Traversing Knowledge Graphs with Good Old (and New) Joins

Paolo Atzeni[1], Luigi Bellomarini[2], Davide Benedetto[1], and Emanuel Sallinger[3]

[1] Università Roma Tre
[2] Banca d'Italia
[3] TU Wien & University of Oxford

1 Introduction

Knowledge Graphs (KGs) provide a concise and intuitive abstraction for a variety of domains where edges capture the (potentially recursive) relationships between the entities [9]. This is leading to the rise of systems and tools able to facilitate graph data modeling, processing and analysis, with prominent AI companies developing core systems based on the *property graph model* [2].

In this context, Datalog-based languages are being re-discovered to be ductile to accomplish *reasoning tasks* over complex property graphs as they provide the essential elements to enable graph navigational operations [3].

The semantics of a Datalog program is usually specificed in an operational way via the CHASE procedure [7]. It entails multiple *nondeterministic choices* such as the rule application order and the fact binding order when multiple unification is possible [6]. In state-of-the-art reasoners, CHASE-based procedures are not directly adopted, but encoded in the form of engineered variations of the *volcano iterator model* [8] and so essentially within a pipe-and-filters architecture, where nodes (filters) are relational algebra operators and edges (pipes) are dependency connections between the rules. Such (potentially cyclic) structures, known as *access plans*, need to be translated into *reasoning plans*, where abstract relational algebra operators are transformed into specific project, select and join implementations: many implementations of each operator exist and it is up to the optimizer to choose the best one in terms of execution cost.

This paper focuses on cases where the Datalog reasoning process involves a graph traversal task and investigates the connection between reasoning plans and graph traversal strategies. We move from the observation that the nondeterministic choices posed by the chase can be leveraged to control graph traversals —allowing to alternate breadth-first and depth-first strategies— and study the link of such choices with the reasoning plans. We conclude that in plans, specific join implementations and rule prioritization policies reflect the nondeterministic choices and exploit them to guide graph traversals in modern reasoners. Specifically, we implement our results in the VADALOG System [3], a state-of-the-art knowledge graph management systems and conduct experimental evaluation.

2 Knowledge Graphs

Let us start from some some preliminary notions. A KG can be defined as *semi-structured data model* characterized by three components: (i) a *ground extensional component* (EDB), i.e., a set of relational constructs for schema and data which can be effectively modeled as a property graph; (ii) an *intensional component* (IDB), i.e., a set of inference rules over the constructs of the ground extensional component; (iii) a *derived extensional component* that can be produced as the result of the application of the inference rules over the ground extensional component (with the so-called *reasoning process*) [4].

Reasoning in logic-based KGs substantiates in the application of rules (representing the IDB) to the EDB, in order to generate the derived extensional component by logical inference. This process is commonly known as *forward chaining* [1], typically applied via CHASE-based procedures [7].

KGs are particularly suited for the representation of domains with many interconnected entities: EDB is typically modeled as a *property graph* (PG), while IDB encode the *traversal logic*. We adopt a relational representation of EDBs and thus of PGs where nodes and edges are encoded as facts over relation symbols that are specific to the domain of interest.

3 Traversing Knowledge Graphs

To uncover the relationship between reasoning plans and graph traversals in Datalog, we can start with a basic *st-connectivity* scenario [11].[4] For nodes s and t of a directed graph, st-connectivity is the decision problem of establishing whether t is reachable from s. Let us consider an example.

Example 1. The following set Σ of Datalog rules reasons on st-connectivity between Frankfurt and Zurich. EDB D contains facts of the form Linked(x, y), expressing that a city y is directly reachable from x. The intensional predicate Connected denotes connectivity.

$$\mathit{Connected}(\mathrm{x}, \mathrm{y}) :- \mathit{Linked}(\mathrm{x}, \mathrm{y}) \quad (1)$$

$$\mathit{Connected}(\mathrm{x}, \mathrm{z}) :- \mathit{Connected}(\mathrm{x}, \mathrm{y}), \mathit{Linked}(\mathrm{y}, \mathrm{z}) \quad (2)$$

$$\top :- \mathit{Connected}(\mathrm{source}, \mathrm{target}), \mathrm{source} = \mathrm{Frankfurt}, \mathrm{target} = \mathrm{Zurich} \quad (3)$$

Let us analyze how Datalog rules are applied, by considering the CHASE procedure. The CHASE adds new facts to the source database D until the final result $\Sigma(D)$ satisfies all the rules of Σ. Initially $\Sigma(D) = D$. A *unifier* is a mapping from variables to constants. We say that a rule $\rho = \varphi(\bar{x}, \bar{y}) \rightarrow \psi(\bar{x})$ is *applicable*

[4] While specialized algorithms exist for st-connectivity [11], here we do not aim at providing new heuristics for the problem, but at showing how Datalog evaluation strategies materialize into different traversal algorithms.

to $\Sigma(D)$ if there is a unifier θ_ρ such that $\varphi(\bar{x}\theta_\rho, \bar{y}\theta_\rho) \subseteq \Sigma(D)$ and and $\psi(\bar{x}\theta_\rho)$ does not belong to $\Sigma(D)$. If ρ is applicable to $\Sigma(D)$ with a unifier θ_ρ, then it performs a *chase step*, i.e., it *generates* new facts $\psi(\bar{x}\theta'_\rho)$ that are added to $\Sigma(D)$, where $\bar{x}\theta_\rho = \bar{x}\theta'_\rho$. The chase performs chase steps until no rule in Σ is applicable.

The chase poses two classes of nondeterministic choices: (i) for an applicable rule, multiple possible unifiers can exist and, (ii) multiple rules can be applicable at the same time. By *handle* we mean a mechanism by which a specific nondeterministic choice in the chase can be leveraged to control the resulting graph traversal behaviour. We recognize two of them:

- *unification anatomy*, i.e., controlling the application order of logical unifiers;
- *unification morphology*, i.e., controlling the application order of rules.

The unification anatomy induces an implicit order on the bound facts and, indirectly, the order of the EDB; choosing a specific applicable rule prioritizes the application of base vs inductive cases. A combination of the two handles can be used to define specific visits in the graph. Preferring inductive cases to base cases gives rise to depth-first exploration; vice versa, prioritizing base cases produces breadth-first ones. In depth-first traversals, nondeterministic choices of paths are more relevant than in breadh-first and are taken by prioritizing unifiers.

The VADALOG System does not directly adopt the chase procedure, but follows the architecture of traditional relational DBMSs, encoding Datalog rules in terms of reasoning plans, where specific implementations of relational algebra operators are considered —multiple join versions exist— and a set of so-called *routing strategies* are used to decide on rule application priority. These two degrees of freedom allow to act on the anatomy/morphology handles: different join implementations result in different unifiers being applied (anatomy) and the routing strategy is an encoding of the unification morphology. This means that graph traversal strategies in VADALOG can be controlled by choosing routing strategies and join implementations, opening the way to the development of *graph-based optimizers* that compile execution plans into reasoning plans on the basis of specific cost-based heuristics evaluated against the EDB (defining the structure of the graph) and the IDB (encoding the specific problem to be solved). For instance, for our st-connectivity instance, a hybrid depth-/breadth-first approach would pay off, by first optimistically trying multiple direct and deep connectivity paths (even driven by some heuristics, in more sophisticated settings) and eventually resorting to breadth-first search in case of failure.

VADALOG offers multiple routing strategies, e.g., *round-robin* (RR) and *EDB-first*, and join implementations, e.g., the standard *nested-loop join* (NLJ) and *depth-search join* (DSJ), an original implementation we present in this work, specifically devised for depth-first traversals. It is intuitive to understand how a combination of RR and NLJ can be used to simulate general purpose breadth-first traversals. In the next section we briefly describe DSJ, which addresses depth-first exploration.

Algorithm 1: Depth-search Join

```
1  static LSTACK, static RCUR; match = false;
2  L, RCUR_POS = LSTACK.pop();          ▷ Skip visited tuples (cycle avoidance)
3  R = RCUR[RCUR_POS];                                  ▷ Position is 0 by default
4  while !match do
5      match = tryJoin(L,R);
6      while !match and RCUR_POS<length(RCUR)-1 do
7          RCUR_POS++;
8          R = RCUR[RCUR_POS];
9          match = tryJoin(L,R);
10     if !match then
11         L, RCUR_POS = LSTACK.pop();
12         RCUR.pos(RCUR_POS);
13     else
14         LSTACK.push((L,RCUR_POS+1));
15         LSTACK.push((join(L,R),0));
16 return match;
```

4 Depth-search Join

Depth-search join is a join algorithm specifically designed to support recursive Datalog rules. Indeed, graph traversal is typically expressed by means of left- or right-recursive rules. For the sake of simplicity and w.l.o.g. here we consider left recursion (e.g., Rule (2) in Example 1). The key idea of DSJ is prioritizing the unification of facts generated by recursive cases over those originating from the base cases. For instance, w.r.t. our example, given a set of possible unifiers for $Connected(x, y)$, $Linked(y, z)$, the ones that bind $Connected(x, y)$ to facts deriving from many applications of Rule (2) take priority over those generated by fewer applications. The rationale is that each activation of Rule (2) applies a depth-first traversal step from which the next activations of Rule (2) must take on. Conversely, NLJ would entail a breadth-first behaviour, exploring all direct connections for each single binding of $Connected(x, y)$.

Algorithm 1 is the pseudocode of the implementation in the VADALOG System of DSJ. We consider binary joins and initialize a stack LSTACK and a cursor RCUR, holding the left-hand and right-hand side join operands, respectively. Each element of LSTACK is a pair composed of a left-hand fact and the last right-hand fact that has been considered in the join; RCUR is assumed to support absolute positions (basically acting as a map or a dictionary).

At the basis of the algorithm, there is the idea of modifying the standard NLJ by introducing a stack to induce the depth-first behaviour. In particular, facts resulting from successful joins (`tryJoin()` primitive) are pushed into the stack (line 15) and popped (lines 2 and 11) so as to take priority over other elements of the left-hand operand. Backtracking is performed by saving and resuming left-hand positions from LSTACK (line 14). The outer loop evaluates the join between left-hand fact (L) and right-hand fact (R) (lines 5-6); the inner loop

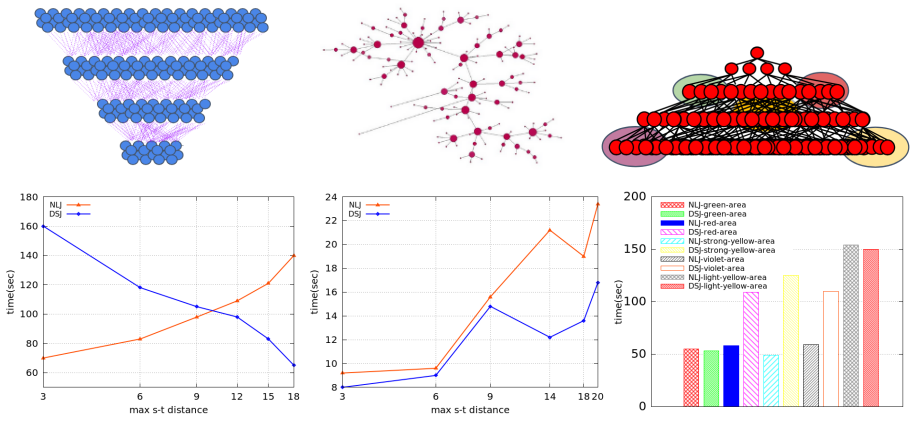
scans the right-hand table until a match is found (lines 6-9); if the join condition is not satisfied, next L is considered (line 10-12), else the current join status is pushed into LSTACK to be resumed. The algorithm returns a Boolean value representing the availability of join tuples, in a streaming fashion so that calling filters in the pipeline can fetch the respective tuples, whenever `true` is returned.

Observe that in general, infinite cyclic invocations are avoided by the `pop()` primitive, which is customized to skip the already visited facts with a marking mechanism that in VADALOG is known as *termination strategy* [5].

5 Experiments

We compared NLJ and DSJ in the VADALOG System in st-connectivity scenarios with different KGs to highlight graph traversal strategies. The three scenarios present different graph size and density configurations, that we chose to analyze the peculiarities of the two algorithms.

Test configuration. We invoked VADALOG via its REST interface and used CSV data to make tests independent of host-side optimizations. We ran each experiment ten times, averaging the elapsed times. We used a cloud instance of the VADALOG System, running in a Linux machine with Ubuntu 18.04.4 LTS with 16 cores and 126 GB of RAM. The reasoning tasks are executed without any use of concurrency or distribution techniques. Any kind of materialization or pre-sorting techniques affecting the input data has been avoided.



(a) Italian company KG (b) Barabási-Albert KG (c) Balanced KG

Fig. 2: Experimental evaluations on real-world and synthetic KGs.

Italian company network scenario (Figure 2(a)). A large real-world scenario with a KG from Bank of Italy about companies (nodes are companies and edges are shareholding relationships). This graph is composed of $\sim 6M$ companies and $\sim$ 6.2M ownership edges. The graph structure is depicted in Figure 2(a): it has a tree-like shape, where the node density decreases with tree level. The longest

shareholding chain between two companies is composed of 18 intermediate nodes, with $\sim 8K$ companies in the last layer. We tested different edge subsets of the graph considering increasing values for the maximum distance between source and target node.

NLJ showed to be faster for shallow graphs, and DSJ clearly outperforms NLJ when distance increases. This is exactly what expected of a proper depth-first traversal, provided that for higher distance the average node degree decreases and so does the likelihood of meeting the target node.

Synthetic Barabási-Albert scenario (Figure 2(b)). We analyzed the behaviour of the joins with an artificial setting, generated with the Barabási-Albert algorithm [10] for scale-free networks. We considered a graph with $\sim$ 1M nodes and $\sim$ 1M edges and of lower density and higher node distance than the real-world graph. The graph structure that is depicted in Figure 2(b) shows the presence of hubs, where the local density is higher, while it exponentially decreases along the paths. We tested st-connectivity for increasing values of s-t distance.

While for low distances NLJ and DSJ show a similar behaviour, when the analyzed depth grows, DSJ is remarkably faster and outperforms NLJ, though graph irregularities produce visible fluctuations in elapsed times.

Synthetic Balanced KG scenario (Figure 2(c)). In this scenario we wanted to evaluate the NLJ and DSJ behavior applied on a tree-search setting, in order to confirm the actual breadth- and depth-first behaviour of NLJ and DSJ, respectively. We built a balanced tree with a branching factor of 4 and height 11. In this tree structure, in contrast to the previous scenarios, the density grows with the tree level. We tested st-connectivity choosing the target nodes from five different areas (denoted by different colors in Figure 2(c)), while s was fixed as the tree root.

Our results confirm that NLJ and DSJ behave respectively as a breadth- and depth-first search. While for the violet and green areas DSJ outperforms NLJ, on the other hand, NLJ wins in light-yellow and red areas, coherently with what we expected with the given graph topology. Finally, for target nodes in the central area (strong-yellow) we obtain similar times for NLJ and DSJ.

6 Conclusion and Future Work

Reasoning on KGs is gaining more and more attention in AI venues, with graph databases increasingly adopted in many domains. While full ontological reasoning calls for a toolbox of sophisticated techniques and algorithms, state-of-the-art reasoners such as VADALOG rely on the vast amount of experience in DBMS architectures and adopt reasoning plans, along the lines of relational query plans. In this work we suggested a twofold handle to induce graph traversal strategies by tweaking standard logical unification and join algorithms.

We believe this technique will lay the basis for graph-aware relational-like optimizers, able to bend —when properly complemented with graph statistics— reasoning strategies to graph topologies.

Acknowledgements. The work on this paper was supported by EPSRC programme grant EP/M025268/1, the EU H2020 grant 809965, and the Vienna Science and Technology (WWTF) grant VRG18-013.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. R. Angles. The property graph database model. In *AMW*, 2018.
3. L. Bellomarini, D. Benedetto, G. Gottlob, and E. Sallinger. Vadalog: A modern architecture for automated reasoning with large knowledge graphs. *Information Systems*, page 101528, 2020.
4. L. Bellomarini, D. Fakhoury, G. Gottlob, and E. Sallinger. Knowledge graphs and enterprise AI: the promise of an enabling technology. In *ICDE*, pages 26–37. IEEE, 2019.
5. L. Bellomarini, E. Sallinger, and G. Gottlob. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. In *VLDB*, 2018.
6. M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *PODS*, pages 37–52, 2017.
7. R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.
8. G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.
9. A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, et al. Knowledge graphs. *arXiv preprint arXiv:2003.02320*, 2020.
10. C. A. H. R. and A. Barabási. Scale-free networks. *Scholarpedia*, 3(1):1716, 2008.
11. M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.