Swift Logic for Big Data and Knowledge Graphs

Swift Logic for Big Data and Knowledge Graphs

Luigi Bellomarini[1], Georg Gottlob[1,2], Andreas Pieris[3], and Emanuel Sallinger[1]

[1]Department of Computer Science, University of Oxford, UK

[2]Institute of Information Systems, TU Wien, Austria

[3]School of Informatics, University of Edinburgh, UK

Abstract

Many modern companies wish to maintain knowledge in the form of a corporate knowledge graph and to use and manage this knowledge via a knowledge graph management system (KGMS). We formulate various requirements for a fully-fledged KGMS. In particular, such a system must be capable of performing complex reasoning tasks but, at the same time, achieve efficient and scalable reasoning over Big Data with an acceptable computational complexity. Moreover, a KGMS needs interfaces to corporate databases, the web, and machine-learning and analytics packages. We present KRR formalisms and a system achieving these goals.

1 Introduction

The so-called *knowledge economy*, characteristic for the current Information Age, is rapidly gaining ground. According to Amidon et al. [3], as cited in Wikipedia [37], "The knowledge economy is the use of knowledge [...] to generate tangible and intangible values. Technology, and, in particular, knowledge technology, help to transform a part of human knowledge to machines. This knowledge can be used by decision support systems in various fields and generate economic value." The importance of knowledge as an essential economic driving force has been evident to most corporate decision makers since the late 1970s, and the idea of storing knowledge and processing it to derive valuable new knowledge existed in the context of *expert systems*. Alas, it seems that the technology of those 'early' times was not sufficiently mature: the available hardware was too slow and main memory too tight for more complex reasoning tasks; database management systems were too slow and too rigid; there was no web where an expert system could acquire data; machine learning, and, in particular, neural networks were ridiculed as largely unsuccessful; ontological reasoning was in its infancy and the available formalisms were much too complex for Big Data applications. Meanwhile, there has been huge technological progress, and also much research progress that has led to a better understanding of many aspects of knowledge processing and reasoning with large amounts of data. Hardware has evolved, database technology has significantly improved, there is a (semantic) web with linked open data, companies can participate in social networks, machine learning has made a dramatic breakthrough, and there is a better understanding of scalable reasoning mechanisms. Because of this, and of some eye-opening showcase projects such as IBM Watson [24], thousands of large and medium-sized companies suddenly wish to manage their own *knowledge graphs*, and are looking for adequate *knowledge graph management systems (KGMS)*.

The term *knowledge graph* originally only referred to Google's Knowledge Graph, namely, "a knowledge base used by Google to enhance its search engine's search results with semantic-search information gathered from a wide variety of sources" [38]. Meanwhile, further Internet giants (e.g. Facebook) as well as some other very large companies have constructed their own knowledge graphs, and many more companies would like to maintain a private corporate knowledge graph incorporating large amounts of data in form of facts, both from corporate and public sources, as well as rule-based knowledge. Such a corporate knowledge graph is expected to contain relevant business knowledge, for example, knowledge about customers, products, prices, and competitors rather than mainly world knowledge from Wikipedia and similar sources. It should be managed by a KGMS, i.e., a knowledge base management system (KBMS), which performs complex rule-based reasoning tasks over very large amounts of data and, in addition, provides methods and tools for data analytics and machine learning, whence the equation:

$$\boxed{\text{KGMS} \;=\; \text{KBMS + Big Data + Analytics}}$$

The word 'graph' in this context is often misunderstood to the extent that some IT managers think that acquiring a graph database system and feeding it with data is sufficient to achieve a corporate knowledge graph. Others erroneously think that knowledge graphs necessarily use RDF triple stores instead of plain relational data. Yet others think that knowledge graphs are limited to storing and analyzing social network data only. While knowledge graphs should indeed be able to manipulate graph data and reason over RDF and social networks, they should not be restricted to this. For example, restricting a knowledge graph to contain RDF data only would exclude the direct inclusion of standard relational data and the direct interaction with corporate databases.

Not much has been described in the literature about the architecture of a KGMS and the functions it should ideally

fulfil. In Section 2 we briefly list what we believe are the main requirements for a fully fledged KGMS. As indicated in Figure 1, which depicts our reference architecture, the central component of a KGMS is its core reasoning engine, which has access to a rule repository. Grouped around it are various modules that provide relevant data access and analytics functionalities (see Section 2 for details). We expect a KGMS to fulfil many of these functions.

The reasoning core of a KGMS needs to provide a language for knowledge representation and reasoning (KRR). The data format for factual data should, as said, match the standard relational formalism so as to smoothly integrate corporate databases and data warehouses, and at the same time be suited for RDF and graph data. The rule language and reasoning mechanism should achieve a careful balance between expressive power and complexity. In Section 3 we present VADALOG, a Datalog-based language that matches this requirement. VADALOG belongs to the Datalog$^{\pm}$ family of languages [5; 9; 10; 11; 12; 22] that extend Datalog by existential quantifiers in rule heads, as well as by other features, and restricts at the same time its syntax so as to achieve decidability and tractability. The core of the VADALOG language corresponds to *Warded Datalog*$^{\pm}$ [5; 22], which captures plain Datalog as well as SPARQL queries under the entailment regime for OWL 2 QL [20], and is able to perform ontological reasoning tasks. Reasoning with core VADALOG is computationally efficient and scalable.

After presenting the logical core of VADALOG and its beneficial properties in Section 3.1, we describe in Section 3.2 several features that have been added to it for achieving more powerful reasoning and data manipulation capabilities. To give just one example here, the language is augmented by monotonic aggregations [35], which permits the use of aggregation (via summation, product, max, min, count) even in the presence of recursion. This enables us to swiftly solve problems such as the *company control* problem (studied e.g. in [14]) as explained in the following example, which will serve as a running example throughout the paper.

Example 1.1 (Running Example.) *Assume the ownership relationship among a large number of companies is stored via facts (i.e., tuples of a database relation) of the form* $\mathrm{Own}(comp_1, comp_2, w)$ *meaning that company* $comp_1$ *directly owns a fraction* w *of company* $comp_2$*, with* $0 \leq w \leq 1$*. A company* x *controls a company* y *if* x *directly owns more than half of the shares of* y *or if* x *controls a set* S *of companies that jointly own more than half of* y*. Computing a predicate* $\mathrm{Control}(x, y)$ *expressing that company* x *controls company* y*, is then achieved in* VADALOG *by two rules:*

$$\begin{aligned} \mathrm{Own}(x, y, w), w > 0.5 \quad &\rightarrow \quad \mathrm{Control}(x, y) \\ \mathrm{Control}(x, y), \mathrm{Own}(y, z, w), & \\ v = \mathit{msum}(w, \langle y \rangle), v > 0.5 \quad &\rightarrow \quad \mathrm{Control}(x, z). \end{aligned}$$

Here, for fixed x*, the aggregate construct* $\mathit{msum}(w, \langle y \rangle)$ *forms the sum over all values* w *such that for some company* y*,* $\mathrm{Control}(x, y)$ *is true, and* $\mathrm{Own}(y, z, w)$ *holds, i.e., company* y *directly owns fraction* w *of company* z*.* ■

In Section 4 we introduce the VADALOG KGMS, which builds on the VADALOG language and combines it with existing

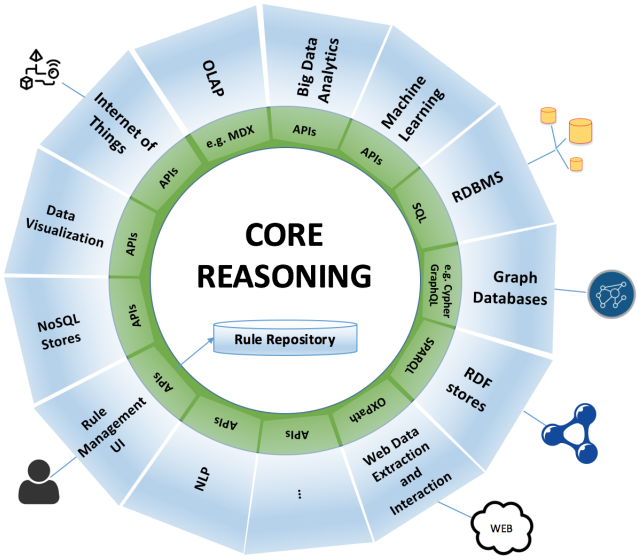

Figure 1: KGMS Reference Architecture.

and novel techniques from database and AI practice such as stream query processing, dynamic in-memory indexing and aggressive recursion control. The VADALOG system is Oxford's contribution to the VADA (*Value Added Data Systems*) research project [1; 18; 26], which is a joint effort of the universities of Edinburgh, Manchester, and Oxford. An outlook on future research and developments is given in Section 5.

2 Desiderata for a KGMS

We proceed to briefly summarize what we think are the most important desiderata for a fully-fledged KGMS. We will list these requirements according to three categories, keeping in mind, however, that these categories are interrelated.

2.1 Language and System for Reasoning

There should be a logical formalism for expressing facts and rules, and a reasoning engine that uses this language, which should provide the following features.

Simple and Modular Syntax: It should be easy to add and delete facts and to add new rules. As in logic programming, facts should conceptually coincide with database tuples.

High Expressive Power: Datalog [14; 25] is a good yardstick for the expressive power of rule languages. Over ordered structures (which we may assume here), Datalog with very mild negation captures PTIME; see, e.g., [16]. A rule language should thus ideally be at least as expressive as plain recursive Datalog, possibly with mild negation.

Numeric Computation and Aggregations: The basic logical formalism and inference engine should be enriched by features for dealing with numeric values, including appropriate aggregate functions.

Probabilistic Reasoning: The language should be suited for incorporating appropriate methods of probabilistic reasoning, and the system should propagate probabilities or certainty values along the reasoning process, that is, compute probabilities or certainty values for derived facts, and make adjustments wherever necessary. Appropriate probabilistic models

may range from simple triangular norm operators (T-norm – cf [23]) over probabilistic database models [36] to Markov Logic networks [32].

Ontological Reasoning: The possibility of ontological reasoning and query answering should be provided. We have two yardsticks here. First, ontological reasoning to the extent of tractable description logics such as DL-Lite$_R$ should be possible. Recall that DL-Lite$_R$ forms the logical underpinning of the OWL 2 QL profile of the Web Ontology Language as standardized by the W3C. Second, the language should also be expressive enough to cover all queries over RDF datasets that are expressible in SPARQL under the entailment regime for OWL 2 QL [20].

Low Complexity: Reasoning should be tractable in data complexity (i.e. when the rules are assumed to be fixed and the fact base is considered the input). Whenever possible, the system should recognize and take profit of rule sets that can be processed within low space complexity classes such as NLOGSPACE (e.g. for SPARQL) or even AC_0 (e.g. for traditional conjunctive database queries).

Rule Repository, Rule Management, and Ontology Editor: A library for storing recurring rules and definitions should be provided, as well as a user interface for comfortable rule management in the spirit of the ontology editor protégé [30].

Dynamic Orchestration: For larger applications, there must be a master module to allow the orchestration of complex data flows. For simple systems, the process must be easily specifiable. For complex systems, the process must be dynamically controllable through intelligent reasoning techniques or external control facilities and tools (e.g. BPM).

2.2 Accessing and Handling Big Data

Big Data Access: The system must be able to provide efficient access to Big Data sources and systems as well as fast reasoning algorithms over Big Data. In particular, the possibility of out-of-memory reasoning must be given in case the relevant data does not fit into main memory. Integration of Big Data processing techniques should be possible where the volume of data makes it necessary (see e.g. [34]).

Database and Data Warehouse Access: Seamless access to relational, graph databases, data warehouses, RDF stores, and major NoSQL stores should be granted. Data in such repositories should be directly usable as factual data for reasoning.

Ontology-based Data Access (OBDA): OBDA [13] allows a system to compile a query that has been formulated on top of an ontology into a query that acts directly on the database. OBDA should be made possible whenever appropriate.

Multi-Query Support: Where possible and appropriate, partial results from repeated (sub-)queries should be evaluated once [33]. The system should be optimized in this regard.

Data Cleaning, Exchange and Integration: Integrating, exchanging and cleaning data should be supported both directly (through an appropriate KRR formalism that is made available through various applications in the knowledge repository), and by allowing integration of third-party software.

Web Data Extraction, Interaction, and IoT: A KGMS should be able to interact with the web by (i) extracting relevant web data (e.g. prices advertised by competitors) and integrating these data into the local fact base, and (ii) exchanging data with web forms and servers that are available through a web interface. One way to achieve this will be discussed in Section 3.2. Similar methods can be used for interacting with the IoT through appropriate network accessible APIs.

2.3 Embedding Procedural and Third-Party Code

Procedural Code: The system should have encapsulation methods for embedding procedural code (proprietary and third party) written in a variety of programming languages and offer a logical interface to it.

Third-Party Packages for Machine Learning, Text Mining, NLP, Data Analytics, and Data Visualization: The system should be equipped with direct access to powerful existing software packages for machine learning, text mining, data analytics, and data visualization. Given that excellent third-party software for these purposes exists, we believe that a KGMS should be able to use a multitude of such packages via appropriate logical interfaces.

3 The VADALOG Language

As said before, VADALOG is a KR language that achieves a careful balance between expressive power and complexity, and it can be used as the reasoning core of a KGMS. In Section 3.1 we introduce the logical core of VADALOG and some interesting fragments of it, while in Section 3.2 we discuss how this language can be extended with additional features that are much needed in real-world applications.

3.1 Core Language

The logical core of VADALOG is a member of the Datalog$^{\pm}$ family of knowledge representation languages, which we call Warded Datalog$^{\pm}$. The main goal of Datalog$^{\pm}$ languages is to extend the well-known language Datalog with useful modeling features such as existential quantifiers in rule heads (the symbol '+'), and at the same time restrict the rule syntax in such a way that the decidability and data tractability of reasoning is guaranteed (the symbol '-'). Before introducing Warded Datalog$^{\pm}$, let us first recall the theoretical foundations underlying Datalog$^{\pm}$ languages.

Foundations of Datalog$^{\pm}$. Let $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$ be disjoint countably infinite sets of *constants*, *(labeled) nulls* and (regular) *variables*, respectively. A *(relational) schema* $\mathbf{S}$ is a finite set of relation symbols (or predicates) with associated arity. A *term* is a either a constant, null or variable. An *atom* over $\mathbf{S}$ is an expression of the form $R(\bar{v})$, where $R \in \mathbf{S}$ is of arity $n > 0$ and $\bar{v}$ is an n-tuple of terms. An *instance* over $\mathbf{S}$ is a (possibly infinite) set of atoms over $\mathbf{S}$ that contain constants and nulls, while a *database* over $\mathbf{S}$ is a finite set of atoms over $\mathbf{S}$ that contain only constants. The *active domain* of an instance I, denoted $dom(I)$, is the set of terms in I.

The core of Datalog$^{\pm}$ languages consists of rules known as *existential rules* or *tuple-generating dependencies*, which essentially generalize Datalog rules with existential quantifiers in rule heads; henceforth, we adopt the term existential rule. Such a rule is a first-order sentence of the form

$$\forall \bar{x} \forall \bar{y} (\varphi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\, \psi(\bar{x}, \bar{z}))$$

where φ (the *body*) and ψ (the *head*) are conjunctions of atoms with constants and variables. For brevity, we write this existential rule as $\varphi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\, \psi(\bar{x}, \bar{z})$ and use comma instead of $\wedge$ for conjoining atoms. The intuitive meaning of such a rule is as follows: if the atoms $\varphi(\bar{t}, \bar{t}')$ occur in an instance I, then there exists a tuple $\bar{t}''$ of constants and nulls such that the atoms $\psi(\bar{t}, \bar{t}'')$ are also in I. Formally, the semantics of a set of existential rules Σ over a database D, denoted $\Sigma(D)$, is defined via the well-known chase procedure. Roughly, the chase adds new atoms to D (possibly involving null values used for satisfying the existentially quantified variables) until the final result $\Sigma(D)$ satisfies all the existential rules of Σ. Notice that, in general, $\Sigma(D)$ is infinite. Now, given a pair $Q = (\Sigma, \mathrm{Ans})$, where Σ is a set of existential rules and Ans an n-ary predicate, the evaluation of Q over a database D, denoted $Q(D)$, is defined as the set of tuples $Q(D) = \{\bar{t} \in dom(D)^n \mid \mathrm{Ans}(\bar{t}) \in \Sigma(D)\}$.

The main reasoning task that we are interested in is *tuple inference*: given a database D, a pair $Q = (\Sigma, \mathrm{Ans})$, and a tuple of constants $\bar{t}$, decide whether $\bar{t} \in Q(D)$. This problem is computationally very hard; in fact, it is undecidable, even when Q is fixed and only D is given as input [9]. This has led to a flurry of activity for identifying restrictions on existential rules that make the above problem decidable. Each such restriction gives rise to a new Datalog$^{\pm}$ language. Notice that as soon as we have an algorithm for solving the tuple inference problem under a certain Datalog$^{\pm}$ language $\mathcal{L}$, then we have an algorithm for computing the certain answers to a conjunctive query over a database and set of rules Σ in $\mathcal{L}$, assuming that adding a conjunctive query to Σ, the obtained set of rules is still in $\mathcal{L}$. This is the case for Warded Datalog$^{\pm}$.

Warded Datalog$^{\pm}$: The Logical Core of VADALOG. The logical core of VADALOG relies on the notion of wardedness, which gives rise to Warded Datalog$^{\pm}$ [22]. In other words, VADALOG is obtained by extending Warded Datalog$^{\pm}$ with additional features of practical utility such as data types, aggregation, etc., which are discussed in the next section. Wardedness applies a restriction on how the "dangerous" variables of a set of existential rules are used. Roughly, a "dangerous" variable is a body-variable that can be unified with a labeled null value during the construction of the chase instance, and it is also propagated to the head of the rule. For example, given the set Σ consisting of the existential rules

$$P(x) \rightarrow \exists z\, R(x, z) \qquad R(x, y) \rightarrow T(y),$$

the variable y in the body of the second rule is "dangerous" (w.r.t. Σ) since starting, e.g., from the database $D = \{P(a)\}$, the chase will apply the first rule and generate $R(a, \nu)$, where ν is a null value that acts as a witness for the existentially quantified variable z, and then the second rule will be applied with the variable y being unified with ν that is propagated to the obtained atom $T(\nu)$. The goal of wardedness is to tame the way null values are propagated during the construction of the chase instance by forcing (1) all the "dangerous" variables to coexist in a single body-atom α, while (2) α can share only "harmless" variables with the rest of the body, where a "harmless" variable is a variable that is unified only with database constants during the construction of the chase. To formalize these two conditions we need to recall some auxiliary notions.

Given a predicate P, a *position* $P[i]$ identifies the i-th attribute of P, and we write α_P for the arity of P. Given a set of existential rules Σ, where $\mathbf{S}$ is the set of predicates occurring in Σ, the set of positions of Σ, denoted $pos(\Sigma)$, is the set $\{P[i] \mid P \in \mathbf{S} \text{ and } i \in \{1, \ldots, \alpha_P\}\}$. The set of *affected positions* of Σ, denoted by $affected(\Sigma)$, which is a subset of $pos(\Sigma)$, is inductively defined as follows: (i) if there exists $\sigma \in \Sigma$ such that at position π an existentially quantified variable occurs, then $\pi \in affected(\Sigma)$; (ii) if there exists $\sigma \in \Sigma$, and a variable v in the body of σ only at positions of $affected(\Sigma)$, and v appears in the head of σ at position π, then $\pi \in affected(\Sigma)$. We denote by $nonaffected(\Sigma)$ the set of positions $(pos(\Sigma) \setminus affected(\Sigma))$, i.e., the set of *non-affected positions* of Σ. The notion of the (non-)affected position allows us to classify the body-variables of an existential rule into harmless, harmful and dangerous. Let Σ be a set of existential rules. Fix a rule $\sigma \in \Sigma$ and a variable v in the body of σ. Then: (i) v is Σ-*harmless* if at least one occurrence of it appears in the body of σ at a position of $nonaffected(\Sigma)$; (ii) v is Σ-*harmful* if it is not Σ-harmless; and (iii) v is Σ-*dangerous* if it is Σ-harmful and appears in the head of σ. Let $harmless(\sigma, \Sigma)$ and $dangerous(\sigma, \Sigma)$ be the set of body-variables of σ that are Σ-harmless and Σ-dangerous, respectively. We are now ready to introduce Warded Datalog$^{\pm}$.

A set of existential rules Σ is called *warded* if, for each $\sigma \in \Sigma$, either $dangerous(\sigma, \Sigma) = \emptyset$, or there exists an atom α in the body of σ, called *ward*, such that:

1. each variable of $dangerous(\sigma, \Sigma)$ occurs in α, and
2. for each variable v in α that occurs in at least one body-atom of σ other than α, $v \in harmless(\sigma, \Sigma)$.

Warded Datalog$^{\pm}$ consists of all the (finite) sets of existential rules that are warded. It is clear that the above two conditions capture the intuition underlying wardedness described above, with the aim of taming the way null values are propagated during the construction of the chase.

At this point, let us clarify that Warded Datalog$^{\pm}$ is a refinement of *Weakly-Frontier-Guarded Datalog$^{\pm}$*, which is defined in the same way but without the condition (2) given above [7]. Weakly-Frontier-Guarded Datalog$^{\pm}$ is highly intractable in data complexity; in fact, it is EXPTIME-complete. This justifies Warded Datalog$^{\pm}$, which is a (nearly) maximal tractable fragment of Weakly-Frontier-Guarded Datalog$^{\pm}$.

Warded Datalog$^{\pm}$ enjoys several favourable properties that make it a robust core towards more practical languages:

- Our main reasoning task under Warded Datalog$^{\pm}$ is data tractable; in fact, it is PTIME-complete when the query and the set of rules are fixed.
- Warded Datalog$^{\pm}$ captures Datalog without increasing the complexity. Indeed, a set Σ of Datalog rules is trivially warded since there are no Σ-dangerous variables.
- Warded Datalog$^{\pm}$ generalizes central ontology languages such as the OWL 2 QL profile of OWL, which in turn relies on the description logic DL-Lite$_R$.
- Warded Datalog$^{\pm}$ is suitable for querying RDF graphs. In particular, by adding stratified and grounded negation to Warded Datalog$^{\pm}$, we obtain a language, called TriQ-Lite 1.0 [22], that can express every SPARQL query under the entailment regime for OWL 2 QL.

Other Swift Logics. Although polynomial time data complexity is desirable for conventional applications, it can be prohibitive for "Big Data" applications; in fact, this is true even for linear time data complexity. This raises the question whether there are fragments of Warded Datalog$^{\pm}$ that guarantee lower data complexity, but at the same time maintain the favourable properties discussed above. Of course, such a fragment should be weaker than full Datalog since Datalog itself is already PTIME-complete in data complexity. On the other hand, such a fragment should be powerful enough to compute the transitive closure of a binary relation, which is a crucial feature for reasoning over graphs, and, in particular, for capturing SPARQL queries under the entailment regime for OWL 2 QL. Therefore, the complexity of such a refined fragment is expected to be NLOGSPACE-complete.

Such a fragment of Warded Datalog$^{\pm}$, dubbed *Strongly-Warded*, can be defined by carefully restricting the way recursion is employed. Before giving the formal definition, let us first recall the standard notion of the predicate graph of a set Σ of existential rules, which essentially encodes how the predicates in Σ interact. The *predicate graph* of Σ, denoted $PG(\Sigma)$, is a directed graph (V, E), where the node set V consists of all the predicates occurring in Σ, and we have an edge from a predicate P to a predicate R iff there exists $\sigma \in \Sigma$ such that P occurs in the body of σ and R occurs in the head of σ. Consider a set of nodes $S \subseteq V$ and a node $R \in V$. We say that R is Σ-*reachable* from S if there exists at least one node $P \in S$ that can reach R via a path in $PG(\Sigma)$. We are now ready to introduce strong-wardedness.

A set of existential rules Σ is called *strongly-warded* if Σ is warded, and, for each $\sigma \in \Sigma$ of the form

$$\varphi(\bar{x}, \bar{y}) \ \rightarrow \ \exists \bar{z}\, P_1(\bar{x}, \bar{z}), \ldots, P_n(\bar{x}, \bar{z}),$$

there exists at most one atom in $\varphi(\bar{x}, \bar{y})$ whose predicate is Σ-reachable from $\{P_1, \ldots, P_n\}$. *Strongly-Warded Datalog*$^{\pm}$ consists of all the (finite) sets of existential rules that are strongly-warded. Intuitively, in a strongly-warded set of existential rules, each rule σ is either non-recursive, or it employs a mild form of recursion in the sense that an atom generated by σ during the construction of the chase instance can affect exactly one body-atom of σ. Let us clarify that the additional syntactic condition posed on warded existential rules in order to obtain strongly-warded existential rules, is the same as the condition underlying *Piecewise Linear Datalog*; see, e.g., [2].

It can be shown that our main reasoning task of tuple inference under Strongly-Warded Datalog$^{\pm}$ is NLOGSPACE-complete in the data complexity.[1] Moreover, this refined language remains powerful enough for capturing OWL 2 QL, and, extended by a mild form of negation, can express every SPARQL query under the entailment regime for OWL 2 QL. As already explained above, the NLOGSPACE data complexity immediately excludes full Datalog. However, Strongly-Warded Datalog$^{\pm}$ includes some important and well-studied fragments of Datalog: (i) *Non-Recursive Datalog*, where the underlying predicate graph is acyclic, and (ii) *IDB-Linear Datalog*, where each rule can have at most one intensional predicate (which appears in the head of at least one rule) in its body, while all the other predicates are extensional.

A lightweight fragment of Strongly-Warded Datalog$^{\pm}$ that is *FO-Rewritable* is *Linear Datalog*$^{\pm}$, where each existential rule can have exactly one body-atom [10]. FO-Rewritability means that, given a pair $Q = (\Sigma, \mathrm{Ans})$, we can construct a (finite) first-order query Q_{FO} such that, for *every* database D, $Q(D)$ coincides with the evaluation of Q_{FO} over D. This immediately implies that tuple inference under Linear Datalog$^{\pm}$ is in AC_0 in data complexity. Despite its simplicity, Linear Datalog$^{\pm}$ is expressive enough for expressing every OWL 2 QL axiom. However, it cannot compute the transitive closure of a binary relation, which is unavoidable if we want to ensure FO-Rewritability. This makes it unsuitable for querying RDF graphs under the entailment regime for OWL 2 QL.

Additional Modeling Features. The Datalog$^{\pm}$ languages discussed above, namely Warded, Strongly-Warded and Linear Datalog$^{\pm}$, can be enriched with useful modeling features without paying a price in complexity. In fact, we can consider *negative constraints* of the form $\forall \bar{x}(\varphi(\bar{x}) \rightarrow \bot)$, where φ is a conjunction of atoms, and $\bot$ denotes the truth constant $false$. We can also consider *equality constraints* (a.k.a. *equality-generating dependencies*) of the form $\forall \bar{x}(\varphi(\bar{x}) \rightarrow x_i = x_j)$, where φ is a conjunction of atoms, and x_i, x_j are variable of $\bar{x}$, as long as they do not interact with the existential rules. This class of equality constraints is known as *non-conflicting*; see, e.g., [12]. Notice that if we consider arbitrary equality constraints, without any restrictions, then our main reasoning task becomes very quickly undecidable [15].

3.2 Extensions

In order to be effective for real-world applications, we extend the core language described above, i.e., Warded Datalog$^{\pm}$, with a set of additional features of practical utility. Although the theoretical properties of the language are no longer guaranteed, our preliminary evaluation has shown that the practical overhead for many of these features remains reasonable in our streaming implementation. In the future, we plan to perform a more thorough complexity analysis and isolate sets of features for which beneficial complexity upper bounds are met and runtime guarantees are given.

Data Types: Variables and constants are typed. The language supports the most common simple data types: integer, float, string, Boolean, date. There is also support for composite data types, such as sets.

Expressions: Variables and constants can be combined into expressions, which are recursively defined as variables, constants or combinations thereof, for which we support many different operations for the various data types: algebraic sum, multiplication, division for integers and floats; containment, addition, deletion of set elements; string operations (contains, starts-with, ends-with, index-of, substring, etc.); Boolean operations (and, or, not, etc.). Expressions can be used in rule bodies (1) as the left-hand side (LHS) of a *condition*, i.e., the comparison (>,<,>=,<=,<>) of a body variable with the expression itself; (2) as the LHS of an *assignment*, i.e.,

[1] More details about the NLOGSPACE upper bound, as well as additional results on Strongly-Warded Datalog$^{\pm}$, which is still under investigation, will be announced soon in a forthcoming paper.

the definition of a specifically calculated value, potentially used as an existentially quantified head variable. In our running example, variable v is calculated with the expression $\texttt{msum}(w, \langle y \rangle)$ and used in the condition $v > 0.5$.

Skolem Functions: Labeled null values can be suitably calculated with functions defined on-the-fly. They are assumed to be deterministic (returning unique labeled nulls for unique input bindings), and to have disjoint ranges.

Monotonic Aggregations: VADALOG supports aggregation (*min*, *max*, *sum*, *prod*, *count*), by means of an extension to the notion of monotonic aggregations [35], which allows adopting aggregation even in the presence of recursion while preserving monotonicity w.r.t. set containment. The company control example shows the use of `msum`, which calculates variable v, as the monotonically increasing sum of the quota w of company z owned by y, in turn controlled by x. The sum is accumulated so that above the threshold 0.5, we have that x controls z. Recent applications of VADALOG in challenging industrial use cases showed that such aggregations are very efficient in a range of real-world Big Data settings.

Data Binding Primitives: Data sources and targets can be declared by adopting *input/output annotations*, a.k.a. *binding patterns*. Annotations are special facts augmenting sets of existential rules with specific behaviours. The unnamed perspective used in VADALOG can be harmonized with the named perspective of many external systems by means of *bind* and *mapping* annotations, which also support *projection*. A special *query bind* annotation also supports binding predicates to queries against inputs/outputs (in the external language, e.g., SQL-queries for a data source or target that supports SQL). In our example, the extension of the Own predicate is our input, which we denote with an `@input("Own")` annotation. The actual facts then may be derived, e.g., from a relational or graph database, which we would respectively access with the two following annotations (the latter one using neo4j's cypher graph query language):

```
@bind("Own", "rdbms", "companies.ownerships").
@qbind("Own", "graphDB",
    "MATCH (a)-[o:Owns]->(b)
     RETURN a,b,o.weight").
```

A similar approach is also used for bridging external machine learning and data extraction platforms into the system. This uses binding patterns as a form of *behaviour injection*: the atoms in rules are decorated with binding annotations, so that a step in the reasoning process triggers the external component. We give a simple example using the OXPath [17] large-scale web data extraction framework – an extension of XPath that interacts with web applications to extract information obtained during web navigation. In our running example, let us assume that our local company ownership information is only partial, while more complete information can be retrieved from the web. In particular, assume that a company register acts as a web search engine, taking as input a company name and returning, as separate pages, the owned companies. This information can be obtained as follows:[2]

```
@qbind("Own", "oxpath",
  "doc('company_register.com/ownerships')
    /descendant::field()[1]/{$1}
      /following::field()[1]/{click}
        /(/descendant
        ::span:<name=(.)>
        ::span:<percentage=(.)>
          ::a:<Link=(@href)>[.#='Next']/
            {click})*").
```

The above examples show a basic bridging between the technologies. Interesting interactions can be seen in more sophisticated scenarios, where the reasoning process and external component processing is more heavily interleaved.

Probabilistic Reasoning: VADALOG offers support for the basic cases in which scalable computation can be guaranteed. Facts are assumed to be probabilistically independent and a minimalistic form of probabilistic inference is offered as a side product of query answering. Facts can be adorned with probability measures according to the well-known possible world semantics [36]. Then, if the set of existential rules respects specific syntactic properties that guarantee probabilistic tractability (namely, a generalization of the notion of *hierarchical queries* [36]), the facts resulting from query answering are enriched with their marginal probability, safely calculated in a scalable way. In the following extension to our running example, we use probabilistic reasoning to account for uncertain ownerships (e.g., due to unreliable sources), prefixing the facts with their likelihood, so as to derive non-trivial conclusions on company control relationships:

$0.8 :: \text{Own}(\text{“ACME”}, \text{“COIN”}, 0.7)$
$0.3 :: \text{Own}(\text{“COIN”}, \text{“SAVERS”}, 0.3)$
$0.4 :: \text{Own}(\text{“ACME”}, \text{“GYM”}, 0.55)$
$0.6 :: \text{Own}(\text{“GYM”}, \text{“SAVERS”}, 0.4).$

In total, the language allows bridging logic-based reasoning and machine learning in three ways. First, the language supports scalable probabilistic inference in basic cases as seen above. Second, the extensions to the core language provide all the necessary features to abstract and embed advanced inference algorithms (e.g. belief propagation) so that they can be executed directly by the VADALOG system, and hence leverage its optimization strategies. Third, for the more sophisticated machine learning applications, data binding primitives allow a simple interaction with specialized libraries and systems as described before.

Post-processing Annotations: Since specific computations are often needed after the result has been produced, VADALOG supports many of them by means of annotations for the following features: *ordering* of the resulting values, as set semantics is assumed on the output, and yet a particular ordering of the facts may be desired by the consumer: for example, `@orderby("Control",1)` sorts the obtained control facts by the controlling company; *deduplication*, in specific conditions (e.g. in presence of calculated values), the output may physically contain undesired duplicates; *non-monotonic aggregations* on the final result, without the limitations induced by recursion; and *certain answers*.

[2] Concretely, the first position of the Own predicate is bound to the `$1` placeholder in the OXPath expression.

4 The VADALOG System

The functional architecture of the VADALOG system, our KGMS, is depicted in Figure 1. The knowledge graph is organized as a repository, a collection of VADALOG rules, in turn packaged in *libraries*. Rules and libraries can be edited and administered through a dedicated management user interface. The external sources are supported by means of a collection of *transducers*, intelligent adapters that allow active interaction with the sources in the reasoning process.

The VADALOG system fulfils the requirements presented in Section 2. The Big Data characteristics of the sources and the complex functional requirements of reasoning are tackled by leveraging the underpinnings of the core language, which are turned into practical execution strategies. By combining these strategies with a highly engineered architecture, the VADALOG system achieves high performance and an efficient memory footprint.

4.1 The Reasoning Process

In this section we give some indications about how our system exploits the key properties of Warded Datalog$^{\pm}$, explained in Section 3.1, in the reasoning process. We focus on the generic reasoning task of computing the certain answers to a conjunctive query over a knowledge graph (i.e., a database and a set of existential rules). Recall that for Warded Datalog$^{\pm}$, the problem of computing the certain answers can be reduced to the problem of tuple inference.

A useful representation of the instance obtained by the chase is the *chase graph* [10], a directed acyclic graph where facts are represented as nodes and the applied rules as edges. It is implicit in the reasoning algorithms devised for Warded Datalog$^{\pm}$ that after a certain number of chase steps (which, in general, depends on the input database), the chase graph exhibits specific periodicities and no new information, relevant to query answering, is generated. Notice, however, that this number of chase steps is a loose upper bound, while in practice, redundancies appear much earlier. The VADALOG system adopts an *aggressive recursion and termination control* strategy, which detects such redundancy as early as possible by combining compile-time and runtime techniques.

At compile time, thanks to wardedness, which limits the interaction between the labeled nulls, the engine rewrites the program in such a way that joins on specific values of labeled nulls will never occur (*harmful join elimination*).

At runtime, the system adopts an *eager optimal pruning* of redundant and potentially non-terminating chase branches, structured in two parts, *detection* and *pruning*. In detection, whenever a rule generates a fact that is isomorphic to a previously generated one, the sequence of applied rules, namely the *provenance*, is stored. In pruning, whenever a fact exhibits the same provenance as another one and they are isomorphic, the fact is not generated and the chase graph is cut off from that node on. Due to wardedness, the provenance information needed is bounded. Moreover, our technique is somehow *lifted*, in the sense that it highly exploits the structural symmetries within the chase graph: for termination purposes, facts are considered equivalent if they have the same provenance and originate from isomorphic facts. This is a great advantage in terms of performance and memory footprint. In particular, many homomorphism checks are avoided.

4.2 The Architecture

In order to have an efficient KGMS that is also effective and competitive in real-world applications, the VADALOG system adopts the described warded-enabled principles and techniques, which guarantee termination and contain redundancy. We adopt a specialized *in-memory* architecture that makes the most of the existing experience in DBMS development.

From a set of VADALOG rules (rewritten at compile time as explained), we generate a *query plan*, i.e., a graph having a node for each rule and an edge whenever the head of a rule appears in the body of another one. Some special nodes are marked as input or output, when corresponding to datasets in external systems or atoms of the reasoning task, respectively. The query plan is optimized with a range of variations on standard techniques, for example, pushing selections and projections as close as possible to the data sources. Finally, the query plan turned into an *access plan*, where generic rule nodes are replaced by the most appropriate implementations for the corresponding low level operators (e.g. selection, projection, join, aggregation, evaluation of expression, etc.). For each operator a set of possible implementations are available and are activated according to common optimization criteria.

The VADALOG system uses a *pull stream-based approach* (or *pipeline approach*), where the facts are actively requested from the output nodes to their predecessors and so on down to the input nodes, which eventually fetch the facts from the data sources. The stream approach is essential to limit the memory consumption or, at least make it predictable, so that the system is effective for large volumes of data.

Our setting is made more challenging by the presence of multiple interacting rules in a single rule set and the wide presence of recursion. We address this by means of a specialized buffer management technique. We adopt *pervasive local caches* in the form of wrappers to the nodes of the access plan, where the facts produced by each node are stored. The local caches work particularly well in combination with the pull stream-based approach, since facts requested by a node successor can be immediately reused by all the other successors, without triggering further backward requests. Also, this combination realizes an extreme form of *multi-query optimization*, where each rule exploits the facts produced by the others, whenever applicable. To limit memory occupation, the local caches are flushed with an *eager eviction strategy* that detects when a fact has been consumed by all the possible requestors and thus drops it from the memory. Cases of actual cache overflow are managed by resorting to standard disk swap heuristics (e.g. LRU, LFU, etc.).

Local caches are also fundamental functional components in the architecture, since they transparently implement the described recursion and termination control. Indeed, the pull stream-based mechanism is completely agnostic to the termination conditions, and simply produces data for the output nodes as long as the input ones provide facts: it is the responsibility of the local caches to detect periodicity and hence to control termination and cut off computation once a known

# companies	all-rand (s)	query-rand (s)	all-real (s)	query-real (s)
10	0.381	0.342	0.2	0.19
100	0.352	0.34	0.21	0.2
1K	0.555	0.491	0.36	0.25
10K	1.319	1.046	0.85	0.47
50K	3.69	2.76	2.36	1.81
100K	7.688	6.834	N/A	N/A
1M	14.39	8.12	N/A	N/A

Figure 2: Reasoning times for the company control scenario.

pattern reoccurs. In this way, we locally inhibit the production of redundant facts.

For the joins, the VADALOG system adopts a cycle-aware extension of the standard *nested loop join*, suitable for the stream-based approach and efficient in combination with the local caches on the operands. However, in order to guarantee good performance, the local caches are enhanced by *dynamic (i.e. runtime) in-memory indexing*; in particular, the caches involved in the joins can be indexed by means of *hash indices* created at runtime so as to activate an even more efficient *hash join* implementation.

4.3 Systems Status and Performance

Our system currently fully implements the core language and is already in use for a number of industrial applications. Many extensions, especially those important for our partners, are already realized, but others are still missing or under development and will be integrated in the future. Our partners show appreciation for the performance of the system, and we are in the process of conducting a full-scale evaluation. However, we want to give a glimpse on the results so far.

In particular, for the company control scenario from our running example, Figure 2 reports promising results. We considered 7 purely randomly generated company ownership graphs (following the Erdős-Rényi model, relatively dense) from 10 to 1M companies and 5 real-world-like graphs (density and topology resembling the real-world setting), from 10 to 50K companies. For each graph we performed two kinds of evaluations: (1) *all-rand* and *all-real*, where we query the reasoner for the control relationship between all the pairs of companies and measure the reasoning time in seconds; (2) *query-rand* and *query-real*, where we query the system for the control relationship between 50 specific pairs of companies and measure the average reasoning time in seconds. Results are extremely promising and suggest that the engine has very good performance for both batch and interactive applications on large knowledge graphs.

4.4 Related Systems

There are a wide variety of existing tools that are related to the VADALOG system. On one side, we have the progeny of data exchange/cleaning/integration/query answering systems [6; 19; 21; 27; 31], whose most recent representatives provide excellent specific chase implementations [8]. However, they are not suitable to address the KGMS requirements due to the lack of emphasis on scalability guarantees, insufficient coverage of important business-desired features and a general tendency towards prototypical architectures, which do not make them ideal for enterprise settings. Similar observations can be made for existing Datalog systems [28].

Another group of related tools is Datalog-based systems. In particular, RDFox [29] and LogicBlox [4] deserve special attention. The former is a high-performance RAM-based Datalog engine, while the latter comes with the philosophy of extending the usual notion of a DBMS to support analytical applications. Both systems are extremely good reasoning engines, demonstrated by high performance in benchmarks [8] and feature coverage, respectively. Although they share with the VADALOG engine the view of adopting novel and enhanced algorithms from the Datalog reasoning experience and database systems design practices, in the VADALOG system we put central emphasis on the adopted logical language. Our system is the first to exploit the theoretical underpinnings of wardedness.

5 Conclusion

In this paper, we have formulated a number of requirements for a KGMS, which led us to postulate our reference architecture (see Figure 1). Based on these requirements, we introduced the VADALOG language whose core corresponds to Warded Datalog$^{\pm}$. The basic VADALOG language is extended by features for numeric computations, monotonic aggregation, probabilistic reasoning, and, moreover, by data binding primitives used for interacting with the corporate and external environment. These binding primitives allow the reasoning engine to access and manipulate external data through the lens of a logical predicate. The external data may stem from a corporate database, may be extracted from web pages, or may be the output of a machine-learning program that has been evaluated over previously computed data relations. We then introduced the VADALOG system, which puts these swift logics into action. This system exploits the theoretical underpinning of Warded Datalog$^{\pm}$ and combines it with existing and novel techniques from database and AI practice.

Many core features of the VADALOG system are already integrated and show good performance. Our plan is to complete the system in the near future. We believe that the VADALOG system is a well-suited platform for applications that integrate machine learning (ML) and data analytics with logical reasoning. We are currently implementing applications of this type and will report about them soon.

Acknowledgement. This work has been supported by the EPSRC Programme Grant EP/M025268/1. The VADALOG system as presented here is the intellectual property of the University of Oxford.

References

[1] VADA Project Website. `http://vada.org.uk/`, 2016. [Online; accessed 19-May-2017].

[2] F. N. Afrati, M. Gergatsoulis, and F. Toni. Linearisability on datalog programs. *Theor. Comput. Sci.*, 308(1-3):199–226, 2003.

[3] D. M. Amidon, P. Formica, and E. Mercier-Laurent. *Knowledge Economics: Emerging Principles, Pactices and Policies*. Tartu University Press Tartu, 2005.

[4] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *SIGMOD*, pages 1371–1382, 2015.

[5] M. Arenas, G. Gottlob, and A. Pieris. Expressive languages for querying the semantic web. In *PODS*, pages 14–26, 2014.

[6] J. Baget, M. Leclère, M. Mugnier, S. Rocher, and C. Sipieter. Graal: A toolkit for query answering with existential rules. In *RuleML*, pages 328–344, 2015.

[7] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. On rules with existential variables: Walking the decidability line. *Artif. Intell.*, 175(9-10):1620–1654, 2011.

[8] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *PODS*, pages 37–52, 2017.

[9] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res.*, 48:115–174, 2013.

[10] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *J. Web Sem.*, 14:57–83, 2012.

[11] A. Calì, G. Gottlob, T. Lukasiewicz, B. Marnette, and A. Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *LICS*, pages 228–242, 2010.

[12] A. Calì, G. Gottlob, and A. Pieris. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.*, 193:87–128, 2012.

[13] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. F. Savo. The mastro system for ontology-based data access. *Semantic Web*, 2(1):43–53, 2011.

[14] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer, 2012.

[15] A. K. Chandra and M. Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM J. Comput.*, 14(3):671–677, 1985.

[16] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[17] T. Furche, G. Gottlob, G. Grasso, C. Schallhart, and A. J. Sellers. Oxpath: A language for scalable data extraction, automation, and crawling on the deep web. *VLDB J.*, 22(1):47–72, 2013.

[18] T. Furche, G. Gottlob, B. Neumayr, and E. Sallinger. Data wrangling for big data: Towards a lingua franca for data wrangling. In *AMW*, 2016.

[19] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's all folks! LLUNATIC goes open source. *PVLDB*, 7(13):1565–1568, 2014.

[20] B. Glimm, C. Ogbuji, S. Hawke, I. Herman, B. Parsia, A. Polleres, and A. Seaborne. SPARQL 1.1 entailment regimes, 2013. W3C Recommendation 21 March 2013.

[21] G. Gottlob, G. Orsi, and A. Pieris. Query rewriting and optimization for ontological databases. *ACM Trans. Database Syst.*, 39(3):25:1–25:46, 2014.

[22] G. Gottlob and A. Pieris. Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue. In *IJCAI*, pages 2999–3007, 2015.

[23] P. Hájek. *Metamathematics of fuzzy logic*. Springer, 1998.

[24] R. High. The era of cognitive systems: An inside look at IBM Watson and how it works. *IBM, Redbooks*, 2012.

[25] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In *SIGMOD*, pages 1213–1216. ACM, 2011.

[26] N. Konstantinou et al. The VADA architecture for cost-effective data wrangling. In *SIGMOD*, pages 1599–1602, 2017.

[27] N. Leone, M. Manna, G. Terracina, and P. Veltri. Efficiently computable Datalog$^{\exists}$ programs. In *KR*, 2012.

[28] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[29] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *AAAI*, pages 129–137, 2014.

[30] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson, and M. A. Musen. Creating semantic web contents with protege-2000. *IEEE IS*, 16(2):60–71, 2001.

[31] R. Pichler and V. Savenkov. Demo: Data exchange modeling tool. *PVLDB*, 2(2):1606–1609, 2009.

[32] M. Richardson and P. M. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

[33] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[34] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD*, pages 1135–1149, 2016.

[35] A. Shkapsky, M. Yang, and C. Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. In *ICDE*, pages 867–878, 2015.

[36] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Morgan & Claypool, 2011.

[37] Wikipedia. Knowledge economy. `https://en.wikipedia.org/wiki/Knowledge_economy`, 2017. [Online; accessed 19-May-2017].

[38] Wikipedia. Knowledge graph. `https://en.wikipedia.org/wiki/Knowledge_graph`, 2017. [Online; accessed 19-May-2017].