date	(1-1-1 1-1-1)
Artist	genre	Genre	(1-1-1 1-1-1)

pattern cardinalities

Fig. 4 Pattern inference, instances count and cardinality descriptors calculation workflow

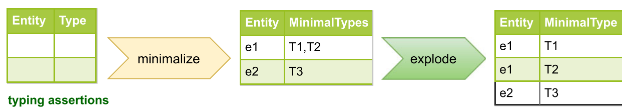

Fig. 5 The minimal types calculation

given a tuple, the number of its duplicates corresponds to the number of assertions it represents.

In Query 3 the group-by operator Γ groups duplicate patterns (same subjectType st, property p and objectType ot) and then counting is performed. The result is the frequency freq of a given pattern.

$$\mathrm{MP} = \rho_{\mathrm{freq} \leftarrow \mathrm{COUNT}(st,p,ot)}(\Pi_{st,p,ot,\mathrm{COUNT}(st,p,ot)} (\Gamma_{st,p,ot}(q2)) \quad (3)$$

Queries 4 and 5 show cardinality descriptors calculation, where x stands for "MIN," "MAX," "AVG" operators.

Let $\mathrm{AKP_{SO}}(\mathrm{AKP}, s, o)$ be a relation schema with subject and object attribute and an AKP attribute. AKP attribute is obtained by string concatenation of the subject type, predicate and object type fields extracted by MP relation created in Query 3.

$$\begin{aligned} \mathrm{AKP}^o &= \rho_{\mathrm{count} \leftarrow \mathrm{COUNT}(o)}(\Pi_{\mathrm{AKP},o,\mathrm{COUNT}(o)} \\ &\quad (\Gamma_{\mathrm{AKP},o}(\mathrm{AKP_{SO}}))) \\ \mathrm{AKP}^o_x &= \rho_{x_o \leftarrow X(\mathrm{count})}((\Pi_{\mathrm{AKP},o,X(\mathrm{count})} \\ &\quad (\Gamma_{\mathrm{AKP}}(\mathrm{AKP}^o)))) \end{aligned} \quad (4)$$

As for direct cardinality descriptors, let us consider a pattern π and the set $\mathcal{A}^P_\pi$ of relational assertions it minimally represents. Please note that every assertion in $\mathcal{A}^P_\pi$ has the same predicate. For each object of $\mathcal{A}^P_\pi$ the number of distinct subjects linked through the same predicate is calculated, and afterward the max, min and average is computed

($\text{AKP}^s_{\min}$, $\text{AKP}^s_{\max}$, $\text{AKP}^s_{\text{avg}}$). In a similar way, the inverse cardinality is calculated. For these statistics, the groupby occurs on the subject and counts the number of linked distinct objects. From this we extract the maximum, minimum and the average inverse cardinality.

$$\begin{aligned}\text{AKP}^s &= \rho_{\text{count}\leftarrow\text{COUNT}(ss)}(\Pi_{\text{AKP},s,\text{COUNT}(s)}\\ &\quad(\Gamma_{\text{AKP},s}(\text{AKP}_{\text{SO}})))\\ \text{AKP}^s_x &= \rho_{x_s\leftarrow X(\text{count})}((\Pi_{\text{AKP},s,X(\text{count})}\\ &\quad(\Gamma_{AKP}(AKP^s)))) \end{aligned}\tag{5}$$

Query 6 shows the join operations between the relations calculated in Queries 4 and 5 that creates the final relation containing all cardinality descriptors associated with all patterns.

$$\begin{aligned}C &= \text{AKP}^s_{\min} \bowtie \text{AKP}^s_{\max} \bowtie \text{AKP}^s_{\text{avg}} \bowtie \text{AKP}^o_{\min}\\ &\quad\bowtie \text{AKP}^o_{\max} \bowtie \text{AKP}^o_{\text{avg}}\end{aligned}\tag{6}$$

Finally, the pattern inference and instances count step is very similar to the minimal types and pattern calculation but instead of making the Cartesian product between minimal types, relational assertions, and minimal types, Cartesian product is calculated between inferred types, relational assertions and inferred types. A UDF I^{UDF} uses a terminology graph. Each type encountered will become one of the inferred types including "seed" types. Result of the I^{UDF} function is then exploded by means of the E^{UDF} producing the $I(e, it)$ relation where for each entity e the inferred type it is reported.

$$\begin{aligned}q3 &= \rho_{st\leftarrow mt}((O_a \rtimes_{s=e} I)\\ q4 &= \rho_{ot\leftarrow mt}(q3 \rtimes_{o=e} I)))\end{aligned}\tag{7}$$

Following the same approach for pattern calculation, we have Queries 7 and 8 with the difference that the pattern generated will not be minimal anymore. In this case the frequency statistic that is calculated coincides with the number of instances statistic.

$$\begin{aligned}PI &= \rho_{\text{freq}\leftarrow\text{COUNT}(st,p,ot)}\\ &\quad(\Pi_{st,p,ot,\text{COUNT}(st,p,ot)}\\ &\quad(\Gamma_{st,p,ot}(q4))\end{aligned}\tag{8}$$

3.3 Complexity

In this section, we first calculate the time complexity of different stages of the workflow (Fig. 2) and then estimate the global complexity by considering the contribution of each stage to the whole workflow. Notice that the following assumption holds true that the workflow is implemented using the standard library *Spark SQL* provided by the Apache Spark distributed processing engine (more implementation details are given in Sect. 4.2.3). Consequently, for the complexity of select–project–join operations, we adapt the cost model described in [6,27] with some simplification due to the fact that we use a purely cloud-based solution.

The first operation is the creation of relations described in Query 1 where three selection–projection sub-queries are performed. Let n be the number of triples stored in the relation D, and w be the number of workers (i.e., agents performing a the query in distributed fashion), the time complexity of Query 1 is $\Theta(n)$ i.e., the computation time linearly depends on the number of triples.

M^{UDF} has a time complexity $\Theta(e)$ as it needs to populate the relation $T^m_a(e, mt)$, while E^{UDF} features a complexity $\Theta(\frac{mt}{e})$ because the function creates a new row if an entity e has more than one minimal type. Thus, the complexity for populating the relation T^m_a is equal to $\Theta(mt)$.

Query 2 comprises two left outer join queries. Subquery q1 is implemented in Spark SQL as SortMergeJoin to limit the memory consumption. According to [27], this join implementation has a time complexity equals to:

$$\begin{aligned}\Theta(|O_a|, |t^m_a|) &= \Theta_s(|O_a|) * \Theta_s(|t^m_a|) *\\ &\quad \log\left(\frac{|O_a|}{w}\right) * \log\left(\frac{|t^m_a|}{w}\right)\\ &\approx \Theta_s\left(\frac{n}{w}\right) * \Theta_s\left(\frac{mt}{w}\right) * \log\left(\frac{n}{w}\right) * \log\left(\frac{mt}{w}\right)\end{aligned}\tag{9}$$

where Θ_s is the time complexity of the shuffle operation. Notice that the cardinality of q2 is the same of q1 ($|q1| = |O_a|$); thus, its complexity is (9) as well.

As described in Sect. 3.2, minimal patterns computation is created by means of Query 3 that uses a groupby operator. Grounding on the analysis of [6], we can assume that its time complexity is $\Theta(\frac{n}{w})$.

To calculate patterns cardinality descriptors, we use Query 4 and 5 that are aggregate and groupby queries over the relation AKP_{SO}. This relation has the same cardinality as the relation MP (|MP|). As a consequence, the time complexity for both queries is $\Theta(\frac{|\text{MP}|}{w})$.

The final step in the profiling workflow is the computation of inferred patterns and instance count. Query 7 and 8 have a similar structure, thus their complexity is $\Theta_s(\frac{n}{w})\Theta_s(e)\log(\frac{n}{w})\log(\frac{e}{w})$.

When the dataset is very large, that is, when $n \gg mt$; $n \gg e$; $n \gg |\text{MP}|$ we can conclude that the complexity of the overall profiling algorithm is:

$$\Theta\left(\frac{n}{w}\log\left(\frac{n}{w}\right)\right)$$

4 ABSTAT: highly distributed

This section presents the architecture of the ABSTAT-HD. We first describe the logical architecture behind the profiling process and then provide an analysis of the scalability issues that served as motivation for the distributed version. Following, we present the new distributed builder and its deployment.

4.1 Architecture

The diagram reported in Fig. 6 is to be considered a minimal representation of the ABSTAT logical architecture; thus, inessential components like the ones in charge of authentication and authorization activities are not reported. ABSTAT architecture is modular so that it can benefit from the Service Oriented Architecture model and the main components are:

- *Viewer*, which provides a graphical user interface to interact with ABSTAT functionalities. A configuration wizard drives the user in the choice/upload of datasets/ontologies along with a configuration setup for further processing. Once the execution has ended, the user can explore computed profiles using the interface for constrained queries (requesting, for instance, the desired subject with or without a predicate and object) and full-text search. In addition, controls for the managing of profiles, datasets, and ontologies are provided.
- *Builder*, which is the core module that executes the profiling algorithms. It takes as input a dataset (in N3 format) and possibly an ontology (in OWL format) along with the user's profile configuration. The configuration received from the Builder contains all the user choices about which step to execute in the profiling pipeline. Both input and output profiles are saved in the Data Lake.
- *Data Loader*, which main task is to feed the storage engines intended for user consultation. After a semantic profile is computed, the Data Loader reads the profile from the Data Lake (internal data model) and maps it in a suitable way for uploading into databases (e.g., MongoDB) or indexing into search engines (currently exploiting Apache Solr). Furthermore, it also creates a copy of the input datasets into Virtuoso triple-store.
- *Explorer*, which exposes a collection of APIs to support profile exploration requests from Viewer or authenticated third-party applications. Examples of such APIs include the Browse API, which provides a subject (predicate, object) constraint consultation of the profiles, the Search API for full-text search functionality over patterns, concepts, and properties, the Autocomplete API for concept/predicate suggestion based on our patterns and, finally, the Validate API, which allows the user to inspect pattern instances with possible data quality issues.

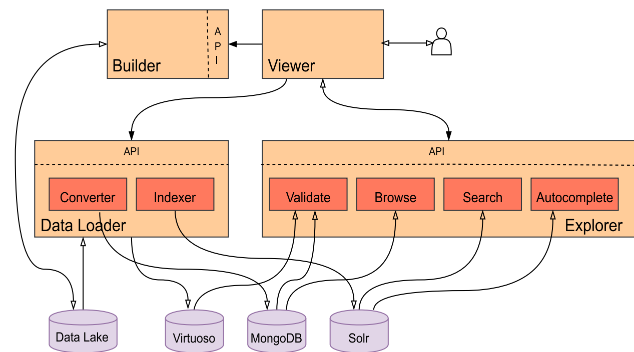

Fig. 6 ABSTAT architecture

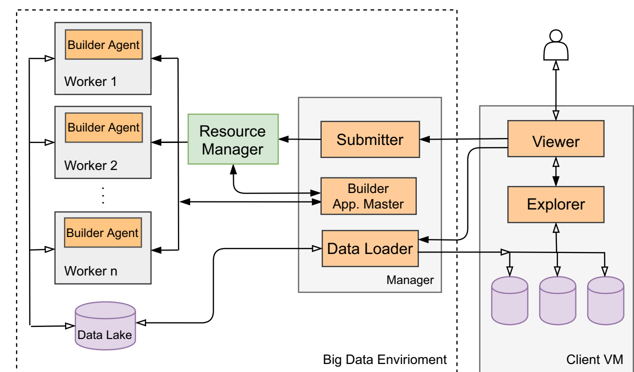

Fig. 7 ABSTAT-HD: architecture and deployment

The modularity of this architecture enhances the flexibility of the components during deployment and their maintainability, which are central features for further extensions.

4.2 ABSTAT-HD builder

In the following, we first discuss the main issues posed by the previous centralized architecture; then, we present the new distributed Builder and the Big Data framework adopted to support its deployment and execution.

4.2.1 Scalability issues

At this state, ABSTAT can compute profiles for small datasets and allows users even on commodity hardware to compute profiles for their confidential data. However, the complexity of the statistics to be calculated (especially the new ones, viz. cardinality and inference) makes ABSTAT unsuitable for processing complex and large datasets with many millions of triples. These considerations, combined with the awareness that the size and complexity of the LOD Cloud assets are continuously growing, led us to carry out a redefinition of the system aimed at seeking horizontal scalability.

By inspecting ABSTAT architecture, we identified two components that primarily influence the system scalability,

namely, the Builder[7] and the Data Loader. The former is involved in creating profiles; the latter writes and reads data from the Data Lake. The Data Loader scalability issues relate to the data ingestion process; they are not faced in this work as there are plenty of production-grade Big Data solutions for efficiently moving large amounts of data (e.g., Apache Flume).

As for the Builder and the related computation bottleneck issue, we worked to overcome the limitations of the centralized approach.

The Builder implements the workflow in Fig. 2. Each element of this pipeline is implemented separately and in a multi-threaded manner (but with centralized synchronization points); moreover, the code over the years has been optimized as much as possible. We discarded the hypothesis of reimplementing ABSTAT codebase in a more efficient programming language (like C) to reuse as much as possible the available code. We have also experimented with parallelizing the file scan (dividing it into chunks) to eliminate the bottleneck due to sequential disk access. Still, the results were not satisfactory due to severe disk contention. The use of increasingly powerful machines did not solve resource saturation for larger datasets, either. For this reason, it has been decided to re-design the Builder component according to the manager–workers model and execute it in a distributed fashion on a collection of machines, where the workers perform in parallel the computation while the manager supervises the execution. We preferred to use a mature Big Data solution, prized and actively developed to implement this approach, that guarantee maintainability, flexibility, security, and high-level languages to describe the processing pipeline. In particular, these tools offer an off-the-shelf replicated and distributed data lake, the management of computational resources, seamless data shuffling, automatic application deployment, data locality aware task scheduling, and a workflow optimization mechanism. More details on the Big Data environment underpinning ABSTAT-HD are reported in Sect. 4.2.3

4.2.2 Extended components

ABSTAT-HD (whose general architecture and deployment are depicted in Fig. 7) addresses and solves the limitations of the previous version. As aforesaid, we decided to exploit a Big Data environment to manage a cluster of machines and run the computation in a distributed fashion. Such an environment consists of an ecosystem of several interacting components; for simplicity, we will only mention the Data Lake, the Resource Manager, and the Application Framework. Data Lake is in charge of partitioning, distributing, and managing datasets to increase locality and reduce skewness during processing. Resource Manager is responsible

[7] https://bitbucket.org/disco_unimib/abstat-distributed

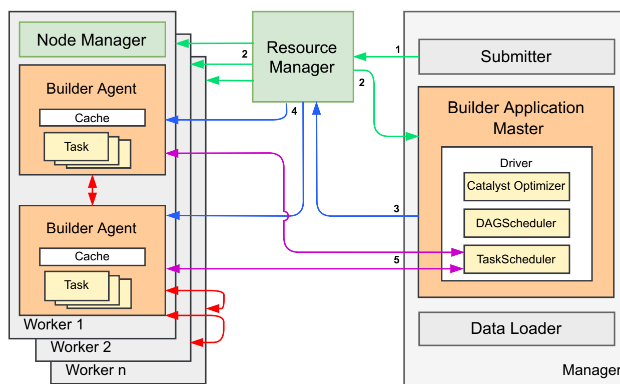

Fig. 8 An overview of ABSTAT-HD Builder's internals

for managing, partitioning, and assigning cluster resources to the applications that require them. Finally, Application Framework consists of a set of classes and libraries to create applications compatible with the particular platform, i.e., communicating with Resource Manager to request the necessary resources and control the program execution flow. Consequently, we adapted some components (namely, Builder and Data Loader) and created a new one (Submitter) to interact with Resource Manager. Other architectural components, like Viewer and Explorer, are kept unchanged. Follows a brief description of those components:

– *Submitter* is a new component. It is a service that implements the interface of the old Builder. Its job is to receive requests from Viewer and submit the Builder executable to the cluster through Resource Manager (green arrow 1 in Fig. 8). Resource Manager distributes the Builder code to the cluster nodes (green arrows 2 in Fig. 8) and executes it. Submitter also checks on the Builder status and exposes it via an API.
– *Builder* has been completely reworked; it is no longer a long-running service but a manager–worker distributed application executing for the time strictly necessary to calculate the profile of a single dataset. The main component of the Builder new architecture (Fig. 8) is the Application Master, which is in charge of task management. The worker component, called Agent, in turn receives and executes tasks on the dataset.. More in detail, the Builder Application Master interacts with Resource manager (blue lines in Fig. 8) to negotiate the access to resource containers (RAM and CPU shares) on the cluster nodes. Within each container, a Builder Agent is then executed. Each Agent receives from the Application Master a set of tasks to perform on specific dataset chunks; partial and final results are stored in a cache structure, but they can be spilled on disk if necessary. As for the Application Master, this consists of a set of modules, among which the *Driver* stands out, which is

in charge of managing the interface with Resource Manager and Agents. It implements the summarization and statistics calculation algorithms exploiting a data frame abstraction and an SQL-like engine over it (both provided by the Application Framework); thus, a dataset is view as a relational table that can be manipulated using relational-algebra operators (Expressions (1) to (8)). A query optimizer module manipulates relational queries (using techniques such as filter pushdown, indexes, bucketing, join type selection, among others) to execute them more efficiently. The resulting query is then compiled into tasks forming a Direct Acyclic Graph (DAG); such structure is analyzed to identify tasks that can be performed in parallel (stages) and data shuffle operations. Eventually, groups of tasks are sent to the Builder Agents by the *TaskScheduler* module for execution (purple lines in Fig. 8). A data shuffle operation is performed at the end of each stage; this is done by sorting local data and distributing them to the other agents according to a partition key (red lines in Fig. 8). Ultimately, the results are returned to Builder Application Master or persisted by Data Lake. Finally, note that the Application Master is enforced to run in a separate node from those of the Agents, co-located with Submitter and Data Loader to reduce skewness.

– *Resource Manager* is a component provided by the Big Data environment. It is in charge of managing the available resources of the underlying cluster, considering several factors ranging from data locality to cluster-level load balance. An application running into the cluster has to interact with this component to access the required resources.
– *Data Loader* is made compatible with the distributed Data Lake (e.g., HDFS) to both write the datasets and read the profiles to be indexed.

4.2.3 Big data environment

Our choice for the ecosystem/framework to achieve horizontal scalability fell on Apache Hadoop[8] with Apache Spark[9] as processing engine and application framework. The Apache Hadoop stack enables the distributed processing of large datasets across computer clusters using high-level programming models. It is designed to detect and handle failures at the application layer, so delivering a highly available service on top of a cluster of computers. The data layer of Apache Hadoop is the Hadoop Distributed File System (HDFS). HDFS splits files into large blocks and distributes them across the cluster nodes. Hadoop resource manager, YARN, then transfers the application code to the nodes for parallel execution. This approach is data locality aware that is, nodes mainly manipulate the data they have direct access to.

Apache Spark has been selected as a distributed computing framework since, unlike the default compute engine Apache MapReduce, it uses node memory better, reducing disk spill and achieving reduced computation times.

Moreover, since the RDF data model can be easily mapped onto a relational table (data frame) with columns "subject," "predicate," "object" and optionally "datatype" (as described in Sect. 3.2), Spark application framework comes in handy as it allows the application to process a dataset in a (distributed) relational fashion, simplifying by far the Builder component development. In particular, a very convenient component of Spark is the Spark SQL API [4], which offers an SQL interface over (semi)structured data. The adoption of a relational-based approach for profile creation enabled us to implement the Builder logics using nothing else than SQL expressions (derived from (1) to (8)), which allowed us to obtain, on the one hand, a robust, compliant with best practices, maintainable and efficient application (highly optimized code is generated by the Catalyst optimizer [4]) and, on the other hand, not to have to deal with headaches typical of distributed data processing systems such as code generation/optimization, data locality aware task distribution, skewness, and resilience management.

5 Evaluation

This section presents the experiments to evaluate ABSTAT-HD performance. Firstly, the experimental setup constituted by the datasets, the environment for the experiments and the workload are introduced. Secondly, we present the performance analysis considering different configurations of the experimental setup and, finally, we discuss the results for each configuration along with a detailed report on the potential errors detected in the Microsoft Academic Knowledge Graph.

5.1 Experimental setup

The experimental setup of ABSTAT-HD is in line with the setup used in the only other approach proposed in the state-of-the-art to distribute the computation of knowledge graph profiling, namely, DistLODStat [43], where the scalability of the distributed and centralized version of the same systems are compared.

5.1.1 Datasets

For our experiments we use two very large and famous datasets: DBpedia and Microsoft Academic Knowledge Graph. DBpedia is one of the most important datasets of

[8] https://hadoop.apache.org/
[9] https://spark.apache.org/

Table 1 Evaluation datasets

Dataset	Size (GB)	Size (triples)	#Types/entity
dbp-2015_{47M}	62	46.963.783	1250
dbp-2014_{566M}	123	566.056.062	2619
$makg_{2.11B}$	253	2.110.667.729	1000
$makg_{2.39B}$	340	2.395.989.657	1639
dbp-$2016_{2.75B}$	538	2.749.621.319	1235
$makg_{6.36B}$	882	6.367.278.909	1219
$makg_{7.74B}$	1.031	7.747.475.306	1219
$makg_{8.19B}$	1.183	8.194.742.011	1219

the Semantic Web community as it contains real, large-scale data and is complex enough with 760 types and 2865 properties. It has a documented schema which might be downloaded easily.[10] For DBpedia we consider three versions with different size: dbp-2014_{566M}[11] (full dataset), dbp-2015_{47M}[12] (the following chunks: types, mapping-based literals, objects and properties about person data only), and dbp-$2016_{2.75B}$[13] (all available chunks except for those with the label *sorted).

The second dataset is Microsoft Academic Knowledge Graph[14] (makg in the following). We considered such dataset as it a very large KG, containing information about scientific publications and related entities, such as authors, institutions, journals, and fields of study. It contains 8 types and 57 properties, thus its schema is not as complex as DBpedia. In order to have datasets with different size but same complexity, we created the following samples: $makg_{2.11B}$ (including the following chunks: Authors and Paper Authors Affiliations), $makg_{2.39B}$ (including only Papers), $makg_{6.36B}$ (all chunks except of Abstracts, URLs and Paper References) and $makg_{7.74B}$ (all chunks except of Abstracts and URLs). Finally $makg_{8.19B}$ represents the full dataset.

The list of used datasets and their respective statistics about size in terms of GB and number of triples and number of types/entities is shown in Table 1.

5.1.2 Experimental setting

The experiments reported in this paper have been performed by deploying the ABSTAT-HD Builder component on a Microsoft Azure Virtual Machine (VM) cluster. In particular, the cluster consists of Standard_D13_v2 VMs featuring 8 virtual CPUs and 56GiB of RAM; one VM (replicated for availability) acts as a manager node while the number of worker nodes is varied from 1 to 5 (with autoscale disabled). The cluster runs the Azure HDInsight[15] platform, based on Apache Hadoop 3.1 and Apache Spark 2.4. Regarding the data store, the cluster uses Azure Blob Store, which also implements the HDFS API. The resource manager is Apache YARN, which has been configured to run a single queue of jobs to execute; in this way, a job can use all cluster resources. Every other configuration has been left with default values. In addition a VM with Standard_D13_v2 configuration and 1TB HDD was used for comparisons with the original ABSTAT builder.

The campaign of experiments aimed to prove ABSTAT-HD scalability has been carried out considering all datasets, the two workload (core-profiling and full-profiling) and varying the worker nodes in the set {1, 2, 4, 5}. Each experiment, identified by the triple (# nodes, core/full-profiling and dataset) has been repeated three times and the average time calculated.

Besides proving the scalability of the new Builder, we also show the importance of the pattern minimalization on type and properties. To achieve this, we compare a full-profiling process and a full-profiling process with no minimalization in terms of execution time and the number of patterns generated. Furthermore, a short experiment has been carried out on the original ABSTAT Builder implementation to assess the scalability.

5.1.3 Workload

Our experiments aim to evaluate the ABSTAT-HD performances in three main orthogonal dimensions: (i) *size*, *complexity*, and *profiling types*.

The *dataset size* is considered a critical property in determining the performance of any profiling tool. Typically, large datasets need more time to be processed and small datasets may need less time. Despite the continuous debates and efforts, there is still no agreed definition of what constitutes a small dataset. In this paper, we do not give a definition about the dataset size (i.e., we do not categorize datasets as small, medium, and large) but consider dataset with increasing number of triples. The smallest dataset with respect to the number of triples is a subset of DBpedia 2015 having only $\sim 47M$ triples while the biggest dataset is the full version of makg having $\sim 8.2B$ triples (Table 1).

The second dimension considered in our experiments is the complexity of the dataset. For the purpose of the summarization approach, the complexity of a dataset is measured in terms of different features that affect different phases of the generation of the profile. For this dimension we consider: (i) the number of distinct entities which affects the cardinality descriptors computation, (ii) the number of types per entity

[10] https://wiki.dbpedia.org/services-resources/ontology

[11] Available at http://downloads.dbpedia.org/2014/en/.

[12] Downloaded from http://downloads.dbpedia.org/2015-10/core-i18n/en/.

[13] Available at http://downloads.dbpedia.org/2016-10/core-i18n/en/.

[14] http://ma-graph.org/rdf-dumps/

[15] https://docs.microsoft.com/en-us/azure/hdinsight/

Table 2 Evaluation ontologies

Ontology	Props.	Types	Subclass	Subprop.
dbp-2014	2795	683	745	965
dbp-2015-10	2833	739	739	941
dbp-2016-10	2865	760	760	948
ma-graph	57	8	0	0

which affects the number of the minimal pattern base, and (iii) the ontology features like the number of types (subclass) and properties (subproperties) relations which affect the minimal types calculation and inference calculation. Table 2 outlines the above dimension for all datasets.

Notice that the number of types per entity for each dataset is very useful to evaluate the performance of ABSTAT-HD with respect to the complexity of the dataset that is being profiled. Observe that versions of DBpedia are more complex than makg ones. The terminology graph of DBpedia has more types, properties and the average length of its taxonomy branches is greater than makg terminology graph. In fact, makg ontology has no subtype and subproperty relations. Moreover, the overall number of types and properties is almost 40 times smaller for makg. Therefore, all above reasons make DBpedia datasets more complex. Finally, the third dimension considers the processing load which is addressed by including different set of profiling features to compute. The whole profiling pipeline is presented in Fig. 2. We consider two profiling configurations: (i) *core-profiling* which includes features such as minimalization on types and frequency statistics for types, predicates and patterns and (ii) *full-profiling* (the whole set of features). In this way, we created two processing loads that require different efforts.

5.2 Performance analysis

In this section, we analyze how the scalability of ABSTAT-HD is affected by the three considered dimensions (size, complexity, and profiling type).

Figures 9 and 10 show how the size of the dataset affects the scalability. In both figures each curve corresponds to a different sample of the two selected KGs. The figures show how ABSTAT-HD performance changes as a function of the number of worker nodes. It is clear that the time required to complete the computation halves if the number of worker nodes doubles. In Fig. 9 the dbp-2014 requires $\sim$ 963 min to be profiled using one worker node, $\sim$ 47 min on 2 worker nodes, $\sim$ 253 min on 4 worker nodes and, finally, $\sim$ 20 min on 5 worker nodes. This behavior is maintained regardless of the type of profiling selected and the complexity of the dataset. The same trend is shown in both core-profiling (Fig. 10) and full-profiling (Fig. 11) plots for both DBpedia

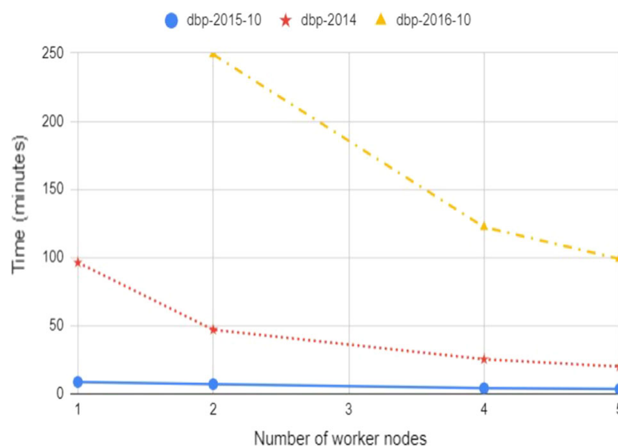

Fig. 9 Scalability on worker nodes for DBPedia (core-profiling)

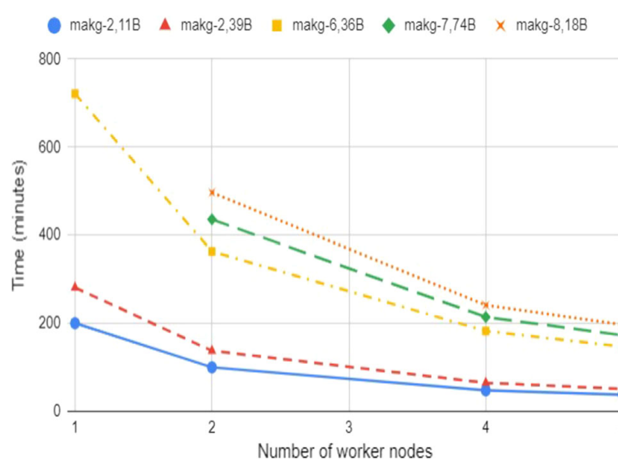

Fig. 10 Scalability on worker nodes for MAKG (core-profiling)

and makg. These experimental results confirm the complexity analysis shown in Sect. 3.3.

The comparison of the performance of the same sample of makg considering the two different types of profiling is shown in Fig. 12; it is evident that the full-profiling requires more time than the core one. This behavior can be explained by considering that in the full-profiling a bigger set of statistics is calculated for each minimal pattern according to Queries 5 and 8.

To assess the scalability of ABSTAT-HD in function of the dataset complexity, we consider dbp-$2016_{2.75B}$ and $makg_{2.39B}$. These two samples have approximately the same number of triples but have very different complexity (Table 2). Figure 13 reports the scalability of two datasets in function of the number of worker nodes. Notice that the complexity of dataset effects the slope of the curve: the more complex the dataset the worse the performance, but in any case curve still shows a good scalability. As a last remark, for big and complex dataset such as DBpedia it is impossible to compute any type of profile in a non-distributed way. Our

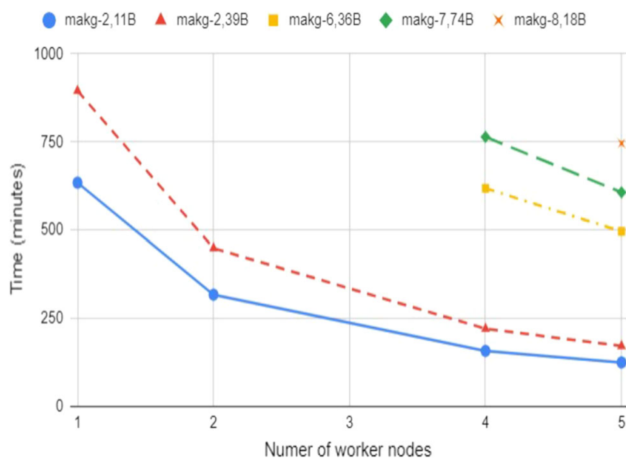

Fig. 11 Scalability on worker nodes for MAKG (full-profiling)

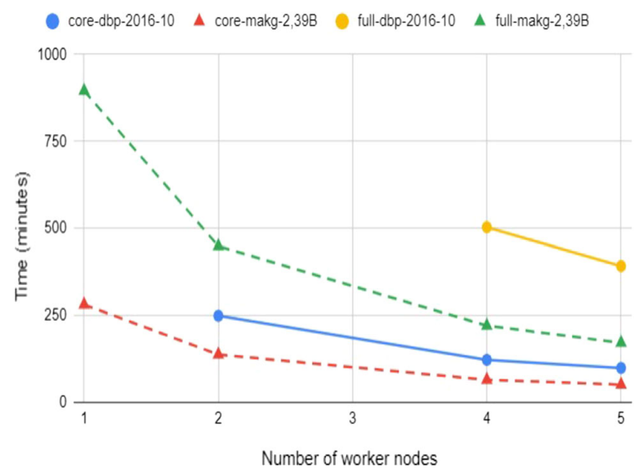

Fig. 13 Complexity of datasets and scalability

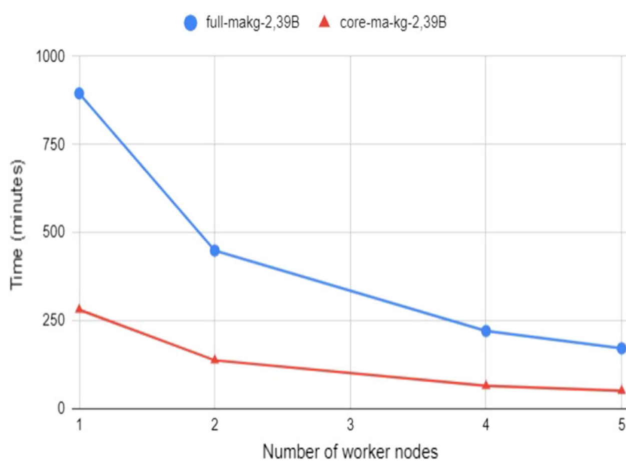

Fig. 12 Difference of profiling types

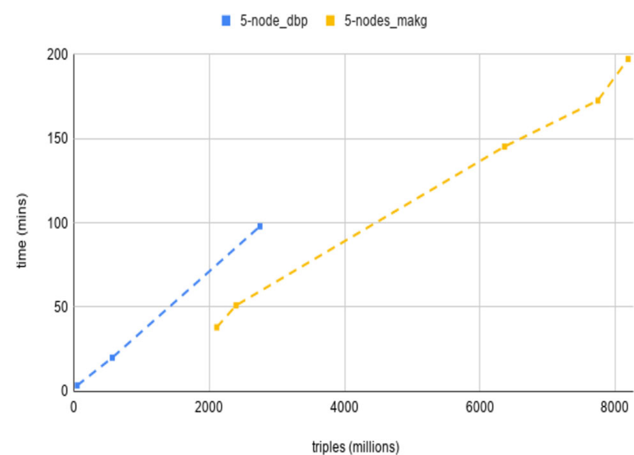

Fig. 14 Size scalability for DBPedia and makg (core-profiling)

experiments show that to profile such type of datasets at least 4 worker nodes are needed.

Figure 14 shows the performance of ABSTAT-HD with respect to the size of the datasets on core-profiling computed on 5 worker nodes. The time has a linear behavior with respect to the size dimension. However, despite the complexity of DBpedia (the blue curve) the time needed is similarly linear to the makg. This means that the complexity relevance declines while working on many nodes as already defined in Sect. 3.3

To conclude, the analysis about the scalability shows that ABSTAT-HD scales well considering all three dimensions (size, complexity, and type of summarization).

Another important question to answer for a better comprehension of ABSTAT-HD is to identify what are, among the three dimensions, the ones that are more time consuming in the summarization process. By analyzing results reported in Figs. 9 and 10 it is possible to confirm that:

1. Concerning the size of the dataset, given a fixed number of worker nodes (e.g., 4 or 5) there is a linear correlation between the number of triples to analyze and the requested time to process them.
2. The complexity of the dataset seems to not impact the performance. In fact, by considering the sample ma-graph$_{2.39B}$ and dbp-2016-10 (see Table 2), despite the fact that the later has an ontology that is 40 times bigger, the time needed for the full-profiling with 5 worker nodes is only about two times greater than the corresponding time of ma-graph$_{2.39B}$.
3. Independently from the size and complexity dimensions, the full-profiling of a given dataset requires up to 3 times more time than the core-profiling. This is due to the fact that, for the full-profiling a greater set of statistics is computed (Queries 5 and 8).

5.3 ABSTAT-HD versus related work

This section analyzes the impact of minimalization on execution time and number of generated patterns. To this end, we compare ABSTAT-HD and ABSTAT over selected tasks and datasets.

The main hypothesis behind the introduction of minimalization is that, when the input KG includes several assertions inferred from deep hierarchies of types and properties defined in the ontology, minimalization reduces significantly the number of patterns. We test this hypothesis by executing full-profiling with and without type and property minimalization on the dbp-2014$_{566M}$ KG; in the first case, the number of generated patterns is 1.636.629 and the time required for the process to complete is 71.3 min while in the second scenario we get 2.919.869 patterns calculated in 94.4 min. Thus, it appears evident that minimalization has a significant impact, halving the number of generated patterns and speeding up the workflow by a 31%.

Table 3 reports the results of another experiment in which we have run ABSTAT-HD and ABSTAT in a single node cluster. Results shows that in the same conditions, ABSTAT-HD presents a significantly better performance than ABSTAT.

The average ratio between time elapsed for full-profiling and core-profiling in ABSTAT-HD is roughly 3×, while for ABSTAT is about 42×. Furthermore, we can see that for core-profiling ABSTAT-HD can be up to ∼ 9× faster and for full-profiling can be up to ∼ 35× faster than ABSTAT.

We have further compared ABSTAT-HD with state-of-the-art approaches for which similar settings were used in the original papers. In fact, the experimental settings of ABSTAT-HD are in line with those used in the only other approaches, namely, DistLODStat [43]. Even though DistLODStat and ABSTAT-HD have different scopes (the final output does not provide the same information) and cannot be directly compared, we provide a comparison in terms of size of the datasets that are processed in the experiments. The largest dataset handled by ABSTAT-HD (makg—1.183 GB) is much larger than the datasets considered by the approaches proposed in the literature. In particular, according to what has been reported in [43], it is approximately 6 times larger than the more significant dataset processed by DistLODStat (i.e., 200 GB). Other profiling approaches, such as [20], experimented with real and synthetic graphs of up to 36.5 GB (approx 32 times smaller than makg), while [17] is evaluated on 6 datasets where the biggest one has the size of 56 MB (approx 21.125 times smaller than makg).

5.4 Discussion on the results

Despite the lack of some data points for the heaviest computations is still clearly visible in Figs. 10, 11 and 14 that the trend of the performance is a linear function of the dataset

Table 3 Performance comparison between ABSTAT and ABSTAT-HD (1 node). Time is expressed in minutes

Dataset	dbp-2015$_{47M}$	dbp-2014$_{566M}$	makg$_{2.11B}$
ABSTAT$_{\text{(core)}}$	20.5	713,6	1735
ABSTAT-HD$_{\text{(core)}}$	8,6	96,3	2003
ABSTAT$_{\text{(full)}}$	834	> 4320	–
ABSTAT-HD$_{\text{(full)}}$	24	384	634

number of triples (dataset dimension). Nonetheless, there are some interesting highlights. First, the rapidly increasing slope curve once over 8 billions of triples in core-profiling (Fig. 10) and full-profiling (Fig. 11) indicates that we have reached the limit of the cluster capabilities for any number of nodes. In particular, during the summarization, ABSTAT-HD performs large joins. Especially when such joins are performed on tens of billions of triples, Spark workers write intermediate data on the disk as it shuffles. In case the disk space is not enough it throws an error. This is reflected on dbp-2016$_{2.75B}$ dataset in Fig. 9 on which it was not possible to compute core-profiling with only one worker node. Instead, for the makg$_{6.36B}$ dataset even though the number of triples doubles but its terminology graph is more simple, this is possible. Also in this case, the join cost makes the difference: in DBpedia dataset, joins are more expensive since an ontology with many types, in general, leads to more minimal types of entities. The effect of the dataset complexity can be furthermore noticed by the slope gap trend in Fig. 13. DBpedia takes more time to be processed and this is more evident as the processing load dimension increases. In fact, for all datasets, the slope gap in pairs of curves is higher on full-profiling. In conclusion, when regarding performance, dataset complexity is not a concern if the disk space is big enough to support a large number of joins.

We can obtain useful observations by using the results plots in a more practical way. Let's consider two use cases:

(i) A user has already deployed ABSTAT-HD in a n-nodes cluster and wants to know its performance if the input dataset increases. Figures 10 and 11 show how the time needed to profile datasets changes with respect to the dataset size for a given configuration and processing load.

(ii) A user has already deployed ABSTAT-HD in n-nodes clusters and wants to profile datasets that have similar size. She/he wants to know how the computation performance changes with respect to the number of nodes. Figures 10 and 11 show that despite the size of the dataset, the cluster would perform t for one node $(n = 1)$, while t/n for n nodes. Therefore, $t_{(s,n)} = (t_{(s,1)}/n)$ where s is the dataset size and n is the number of nodes.

Concerning the impact of minimalization, experiments on dbp-2014$_{566M}$ demonstrate that including the minimal-

ization within the overall profiling process leads to a better MPB compression and a reduction in computation time. Minimalization is very effective in pruning the pattern space when the terminology graph $G^{\mathcal{T}}$ is rich in types, properties, and subclass/subproperty relations, and when entities have multiple types, many of which are redundant. Furthermore, reducing the number of minimal patterns for which frequency and cardinality descriptors are computed reduces also the execution time and memory usage for computing these statistics. Queries (2) and (4) show the relation tables for frequency and cardinality calculation where table dimensions depend on the number of patterns, types, and types per entity, thus in cases where a cluster has reached its maximum in memory capacity by executing full-profiling, minimalization can undoubtedly help to reduce the number of patterns and make the whole computation more suitable. Therefore, we can conclude that minimalization reaches the maximum effect on KGs that include the transitive closure of types on typing assertions and the transitive closure on properties on relational assertions (thus having multiple redundant relational assertions), and use rich ontologies.

As reported in Table 3, the large gap in execution time between core-profiling and full-profiling for ABSTAT is caused by intense I/O, sorting, and bucketing operations for instance count and cardinality calculation (which are present only in the full-profiling workload). It is also evident that ABSTAT-HD is much faster than ABSTAT in both workloads, arguably due to the in-memory distributed computation and query optimization offered by the Spark Framework.

5.5 Potential errors detected in the MAKG

This section summarizes some of the potential errors detected in the makg exploring the profile produced by ABSTAT. As from Table 1, such KG has 57 properties and 8 types defined within the ontology of makg.[16] Moreover, it uses 5 external types from the *fabio* ontology[17] (Book, BookChapter, ConferencePaper, JournalArticle, and PatentDocument) and 25 external properties (from ontologies fabio, purl,[18] cito,[19] dbpedia, etc.). The Microsoft Academic Knowledge Graph maintainers have published also the schema[20] as an easier way to visualize relations among types and datatypes. From this schema, a user can easily notice that the KG makes use of two `owl:equivalentClass`: one between `makg:FieldOfStudy` and `fabio:SubjectDiscipline` and the other between `makg:Paper` and `fabio:Work`. However, both relations are present only in the schema depicted in their website, but they are both missing in the owl ontology. All the external types used in the dataset from *fabio* ontology (Book, BookChapter, ConferencePaper, JournalArticle, and PatentDocument) are subtypes of the class `fabio:Expressions`. Intuitively, such types refer to subtypes of Paper, that in the fabio ontology are under `fabio:Expressions` not under `fabio:Work`. Thus, we can deduce that there is a wrong equivalent relation between `makg:Paper` and `fabio:Work`. Instead, the equivalent relation should be between `makg:Paper` and `fabio:Expressions`.

A second problem that clearly emerged thanks to the patterns produced by ABSTAT is related to the domain and range restrictions. The predicate `makg:citationCount` has as defined domain in the ontology the type Author while as range an integer. However, its usage in the dataset does not respect such definition. Indeed, there are 12 patterns in the data that have `makg:citationCount` as predicate. Such patterns take in the subject position types such as `makg:Author`, `makg:Affiliation`, `makg:ConferenceInstance`, `makg:ConferenceSeries`, `makg:FieldOfStudy`, `makg:Journal`, `makg:Paper`, `makg:Book`, `makg:BookChapter`, `fabio:ConferencePaper`, `fabio:JournalArticle` and `fabio:PatentDocument`. Even though, such predicate should be used in the data only with the type Author as stated in the ontology, it is used also with other types that are not in a subtype relation with the type Author, e.g., Affiliation. There is no subtype relationship between affiliation, author, conference instances, conference series, field of study, journal and paper. So here, we can deduce that either a concept that is superconcept of all the above ones is missing or the domain for this property should be better defined in the ontology. The similar potential error is also identified for the predicates `makg:bookTitle`, `makg:paperCount` and `makg:rank`.

The third problem is related to the cardinality values for some predicates. With ABSTAT, we were able to identify several patterns for which the cardinality values seem to identify possible errors in the data. For instance, the pattern `makg:Paper purl:creator makg:Author` occurs 549, 142, 397 times in the data and has the maximum subject–objects cardinality equal to 6760. This means that at least one paper has as creators 6760 different authors. This number exceeds the usual number of authors per paper, thus it might indicate a potential error in the data. Similarly, the pattern `makg:Paper cito:cites makg:Paper` has as maximum subject-objects cardinality equal to 27, 036. This means that a given paper cites up to 27, 036 different papers. Although we can not say that this is an error, in practice, papers cite up to 100 other papers. Moreover, as we can see

[16] http://ma-graph.org/ontology.owl

[17] https://sparontologies.github.io/fabio/current/fabio.html

[18] http://purl.org/dc/terms/

[19] http://purl.org/spar/cito

[20] http://ma-graph.org/schema-linked-dataset-descriptions/

from the statistics that ABSTAT produces, the average number of cited paper for such pattern is equal to 20, thus having a cardinality with value greater than 27 thousand may indicate an error in the data. Other patterns that might indicate quality errors in the data are: papers that have 21 different languages (`makg:Paper purl:language xmls:language`), journal papers that have 17 different disciplines (`makg:JournalPaper fabio:hasDiscipline makg:FieldOfStudy`), journal papers that have up to 329 different URLs (`makg:JournalPaper purl:hasURL owl:Thing`), 5 different affiliations have the same homepage (of type `owl:Thing`), etc.

6 Related work

This section gives an overview of state-of-the-art approaches that summarize Knowledge Graphs (Sect. 6.1) and approaches that have adapted distributed technologies to improve the scalability of processing graphs (Sect. 6.2).

6.1 Knowledge graph profiling

RDF graph profiling has been intensively studied, and various approaches and techniques have been proposed to provide a concise and meaningful representation of the RDF KGs. There are different recent surveys that discuss some of the approaches to profile knowledge graphs such as [9,44,54]. Most of the work on KGs profiling has been done in the field of KG summarization, which has been extensively surveyed in [9]. However, the related work discussed in this section are different as we focus not only on the summarization approaches but also on profiling ones.

Loupe [28] is the approach most similar to ABSTAT. It extracts patterns that describe relations among types, along with a rich set of statistics about their use within the dataset. The triple inspection functionality provides information about triple patterns (of the form $< subjectType, property, objectType >$) that appear in the dataset and their frequency. Loupe extracts also other information such as the namespace used in the dataset. Differently from ABSTAT, Loupe does not adopt a minimalization technique, thus, Loupe's profiles contain many more patterns and consequently they are not as compact as ABSTAT profiles.

A data graph summary that assists users in formulating queries across multiple data sources by considering vocabulary usage is proposed in [8]. This approach extracts clusters called "node collections" to group a set of similar concepts and properties. The final aim of the paper is to help users into efficiently formulating complex SPARQL queries. For this reason a component called Assisted SPARQL Editor is developed. Similarly, ABSTAT patterns also help users formulate SPARQL queries as they encode useful information to understand the structure of the data [46]. Differently, ABSTAT does not group nodes with similar characteristics and does not have an interface to help users formulate SPARQL queries (for this task, users can use the endpoint of the dataset itself, e.g., http://dbpedia.org/sparql).

In [13] structural summaries are constructed by using bisimilarity to group nodes of a dataset as the notion of equivalence with the aim to provide users a summary-based exploration. Such is the backbone of S+EPPS where summaries are constructed of blocks and each block represents a non-overlapping subset of the original dataset. Blocks are connected by edges that summarize the relationships between dataset nodes across blocks (e.g., :person, :city, :location, etc.). ABSTAT does not use bisimilarity and does not extract summaries blocks but instead uses minimal type patterns to construct its summaries.

Structural equivalence is considered in [41] to provide users a summary that has a reduced size with respect to the KGs itself. This approach summarizes structural similar subgraphs by considering them to be bisimilar if they cannot be distinguished by their outgoing paths. Additionally, ASGG, proposed in [52], uses structural similarity for summarizing knowledge graphs. ASSG' summary is constructed by considering equivalence classes by bisimulation relations and it has the adaptive ability to adjust its structure according to different query graphs. Similarly to the above approaches, ABSTAT profiles are compact with respect to the size of the KGs but differently, it does not consider the structural similarity of graphs.

The semi-structured data summarization approach proposed in [10] is query-oriented and it has a very high computational complexity. The summary enables static analysis and helps formulate and optimize queries. The scope is to reflect whether the query has some answers on this graph, or to find a simpler way to formulate the query. Similar to ABSTAT, information that can be easily inferred is excluded from the summary.

Other approaches consider pattern mining to summarize KGs [3,10,38,44]. Summarizing entities considering their neighborhood similarity up to a distance d is the aim of [44]. Users might specify a bound k as the maximum number of the desired patterns to be included in the summary. The k d-summaries/patterns are chosen to satisfy and maximize *informativeness* (the total amount of information; entities and their relationships in a kg) and *diversity* (cover diverse concepts with informative summaries).

A weighted summary composed of supernodes connected by superedges as a result of the partitioning of the original set of vertices in the graph is proposed in [38]. Edge densities between vertices in the corresponding supernodes are considered as weights. A reconstruction error is proposed to introduce the error for the dissimilarity between the graph

and the summary. ABSTAT approach does not consider edge densities.

RDF graphs might be more comprehensible by reducing their size as proposed by [3]. Size reduction is a result of bisimulation and agglomerative clustering (one of the most common types of hierarchical clustering) which discovers subgraphs that are similar with respect to their structure. ABSTAT does not use clustering but instead reduces the number of patterns to be explored by adopting a minimalization technique.

There is a bunch of work that summaries KGs quantitatively to represent the content of the RDF graph such as [5,7,16,18,19,26].

SPADE allows exploring summaries through the prism of interesting aggregate statistics [16]. It uses OLAP-style aggregation to provide users with meaningful content of an RDF graph. Users may refine a given aggregate, by selecting and exploring its subclasses. The aggregation is centered around a set of facts, which are nodes of the RDF graph. LODSight [18] is a web-based tool that displays a summary based on type-property and datatype-property paths. The tool visualizes classes, datatypes and predicates used in the dataset with the aim to help users to quickly and easily find out what kind of data the dataset contains. It also shows how vocabularies are used in the dataset. This tool is similar to ABSTAT but it does not extract minimal types and is not maintained any more.

LODOP is a framework for executing, optimizing, and benchmarking profiling tasks in Linked Data [19]. These tasks include the calculation of: number of triples, average number of triples per resources/ per object URI, number of properties, average number of property values, inverse properties, etc.

Thirty-two different statistical criteria for RDF datasets can be obtained by LODStats profiling tool [5]. These statistics describe the dataset and its schema and include statistics about the number of triples, triples with blank nodes, labeled subjects, number of owl:sameAs links, class and property usage, class hierarchy depth, cardinality descriptors, etc. These statistics are then represented using Vocabulary of Interlinked Datasets (VOID) and Data Cube Vocabulary.[21]

Several algorithms to compute different profiling, mining, or cleansing tasks [1] are implemented in a web browser tool called ProLOD++. The profiling task includes the calculation of: frequencies and distribution of distinct subjects, predicates and objects, range of predicates, etc. ProLOD++ can also identify predicates combinations that contain only unique values as key candidates to identify entities distinctly.

RDFStats generates statistics for datasets behind SPARQL endpoint [26]. These statistics include the number of anonymous subjects and different types of histograms; URIHistogram for URI subject, and histograms for each property and the associated range(s). It also provides the total number of instances for a given class or a set of classes and methods to obtain the URIs.

Differently from the above approaches, ABSTAT does not use aggregation methods for different summary resolution. Instead, it uses a terminology graph to extract only those patterns that describe relationships between instances of the most specific types.

6.2 Scalable graph processing

Graph processing approaches can be divided into two major categories: (1) centralized (storing the KG as a single node) and (2) distributed (distributing the KG among multiple cluster nodes). In this section, we focus only on the second category. Scalable graph processing has been reviewed recently by [2,32,33,50]. Most of the approaches might be categorized by their main purpose such as; *data storage, indexing, query languages* and *query execution*. These purposes are orthogonal, thus, a work may be classified in multiple categories.

SANSA is a graph processing tool that has adopted distributed technologies to enhance scalability [25]. It provides a unified framework for several applications such as link prediction, knowledge base completion, querying, and reasoning. It is built upon general-purpose processing engines such as Apache Spark and Apache Flink. Similar to ABSTAT, the architecture of SANSA is also modular where each component has its own functionality. Among the main functionalities of SANSA are: read and write native RDF or OWL data from HDFS; supports different RDF and OWL serializations; provides different partitioning strategies (semantic-based, vertical, and graph-based partitioning); it computes several RDF statistics (such as the number of triples, RDF terms, properties per entity, and usage of vocabularies across datasets), and apply quality assessment in a distributed manner.

Entity Aware Graph compression technique (EAGRE) [53] proposes a new representation of RDF data on Cloud platforms with the aim to efficiently evaluate SPARQL with sequence modifiers such as projection, order by, etc., as quickly as possible. Such approach stores RDF data in HDFS in a (key, value) form. Entity graph is partitioned among worker machines using an indexing structure that adopts a space-filling curve technique used to index high dimensional data. The minimization of the input and output costs for SPARQL query processing is achieved by efficiently distributing schedules. The scope of this approach is to reduce the reading of data blocks which should be read for query evaluation.

Trinity.RDF [51] is a distributed in-memory RDF system that stores RDF data in its native graph form (i.e., represent-

[21] http://www.w3.org/TR/vocab-data-cube/

ing entities as graph nodes, and relationships as graph edges). Each entity is stored giving a unique id as a key and as a value an adjacency list with incoming and outgoing edges. Such values contain predicate and node id of the connected nodes. Representing the graph in this way leads to optimization for SPARQL query processing, but also supports more advanced graph analytics on RDF data. Trinity.RDF uses efficient in-memory graph exploration instead of join operations for SPARQL processing. A SPARQL query is decomposed into a set of triple patterns, where for each pattern firstly matches are found, and then starting from these matches graph is explored. The exploration-based approach allows to perform exploration in parallel, thus saving time.

Similarly to [51] also Triple Asynchronous and Distributed (TriAD) [23] uses graph-exploration strategies based on Message Passing. It adds a multi-threading layer for the paths of a query plan that allows the execution in parallel. TriAD produces a summary graph using bisimulation (where only the predicates of the query triple patterns are labeled with constants) and locality-based summaries (where nodes that share some neighbors are spread across the partitions). This has the aim to index compact synopses of the data graph. A SPARQL query usually involves finding and connecting different parts of a graph, thus such approach works as it prunes. Since SPARQL typically involves finding connected components of the data graph, locality-based approaches are particularly effective in pruning part of triple patterns are labeled with constants.

SparkRDF is an RDF graph processing engine that implements SPARQL query on Spark that has the aim to reduce the high I/O and communication cost [11]. The graph is divided into multi-layer elastic subgraphs based on classes and relations. Spark APIs are employed and an iterative join operations with distributed memory, to minimize the cost of intermediate results to perform subgraph matching by triple patterns.

SemStore uses a Rooted Sub-Graph as the partition unit to partition and store the data with the aim to efficiently localize the four common types of SPARQL queries (SELECT, ASK, DESCRIBE, and CONSTRUCT) [49]. A k-mean partition algorithm is used to avoid redundancy and localize better the query types to a cluster nodes. The architecture of SemStore is master-slave where queries are submitted to the master while the slaves contain local indexes and statistics that will be used during join processing.

S2RDF partition RDF data by using ExtVP (Extended Vertical Partitioning) that uses a semi-join-based preprocessing, similar to the Join Indices in relational databases, to efficiently minimize the query input size regardless of its triple patterns [42]. Such partitioning considers the position of a joint variable that occurs in both triple patterns to determine the columns on which tables must be joined. In terms of updates, insertions and deletions, the first two are performed quickly by appending new triples to ExtVP tables while deletion are a bit more complicated and not so quick.

PRoST [14] (Partitioned RDF on Spark Tables) is a system that stores RDF data in a graph form using hash partitioning. It combines the Vertical Partitioning (VP) approach with the Property Table (PT), to translate SPARQL queries into Spark execution plans. The Vertical Partitioning is used to create a table for each distinct predicate of the input graph, containing all tuples (subject, object) that are connected by that predicate. The Property Table consists of a unique table where each row contains a distinct subject and all object values for that subject, stored in columns identified by the property to which they belong. For the query optimization, it uses Join Trees guided by simple statistics to translate SPARQL queries. The triple patterns that have the same subject are grouped together as a node and a special label is assigned to it (using Property Table), while all other groups with a single triple pattern are translated to nodes (using Vertical Partitioning).

Leon [22] is a distributed RDF system, which mitigates the multi-query problem. It uses a partitioning scheme based on characteristic sets that aims to capture the structure of the dataset and detects common sub-structure efficiently and effectively in a batch of SPARQL queries. The initial cost of such partitioning is very low. RDF strings are encoded into numerical IDs and a bi-directional dictionary is built which stores the ids of characteristic set and subjects. This dictionary is used afterward as an index for optimizing queries.

7 Conclusions

Processing and profiling big knowledge graphs can be a complex and challenging task but it is becoming increasingly important when KGs are used for machine learning activities. In this paper, we present ABSTAT-HD a minimalization-based profiling tool able to provide a profile for very large knowledge graphs. The modular architecture of ABSTAT allows to benefit from the advantages of distributed computing. Given the limitations on the previous version, ABSTAT-HD scales horizontally by adopting technologies such as Apache Hadoop and Spark that allow the distribution of the processing load of large datasets across clusters of computers using simple programming models. Thanks to the ability to detect and handle failures at the application layer, Apache Hadoop delivers a highly available service on top of a cluster of computers. Moreover, Apache Spark is a distributed computing framework that, unlike the default compute engine Apache MapReduce, runs in memory.

To evaluate the scalability performance of ABSTAT-HD we profile several datasets that have different complexity such as DBpedia and Microsoft Academic Knowledge Graph. Three orthogonal dimensions were considered during profiling process: the size of the dataset, its complexity

with respect to the number of types and predicates and ontological features, and the profiling type which considers the overall workload of the profiling process. Experiments show that given a fixed number of worker nodes, there is a linear correlation between the size of the dataset (in terms of number of triples) and the time needed to profile it. Moreover, when regarding the complexity, experiments show that it does not impact the performance. ABSTAT-HD is able to compute the profile for the core-profiling or full-profiling even for complex datasets. ABSTAT-HD is able to process very large KGs such as DBpedia and MAKG for both the core and full-profiling. Clearly, the performance on full-profiling is lower with respect to the core-profiling, as for the latter a greater set of statistics is computed. Finally, despite the size and the complexity of the dataset, full-profiling needs up to 3 times the time for the core-profiling.

Moreover, we have shown that minimalization has an impact also in pruning the pattern space and the execution time. In fact, minimalization halves the number of generated patterns (dbp-2014$_{566M}$) and speeds up the workflow execution by a 31%. Furthermore, we proved that for core-profiling ABSTAT-HD can be up to $\sim 9\times$ faster and for full-profiling can be up to $\sim 35\times$ faster than the previous ABSTAT implementation.

Future works include the enrichment of ABSTAT profiles with other statistics about the data. Moreover, we plan to represent profiles based on exiting vocabulary in order to increase the automatic analysis of profile in the exploratory data analysis phase of any machine learning task based on a KG. Furthermore, we want to use ABSTAT-HD to profile a set of KGs in specific field such as biology, geography and so on, with the aim to offer to the community complete, precise and ready to use data based on FAIR principles.

Funding Open access funding provided by Università degli Studi di Milano - Bicocca within the CRUI-CARE Agreement.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

8 Appendix

8.1 Proof of Theorem 1

Before proving the equivalence about minimal pattern bases (MPBs) stated in Theorem 1 in the two directions (respectively, $\rightarrow$ and $\leftarrow$), we prove a lemma that will be used in the proof. In the following we use *instance* to refer to either an individual or a literal.

Lemma 1 *If a pattern* $\pi = (C, P, D)$ *is in the minimal pattern base* $\Pi^{\mathcal{A},\mathcal{T}}$ *for a set of assertions* $\mathcal{A}$ *under a terminology graph* $G^{\mathcal{T}}$*, there must exist a pair of instances a and b such that* $P(a, b) \in \mathcal{A}$ *is non-redundant and* π *minimally represents* $P(a, b)$.

The condition for a pattern to be part of the minimal pattern base for $\mathcal{A}$ is that the pattern minimally represents some relational assertion of $\mathcal{A}$. Relational assertions are either redundant or not redundant. If a pattern minimally represents a redundant relational assertion ϕ, there must be some assertion ψ, from which ϕ can be inferred, such that it is part of the $G^{\mathcal{T}}$-inference base of ϕ and not redundant. Therefore, there must exist some pair of instances a and b such that $\phi = P(a, b)$ to comply with the condition for which a pattern $\pi = (C, P, D)$ represents a non-redundant relational assertion iff there exists a set $\{C(a), D(b), P(a, b)\} \subseteq \mathcal{A}$.

Theorem 1 ($\rightarrow$ direction) *if a pattern* π *is in the MPB* $\Pi^{\mathcal{A},\mathcal{T}}$ *for a set of assertions* $\mathcal{A}$ *under a terminology graph* $G^{\mathcal{T}}$*, then it is also in the set* $\Pi^{\mathcal{A}^-}$ *of patterns that represent all the relational assertions in* $\mathcal{A}^-$.

Proof If a pattern $\pi = (C, P, D)$ is in an MPB $\Pi^{\mathcal{A},\mathcal{T}}$, then it must also represent some non-redundant relational assertion $P(a, b) \in \mathcal{A}$ for Lemma 1 and $\{C(a), P(a, b), D(b)\} \subseteq \mathcal{A}$. We show that $\{C(a), P(a, b), D(b)\} \subseteq \mathcal{A}^-$ also holds. Since π in minimal then it does not exist a pattern $\pi' \prec^{G^{\mathcal{T}}} \pi$ such that π' represents $P(a, b)$. Since $P \preceq^{G^{\mathcal{T}}} P$, the latter implies that there does not exist a type C' such that $C'(a) \in \mathcal{A}$, and $C' \prec^{G^{\mathcal{T}}} C$ and a type D' such that $D'(a) \in \mathcal{A}$ and $D' \prec^{G^{\mathcal{T}}} D$. $P(a, b) \in \mathcal{A}^-$ because it is not redundant by hypothesis. $C(a) \in \mathcal{A}^-$ because otherwise it would have been removed because redundant in favor of some assertion $C'(a) \in \mathcal{A}^-$ with $C' \preceq^{G^{\mathcal{T}}} C$, which would make (C', P, D) a strict subpattern of (C, P, D) thus contradicting the hypothesis that (C, P, D) is a minimal pattern for $P(a, b)$ under $G^{\mathcal{T}}$. The same argument clearly applies also to $D(a)$. □

Theorem 1 ($\leftarrow$ direction) *if a pattern* π *is in the set* $\Pi^{\mathcal{A}^-}$ *of patterns, which represent every relational assertions in* $\mathcal{A}^-$*, then it is also in the minimal pattern base* $\Pi^{\mathcal{A},\mathcal{T}}$ *for the set of assertions* $\mathcal{A}$ *under a terminology graph* $G^{\mathcal{T}}$.

Proof If $\pi \in \Pi^{\mathcal{A}^-}$, then, for $\pi = (C, P, D)$, $\{C(a), P(a,b), D(b)\} \subseteq \mathcal{A}^- \subseteq \mathcal{A}$. Because of how $\mathcal{A}^-$ is defined, we know that all these assertions are not redundant, which implies that there cannot be any other pattern $\pi' \prec_{G^\mathcal{T}} \pi$ and π is therefore minimal and thus in the minimal pattern base. □

8.2 Profile updating

Although optimizing and testing incremental updates is out of the scope of this paper, we discuss, in this section, how profiles could be updated upon changes in a KG without recomputing them from scratch. We consider updates to the set of *assertions* $\mathcal{A}$ of a KG, which is expected to change more frequently than its *terminology*. Updates may affect two different kinds of assertions in the KG, relational and typing assertions, and consist of two basic operations: addition and delete of an assertion ϕ (a change in one triple can be modeled as a sequence of removal and addition operations), yielding to four cases.
A relational assertion ϕ *is deleted*. In this case, the profile is updated as follows: **(1)** if ϕ is redundant (i.e., if the $G^\mathcal{T}$-inference base of ϕ is not empty), stop here; otherwise, **(2)** calculate $\Pi^{\phi,\mathcal{T}}$, i.e., the minimal pattern base for ϕ (which is equivalent to applying the profiling algorithm to a relational assertion set $\{\phi\}$), and **(3)** update statistics for all $\pi \in \Pi^{\phi,\mathcal{T}}$ as follows:

(a) decrease the frequency by one;
(b) calculate the inferred patterns, and, for each inferred pattern decrease its instances by one;
(c) recompute cardinality descriptors.

(4) check whether removing ϕ makes some relational assertion ψ in $\mathcal{A}$ not redundant anymore; if not, stop here; otherwise, **(5)** calculate the minimal patterns for each no longer redundant assertion $\psi \in \mathcal{A}$, and **(6)** update the statistics as follows:

(a) increase the frequency by one;
(b) calculate the inferred patterns, and increase the number of instances by one;
(c) recompute cardinality descriptors.

A relational assertion ϕ *is added*. In this case, if $\phi = P(a,b)$, the profile is updated with the following procedure: **(1)** if ϕ is redundant, stop here; otherwise, **(2)** check if ϕ makes some assertion in $\mathcal{A}$ redundant, delete them from $\mathcal{A}$, and update the profiles following the procedure for deleted relational assertions; **(3)** compute $\Pi^{\phi,\mathcal{T}}$, i.e., the minimal patterns for ϕ, and add them to the profiles; **(4)** update the statistics, which is performed similarly as described for the above deletion case, but increasing instead of decreasing the values.

A typing assertion ϕ *is deleted*. In this case, if $\phi = C(a)$, we proceed as follows: **(1)** if ϕ is redundant, stop here; otherwise, **(2)** let be facts(a) the set of relational assertions in $\mathcal{A}$ having a in subject/object position, calculate $\Pi^{\text{facts}(a),\mathcal{T}}$ (the set of minimal patterns for $facts(a)$) and two more sets: $mp_+(a)$ and $mp_-(a)$, which are, respectively, the subset of novel patterns for facts(a) and the set of patterns that are no longer minimal patterns for facts(a) as a consequence of removing ϕ, **(3)** update statistics as follows:

(a) for all $\pi \in mp_+(a)$, increase frequency by one and for all $\pi \in mp_-(a)$ decrease the frequency by one
(b) for all π inferred from $mp_-(a)$ excluding those that can be also inferred from $\Pi^{\text{facts}(a),\mathcal{T}}$, decrease the number of instances by one.
(c) for all $\pi \in mp_+(a) \cup mp_-(a)$ revise cardinality descriptors

A typing assertion ϕ *is added*. In this case, if $\phi = C(a)$, we proceed as follows: **(1)** if ϕ is redundant, stop here; **(2)** check if ϕ make some triple in $\mathcal{A}$ redundant and eventually delete them from $\mathcal{A}$ and update the profiles following the procedure for deleted typing assertions; **(3)** compute the set $mp_+(a)$ of novel minimal patterns for facts(a) as a consequence of adding ϕ, **(4)** update statistics as follows:

(a) for all $\pi \in mp_+(a)$ increase frequency by one
(b) for all π inferred from $mp_+(a)$ increase the number of patterns by one
(c) for all $\pi \in mp_+(a)$ revise cardinality descriptors

We make a few observations about the cost of key operations that are part of the update procedure. The update of pattern count statistics (e.g., steps 3.a and 3.b in the delete of a relational assertion) operates only over the (smaller set of) patterns, while the update of cardinality descriptors (e.g., 3.c) in the delete of a relational assertion) must retrieve every relational assertion minimally represented by $\Pi^{\phi,\mathcal{T}}$. An open issue is how to optimize the efficiency of the latter operation, whether by persisting the table that is used in memory to associate patterns (adding I/O disk overhead) and relational assertions, or by submitting queries on-the-fly. Empirically answering to this question is out of the scope of this paper and is left for future work. A second observation concerns the search for potentially redundant relational and typing assertions in several steps of the update procedure. Although frequent, these searches are local in the sense that inference-related dependencies between relational assertions can occur only between assertions having the same subject and object (e.g., $P(a,b)$ and $Q(a,b)$). Similarly, dependencies between typing assertions can be found only between assertions having the same subject (e.g., $C(a)$ and $D(a)$).

References

1. Abedjan, Z., Grütze, T., Jentzsch, A., Naumann, F.: Profiling and mining RDF data with prolod++. In: 2014 IEEE 30th International Conference on Data Engineering, pp. 1198–1201. IEEE (2014)
2. Ali, W., Saleem, M., Yao, B., Hogan, A., Ngomo, A.-C.N.: Storage, indexing, query processing, and benchmarking in centralized and distributed RDF engines: a survey. arXiv:2009.10331 (2020)
3. Alzogbi, A., Lausen, G.: Similar structures inside RDF-graphs. LDOW **996** (2013)
4. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: relational data processing in spark. In: SIGMOD 15, pp. 1383–1394. Association for Computing Machinery (2015)
5. Auer, S., Demter, J., Martin, M., Lehmann, J.: LODSTATS–an extensible framework for high-performance dataset analytics. In: International Conference on Knowledge Engineering and Knowledge Management, pp. 353–362. Springer (2012)
6. Baldacci, L., Golfarelli, M.: A cost model for spark SQL. IEEE Trans. Knowl. Data Eng. **31**(5), 819–832 (2019)
7. Böhm, C., Naumann, F., Abedjan, Z., Fenz, D., Grütze, T., Hefenbrock, D., Pohl, M., Sonnabend, D.: Profiling linked open data with prolod. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), pp. 175–178. IEEE (2010)
8. Campinas, S., Perry, T.E., Ceccarelli, D., Delbru, R., Tummarello, G.: Introducing RDF graph summary with application to assisted SPARQL formulation. In: 2012 23rd International Workshop on Database and Expert Systems Applications (DEXA), pp. 261–266. IEEE (2012)
9. Čebirić, Š, Goasdoué, F., Kondylakis, H., Kotzinos, D., Manolescu, I., Troullinou, G.: Summarizing semantic graphs: a survey. VLDB J. **28**(3), 295–327 (2019)
10. Čebirić, Š, Goasdoué, F., Manolescu, I.: Query-oriented summarization of RDF graphs. Proc. VLDB Endow. **8**(12), 2012–2015 (2015)
11. Chen, X., Chen, H., Zhang, N., Zhang, S.: SPARKRDF: elastic discreted RDF graph processing engine with distributed memory. In: 2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT), vol. 1, pp. 292–300. IEEE (2015)
12. Christmann, P., Roy, R.S., Abujabal, A., Singh, J., Weikum, G.: Look before you hop: Conversational question answering over knowledge graphs using judicious context expansion. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 19, pp. 729–738. Association for Computing Machinery, New York (2019)
13. Consens, M.P., Fionda, V., Khatchadourian, S., Pirro, G.: S+ epps: construct and explore bisimulation summaries, plus optimize navigational queries; all on existing SPARQL systems. Proc. VLDB Endow. **8**(12), 2028–2031 (2015)
14. Cossu, M., Färber, M., Lausen, G.: Prost: distributed execution of SPARQL queries using mixed partitioning strategies. arXiv:1802.05898 (2018)
15. di Noia, T., Maurino, A., Magarelli, C., Palmonari, M., Rula, A.: Using ontology-based data summarization to develop semantics-aware recommender systems. In: The Semantic Web—ESWC 2018 Satellite Events, Heraklion, Crete, Greece, June 3–7, 2018 (2018)
16. Diao, Y., Guzewicz, P., Manolescu, I., Mazuran, M.: Spade: a modular framework for analytical exploration of RDF graphs (2019)
17. Diao, Y., Guzewicz, P., Manolescu, I., Mazuran, M.: Efficient exploration of interesting aggregates in RDF graphs. arXiv:2103.17178 (2021)
18. Dudáš, M., Svátek, V., Mynarz, J.: Dataset summary visualization with lodsight. In: European Semantic Web Conference, pp. 36–40. Springer (2015)
19. Forchhammer, B., Jentzsch, A., Naumann, F.: LODOP-multi-query optimization for linked data profiling queries. In: PROFILES@ ESWC (2014)
20. Goasdoué, F., Guzewicz, P., Manolescu, I.: RDF graph summarization for first-sight structure discovery. VLDB J. **29**(5), 1191–1218 (2020)
21. Guo, Q., Zhuang, F., Qin, C., Zhu, H., Xie, X., Xiong, H., He, Q.: A survey on knowledge graph-based recommender systems. IEEE Trans. Knowl. Data Eng. p. 1 (2020)
22. Guo, X., Gao, H., Zou, Z.: Leon: A distributed RDF engine for multi-query processing. In: International Conference on Database Systems for Advanced Applications, pp. 742–759. Springer (2019)
23. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: Triad: a distributed shared-nothing RDF engine based on asynchronous message passing. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 289–300 (2014)
24. Hogan, A., Blomqvist, E., Cochez, M., dAmato, C., de Melo, G., Gutierrez, C., Gayo, J.E.L., Kirrane, S., Neumaier, S., Polleres, A., Navigli, R., Ngomo, A.-C.N., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., Zimmermann, A.: Knowledge graphs (2020)
25. Jabeen, H., Graux, D., Sejdiu, G.: Scalable knowledge graph processing using SANSA. In: Knowledge Graphs and Big Data Processing, pp. 105–121. Springer (2020)
26. Langegger, A., Woss, W.: RDFSTATS-an extensible RDF statistics generator and library. In: 2009 20th International Workshop on Database and Expert Systems Application, pp. 79–83. IEEE (2009)
27. Lian, X., Zhang, T.: The optimization of cost-model for join operator on spark SQL platform. MATEC Web Conf. **173**, 01015 (2018)
28. Mihindukulasooriya, N., Poveda-Villalón, M., García-Castro, R., Gómez-Pérez, A.: Loupe-an online tool for inspecting datasets in the linked data cloud. In: International Semantic Web Conference (Posters and Demos) (2015)
29. Mohamed, S.K., Novácek, V., Nounu, A.: Discovering protein drug targets using knowledge graph embeddings. Bioinformatics **36**(2), 603–610 (2020)
30. Myklebust, E.B., Jiménez-Ruiz, E., Chen, J., Wolf, R., Tollefsen, K.E.: Knowledge graph embedding for ecotoxicological effect prediction. In: The Semantic Web—ISWC, Proceedings, Part II, volume 11779 of Lecture Notes in Computer Science, pp. 490–506. Springer (2019)
31. Noy, N.F., Gao, Y., Jain, A., Narayanan, A., Patterson, A., Taylor, J.: Industry-scale knowledge graphs: lessons and challenges. Commun. ACM **62**(8), 36–43 (2019)
32. Özsu, M.T.: A survey of RDF data management systems. Front. Comp. Sci. **10**(3), 418–432 (2016)
33. Pan, Z., Zhu, T., Liu, H., Ning, H.: A survey of RDF management technologies and benchmark datasets. J. Ambient. Intell. Humaniz. Comput. **9**(5), 1693–1704 (2018)
34. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 165–178 (2009)
35. Principe, R.A.A., Spahiu, B., Palmonari, M., Rula, A., De Paoli, F., Maurino, A.: Abstat 1.0: compute, manage and share semantic profiles of RDF knowledge graphs. In: European Semantic Web Conference, pp. 170–175. Springer (2018)
36. Ragone, A., Tomeo, P., Magarelli, C., Di Noia, T., Palmonari, M., Maurino, A., Di Sciascio, E.: Schema-summarization in linked-data-based feature selection for recommender systems. In: Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3–7, 2017, pp. 330–335 (2017)
37. Reza, T., Halawa, H., Ripeanu, M., Sanders, G., Pearce, R.: Scalable pattern matching in metadata graphs via constraint checking. arXiv:1912.08453 (2019)

38. Riondato, M., García-Soriano, D., Francesco, B.: Graph summarization with quality guarantees. Data Min. Knowl. Disc. **31**(2), 314–349 (2017)
39. Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., Özsu, M.T.: The ubiquity of large graphs and surprising challenges of graph processing: extended survey. VLDB J. 1–24 (2019)
40. Schaible, J., Gottron, T., Scherp, A.: Termpicker: enabling the reuse of vocabulary terms by exploiting data from the linked open data cloud. In: International Semantic Web Conference, pp. 101–117. Springer (2016)
41. Schätzle, A., Neu, A., Lausen, G., Przyjaciel-Zablocki, M.: Large-scale bisimulation of RDF graphs. In: Proceedings of the Fifth Workshop on Semantic Web Information Management, p. 1. ACM (2013)
42. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2rdf: RDF querying with SPARQL on spark. Proc. VLDB Endow. **9**(10) (2016)
43. Sejdiu, G., Ermilov, I., Lehmann, J., Mami M.N.: DISTLODSTATS: distributed computation of RDF dataset statistics. In: International Semantic Web Conference, pp. 206–222. Springer (2018)
44. Song, Q., Yinghui, W., Lin, P., Dong, L.X., Sun, H.: Mining summaries for knowledge graph search. IEEE Trans. Knowl. Data Eng. **30**(10), 1887–1900 (2018)
45. Spahiu, B., Maurino, A., Palmonari, M.: Towards improving the quality of knowledge graphs with data-driven ontology patterns and SHACL. In: ISWC Best Workshop Papers, pp. 103–117 (2018)
46. Spahiu, B., Porrini, R., Palmonari, M., Rula, A., Maurino, A.: ABSTAT: ontology-driven linked data summaries with pattern minimalization. In: European Semantic Web Conference, pp. 381–395. Springer (2016)
47. Staab, S., Studer, R.: Handbook on Ontologies. Springer Science and Business Media, Singapore (2010)
48. Trotter, W.T.: Partially ordered sets. Handb. Comb. **1**, 433–480 (1995)
49. Wu, B., Zhou, Y., Yuan, P., Jin, H., Liu, L.: SEMSTORE: A semantic-preserving distributed RDF triple store. In: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, pp. 509–518 (2014)
50. Wylot, M., Hauswirth, M., Cudré-Mauroux, P., Sakr, S.: RDF data storage and query processing schemes: a survey. ACM Comput. Surv. (CSUR) **51**(4), 1–36 (2018)
51. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale RDF data. Proc. VLDB Endow. **6**(4), 265–276 (2013)
52. Zhang, H., Duan, Y., Yuan, X., Zhang, Y.: ASSG: Adaptive structural summary for RDF graph data. In: International Semantic Web Conference (Posters and Demos), pp 233–236. Citeseer (2014)
53. Zhang, X., Chen, L., Tong, Y., Wang, M.: EAGRE: towards scalable i/o efficient SPARQL query evaluation on the cloud. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 565–576. IEEE (2013)
54. Zneika, M., Vodislav, D., Kotzinos, D.: Quality metrics for RDF graph summarization. Semantic Web (Preprint):1–30 (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.