Reasoning at Scale

Urbani, Jacopo

published in
Encyclopedia of Big Data Technologies
2018

DOI (link to publisher)
10.1007/978-3-319-63962-8_317-1

document version
Publisher's PDF, also known as Version of record

R

Reasoning at Scale

Jacopo Urbani
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

Synonyms

Inference; Knowledge base; Rule-based processing

Definition

Reasoning is the process of deriving new conclusions from knowledge bases using a series of logical steps. Reasoning at scale refers to the ability of applying this process to very large knowledge bases, such as modern knowledge graphs that are available on the Web.

Overview

The Web contains a very large amount of semi-structured datasets that cover encyclopedic knowledge, social or co-authorship networks, experimental results, etc. This data is encoded using RDF (Brickley et al. 2014), and it is interlinked to each other following the principles of linked open data, thus forming a large network of datasets called the Web of Data (WoD) (Bizer et al. 2009).

Automated reasoning can derive a wealth of nontrivial knowledge from these datasets, which can be used, for instance, to augment the WoD with new knowledge or to detect inconsistencies. Unfortunately, the large size of the WoD makes reasoning a challenging task. In fact, the datasets might be too large to be stored in a single machine, requiring thus some form of distributed computing. Moreover, parallelizing the computation is not trivial due to factors like the input skewness or special corner cases that require sequential processing.

What constitutes the state of the art for reasoning on a large scale? To answer this question, this chapter offers a broad overview of the most recent efforts to execute rule-based reasoning on large inputs. More in particular, it describes the most important optimizations that can be applied to improve the efficiency of reasoning. Some of these optimizations work only with specific rules, while others are more generic. Even though none of them work with all possible inputs, in practice they turned out to be very effective as they enabled reasoning on large knowledge bases, with up to 100 billion triples in the largest experiments.

What Is Scale?

Generally speaking, the term scalability refers to the ability of a system to handle larger instances of a given problem. There can be several reasons that hinder the scalability of a system. First,

S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_317-1

the problem might have an unfavorable computational complexity which precludes termination within a reasonable time. Second, the algorithms might be poorly implemented. In this case, the system cannot scale well despite the problem is tractable. Third, the hardware might not have enough resources to carry on the computation. Scalability is thus a property that can be judged from three different angles: the theoretical complexity, the implementation, and the hardware requirements.

Before discussing the state of the art and describe how it deals with these challenges, it is important to define more precisely what reasoning is supposed to compute. Let KB be a generic RDF dataset, that is, a set of RDF statements. This dataset can be represented as a labeled directed graph where each triple $\langle s, p, o \rangle$ maps to an edge that connects s to o and is labeled with p. These graphs are typically called *knowledge graphs*. Let $KG = (V, E)$ be such a knowledge graph where V is the set of entities and E the labeled edges that connect the entities. Given in input a knowledge graph KG, the goal of reasoning is to derive new knowledge that can be inferred from KG. This knowledge takes the form of new triples which can be logically deduced from the KG. For now, it is assumed that V is complete, i.e., KG already contains all the entities of interest. With this assumption in mind, then the triples derived from reasoning can be represented by new edges in KG.

The derivation of new triples is determined by a set of rules, which must be provided as input. Rules are expressions of the form

$$B_1, \ldots, B_n \rightarrow H \tag{1}$$

where $B_1, \ldots, B_n$ are called *atoms*. An *atom* is an expression $p(\mathbf{x})$ where p is a *predicate* and $\mathbf{x} = x_1, \ldots, x_m$ is a tuple of terms that can be either variables or constants. A *fact* is an atom without any variable. In our context, facts are used to represent the KG. They can be unary (e.g., to express the *isA* relation – $Person(Mark)$), binary (e.g., $livesIn(Mark, Amsterdam)$), or ternary (e.g., $T(Mark, livesIn, Amsterdam)$). The set of atoms $B_1, \ldots, B_n$ is called the rule's *body*, while H is the rule's *head*.

Notice that rules can be more complex than in (1). For instance, some body atoms might be negated, or the head might contain a conjunction of multiple atoms. All scalable approaches which will be discussed in this chapter assume that rules only contain positive atoms and only one atom occurs in the head of the rule. Moreover, they assume that every variable in the head must also appear in the body (safeness condition). The reason behind these constraints is that they simplify the computation. For instance, negation can introduce non-determinism, while dropping safeness might lead to nontermination.

The computation of the rules can be formalized as follows. Let I be a generic database of facts (i.e., the input KG); σ be a *substitution*, i.e., a partial mapping from variables to other variables or constants; and $r \in P$ be a rule of the form (1) in the program P. Then, $r(I) = \{H\sigma \mid B_1\sigma, \ldots, B_n\sigma \in I\}$ is the set of derivations that can be derived from I using r and $P(I) = \bigcup_{r \in P} r(I)$ is its extension to all rules in the program. The exhaustive application of all rules can be defined recursively by setting $P^0(I) = I$ and $P^{i+1}(I) = P^i(I) \cup P(P^i(I))$. Since the rules are safe and the set of constants is finite, there will be a j s.t. $P^{j+1}(I) = P^j(I)$. In this case, $P^j(I)$ is called the *closure* or *materialization* of I with P.

Materialization with Fixed Rules

Ontological languages are used to serialize semantic relations in RDF knowledge bases in a machine-readable format. For instance, they allow the user to define various semantic relations like subsumption between classes (e.g., *Student* is a subclass of *Person*) or specify that a relation is transitive (e.g., *ancestorOf* or *partOf*).

Ontological statements like the previous two examples can be translated into rules by either considering the standard constructs of the language or by also including the ontology at hand. The following example is useful to understand this difference.

Example 1 Let us assume that the KG contains the following ontological statements: $\langle$:Bob, isA, :Actor$\rangle$ and $\langle$:Actor, soc, :Man$\rangle$ where soc is an abbreviation for the standard RDF schema IRI of class subsumption.

A rule of the first type could be

$$T(A, \texttt{isA}, B), T(B, \texttt{soc}, C) \rightarrow T(A, \texttt{isA}, C) \quad (2)$$

while a rule of the second type could be

$$isA(A, \texttt{:Actor}) \rightarrow isA(A, \texttt{:Man}) \quad (3)$$

In the first case, rule (2) simply translates the inference that it is possible to obtain considering the isA and soc relations. The rule is domain-independent and can be applied to any KG. In contrast, rule (3) is simpler because it does not require any data join but it has the disadvantage that it can be applied only to the input dataset.

In this chapter, rules of the first type are called *standard rules* since they are derived by standard ontological languages like RDF schema (Brickley et al. 2014) or OWL (Motik et al. 2009). Standard rules are important because they are universal in the sense that they do not depend on a particular input. Therefore, it is possible to introduce tailor-made optimizations to speed up their execution without any loss of generality. However, there are cases when nonstandard rules are preferable since they might be easier to execute. The remaining of this section will describe five optimizations which are crucial to enhance the scalability of reasoning using standard rules. The next section will address scalability with nonstandard rules.

Split instance/schema triples. The first, and perhaps most effective, optimization on current knowledge bases consists of splitting the input statements between the ones that describe instances and the ones that describe the schema. The last type of statements is typically ontological statements that use constructs from the language (e.g., OWL). One key property of current large KGs is that they contain many more instance statements than schema ones, and this is important because there are many standard rules which have two body atoms, one which matches instance statements while the other matches schema ones (Urbani et al. 2012). Rule (2) is such an example: Here, typically there will be many more triples of the form $T(A, \texttt{isA}, B)$ than of the form $T(B, \texttt{soc}, C)$.

A strategy to parallelize the computation of such rules is to simply range-partition the instance triples and assign each range to a different processor. If the processors operate in separated memory spaces (e.g., different machines), then schema triples can be replicated on each space.

After the partitioning is done, the rule can be executed in parallel without any intermediate node communication. An example of such computation is graphically depicted in Fig. 1a.

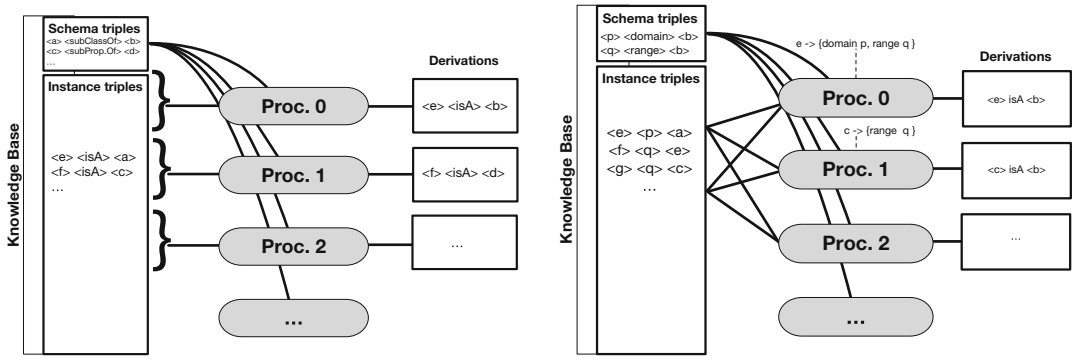
Reducing duplicates. There are cases where different rules might produce the same derivation. For instance, the two standard rules

$$T(A, P, B), T(P, \texttt{domain}, C) \rightarrow T(A, \texttt{isA}, C) \quad (4)$$

$$T(B, P, A), T(P, \texttt{range}, C) \rightarrow T(A, \texttt{isA}, C) \quad (5)$$

can derive the same information for an entity which is used both as a subject and as an object in different triples. For instance, $\langle Bob, worksIn, Amsterdam\rangle$ and $\langle Alice, daughterOf, Bob\rangle$ can both lead to the derivation $\langle Bob, isA, Person\rangle$ if the domain and range of the two predicates are $Person$.

A strategy to remove these types of duplicates is to group instance triples by the list of terms which are used in the head. In the previous case, triples can be grouped either by subject of by object, depending on the rule. Then, rules can be executed in parallel on each group. In this way, it is impossible that two different groups will produce the same derivation because the grouping criterion ensures that derivations must differ by at least one term (i.e., the grouping key).



Reasoning at Scale, Fig. 1 (**a**) Parallel execution of a rule that requires a join between schema and instance triples. (**b**) Execution of a group of rules: in this example, processor 0 receives all information regarding the entity "e," which allows it to apply both rules (4) and (5) and remove duplicates locally

During the rule computation, duplicates can still be produced within the same group. However, in this case they can be removed in parallel without any synchronization between the various processors. Figure 1b shows an example of such computation for rules (4) and (5).

***sameAs* table.** One category of rules which is widely used in modern knowledge bases encodes reasoning over the equality of concepts, which is a relation that is stated with the predicate *owl:sameAs*. Equality is transitive (if a is the same as b and b is the same as c, then a is the same as c) and symmetric (if a is the same as b, then b is the same as a); thus rules in this category produce a large number of materializations.

To improve the performance, reasoners typically avoid materializing all conclusions but instead build a dedicated table where groups of equal terms receive a unique representative ID, and all occurrences of these terms in the KB are replaced by the corresponding ID. During the materialization, the reasoner might derive more equality relations. If this happens, then the table needs to be updated, and more occurrences must be replaced with the corresponding ID. Notice that also rules need to be rewritten if they contain constants in the table.

After the materialization is computed, the augmented $P^j(I)$ will contain a compressed version of the full materialization since further derivations can be obtained by simply replacing the occurrences of representative IDs with each of the members of their equality group. However, in practice this operation is often omitted as it can be trivially computed on-the-fly whenever is needed.

Sorting rule execution. For some standard rulesets like the rules from RDF schema, it is possible to define a rule application order to reduce to the minimum the chance that the output of one rule can be used as input for another one. Unfortunately, this optimization guarantees a complete output only on some specific cases, and it is not possible to define an execution order which avoids repeated executions with more complex rulesets like, for instance, the OWL2 RL/RDF ruleset (Motik et al. 2009).

Memoization. Another technique that can be applied to improve the performance is *memoization* (Urbani et al. 2014). Memoization is a special type of caching which consists of storing the output of expensive functions to reduce the cost of repeated executions. For the problem of reasoning, memoization can be used to precompute all answers of some particular atoms. This enables a faster computation of data joins.

An example is useful to clarify this optimization. Let us consider, once again, rule (2). This rule contains two body atoms: One atom matches instance triples while the other matches schema

triples. During the materialization, the number of facts that match the second atom will change if other rules derive new triples with soc as predicate. With memoization, before the materialization starts a query-driven materialization procedure like QSQR (Abiteboul et al. 1995) or Magic set (Bancilhon et al. 1985), two well-known query-driven algorithms are invoked to compute all answers for the query $T(B, \texttt{soc}, C)$. Once this procedure is terminated, we can safely assume that the collection of facts that match this atom is immutable; thus the engine can index this collection more efficiently to facilitate the execution of the rule. In some cases, memoization does not lead to any reduction of the materialization runtime, while in other cases, the advantage is significant.

Materialization with Generic Rules

Nonstandard rules are typically easier to execute since they require less joins and predicates have a smaller arity (i.e., they are unary or binary only). On these rules, however, the previous optimizations might not be applicable. Still, the execution can be improved in two ways: Either by applying more general parallel algorithms or by considering multiple facts at the same time. Both types of improvements are described below.

Parallelizing rule execution. Three different types of parallelism can be applied to the rule execution: *Intra-rule parallelism*, *inter-rule parallelism*, and *instance-based parallelism*.

With intra-rule parallelism, the goal is to distribute the execution of a single rule among different processors. For example, let us consider rule (1). In this case, facts that match B_1 could be partitioned into n different partitions depending on the value of the terms that should be joined with B_2. Similarly, facts that match B_2 could be partitioned in an equivalent number of partitions in so that "B_1" and "B_2" facts with the same join terms will be in the same partition. In this way, each partition can be processed simultaneously by concurrent processors.

With inter-rule parallelism, the idea is to let concurrent processors execute different rules at the same time. For instance, one processor could execute rule (4) while another one execute rule (5). Notice that neither of these two types of parallelism is perfect: With intra-rule parallelism, the computation could be unbalanced if some partitions are much bigger than others. With inter-rule parallelism instead, the maximum number of concurrent processors is bound by the number of rules.

Intra- and inter-based parallelism are well known in literature and are also used in other scenarios. The third type of parallelism is a more recent variant which was first introduced in the RDFox system (Nenov et al. 2015). The idea is to let a number of concurrent processors to continuously pull not-yet-considered facts from a queue and verify whether they instantiate the body of a rule. If this occurs, then the system searches for other atoms in the database to compute a full rule instantiation. If this process succeeds, then the processor produces a new derivation and puts it back in the queue and database so that it can be further considered, possibly by other processors.

This type of parallelism is significantly different than the other two because here the processors do not receive a predefined amount of work but are free to "steal" computation from each other whenever they become idle. A limitation of this technique is that it requires a number of data structures that allow a fast concurrent access. While hash tables can provide this functionality, they have the disadvantage that they are not cache-friendly, that is, they do not use efficiently the CPU cache.

Set-based rule execution. Another technique for improving the performance consists of generating meta-facts which represents multiple sets of facts. This technique can be intuitively explained with a simple example. Let us consider the rule:

$$P(X, Y) \rightarrow Q(Y, X) \tag{6}$$

and assume that the input database does not contain any q-facts. In this case, it is clear that

each fact that matches the body will generate a new q-fact. Thus, the engine can simply create one single fact $q(y^*, x^*)$ where x^* and y^* are special terms which point to set of terms, namely, all first and second terms that appear in p-facts. For instance, if the database equals to $\{p(a,b), p(c,d)\}$, then $y^* \rightarrow \langle a, c\rangle$ and $x^* \rightarrow \langle b, d\rangle$. Notice that the engine does not need to explicitly materialize the lists $\langle a, c\rangle$ and $\langle b, d\rangle$ but can simply store instructions to compute this list on-the-fly in case it is needed.

This technique was first introduced in the VLog system (Urbani et al. 2016), and empirical results show excellent performance against the state of the art, especially because this technique becomes more effective with larger databases as potentially larger sets of facts can be compressed in a single meta-fact.

References

Abiteboul S, Hull R, Vianu V (1995) Foundations of databases, vol 8. Addison-Wesley, Reading

Bancilhon F, Maier D, Sagiv Y, Ullman JD (1985) Magic sets and other strange ways to implement logic programs. In: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on principles of database systems. ACM, pp 1–15

Bizer C, Heath T, Berners-Lee T (2009) Linked data-the story so far. Int J Semant Web Info Syst 5(3):1–22

Brickley D, Guha RV, McBride B (2014) RDF schema 1.1. W3C Recomm 25:2004–2014

Motik B, Grau BC, Horrocks I, Wu Z, Fokoue A, Lutz C, et al (2009) Owl 2 web ontology language profiles. W3C Recomm 27:61

Nenov Y, Piro R, Motik B, Horrocks I, Wu Z, Banerjee J (2015) RDFox: a highly-scalable RDF store. In: International semantic web conference. Springer, pp 3–20

Urbani J, Kotoulas S, Maassen J, Van Harmelen F, Bal H (2012) WebPIE: a web-scale parallel inference engine using mapreduce. Web Semant Sci Serv Agents World Wide Web 10:59–75

Urbani J, Piro R, van Harmelen F, Bal H (2014) Hybrid reasoning on OWL RL. Semantic Web 5(6):423–447. https://doi.org/10.3233/SW-130120

Urbani J, Jacobs C, Krötzsch M (2016) Column-oriented datalog materialization for large knowledge graphs. In: Proceedings of AAAI, pp 258–264