Fast Dual Simulation Processing of Graph Database Queries

Stephan Mennicke, Jan-Christoph Kalo, Denis Nagel, Hermann Kroll and Wolf-Tilo Balke
Institute for Information Systems, TU Braunschweig, Braunschweig, Germany
{mennicke, kalo, kroll, balke}@ifis.cs.tu-bs.de, denis.nagel@tu-bs.de

***Abstract*—Graph database query languages feature expressive yet computationally expensive pattern matching capabilities. Answering optional query clauses in SPARQL for instance renders the query evaluation problem immediately PSPACE-complete. Light-weight graph pattern matching relations, such as simulation, have recently been investigated as promising alternatives to more expensive query mechanisms like, e. g., computing subgraph isomorphism. Still, pattern matching alone lacks expressive query capabilities: graph patterns may be combined by usual inner joins. However, including more sophisticated operators is inevitable to make solutions more useful for emerging applications. In this paper we bridge this gap by introducing a new *dual simulation process* operating on SPARQL queries. In addition to supporting the full syntactic structure of SPARQL queries, it features polynomial-time pattern matching to compute an overapproximation of the query results. Moreover, to achieve running times competing with state-of-the-art database systems, we develop a novel algorithmic solution to dual simulation graph pattern matching, based on a system of inequalities that allows for several optimization heuristics. Finally, we achieve soundness of our process for SPARQL queries including UNION, AND and OPTIONAL operators not restricted to well-designed patterns. Our experiments on synthetic and real-world graph data promise a clear gain for graph database systems when incorporating the new dual simulation techniques.**

***Index Terms*—graph databases, query processing, graph simulation, SPARQL**

I. Introduction

Extensive knowledge graphs are commonplace backbones in today's information infrastructures. Therefore, scalable query processing in graph databases has sparked a vivid interest in the database community. Already at an early stage specialized graph query languages such as SPARQL, the W3C recommendation for querying RDF data by SQL-like expressions [28], have been designed. Such languages provide easy to use yet expressive query capabilities on graph structures, but need to severely break down structural complexity to allow for fast query evaluation. Indeed, the evaluation of complex graph patterns is computationally expensive and thus a variety of implementational avenues have been proposed [5], [9], [24].

At the heart of SPARQL, basic graph patterns (BGPs) form the syntactically least complex queries. BGPs are simply graphs, and their result sets contain all graph-homomorphic matches from the graph database instance. Consider query ($\mathcal{X}_1$), retrieving all persons (cf. variable `?director`) who directed at least one movie (`?movie`) and at some point collaborated with another person (`?coworker`):

```
SELECT * WHERE {
  ?director directed ?movie .                    (X1)
  ?director worked_with ?coworker . }
```

($\mathcal{X}_1$) consists of two triple patterns. The first requires a `directed` link between assignments to variables `?director` and `?movie` while the second asks for `?director` to be in a `worked_with` relationship with an object matching `?coworker`. An evaluation of ($\mathcal{X}_1$) w. r. t. the database instance depicted in Fig. 1(a) retrieves the two subgraphs in bold print, including nodes `B. De Palma` or `G. Hamilton` assigned to variable `?director`.

Besides full-fledged graph query languages simpler graph pattern matching for diverse querying tasks raised a growing interest in the database community [8], [10]–[13], [16], [18], [23], [30]. Some of these applications employ a form of *simulation graph pattern matching*, showing computational advantages over homomorphic and isomorphic matching. Yet, an in-depth analysis of the approaches incorporating simulation [8], [18], [23], [30] reveals two shortcomings:

(1) The algorithms presented are *not specifically designed for graph database querying* tasks, in contrast to state-of-the-art graph database management systems like Virtuoso [9]. Thus, when it comes to performance evaluation of the simulation algorithms, they are only compared to subgraph isomorphism algorithms. But as their claimed application area is indeed database querying, it would only be fair to test these algorithms against established database systems, too (note that all isomorphism queries can be easily translated to SPARQL queries with conjunction and filter conditions [20]). While performance evaluations of graph pattern matching papers generally show good evaluation times, based on our experience we have reason to believe that Virtuoso and other graph database systems would still perform much better. Therefore, we have to find out whether we can algorithmically catch up with graph database systems, since general simulation queries may not be easily expressed in SPARQL [20].

(2) What all the classical graph pattern matching problems have in common, is that *the input is given as a graph*, i. e., there is no possibility of building more complex patterns as by graph query languages. Hence, we have to study whether there are major boundaries for an incorporation of graph query operators into the pattern matching process.

Towards (1) we investigate dual simulation, a version of simulation specifically developed for the graph data setting [18].

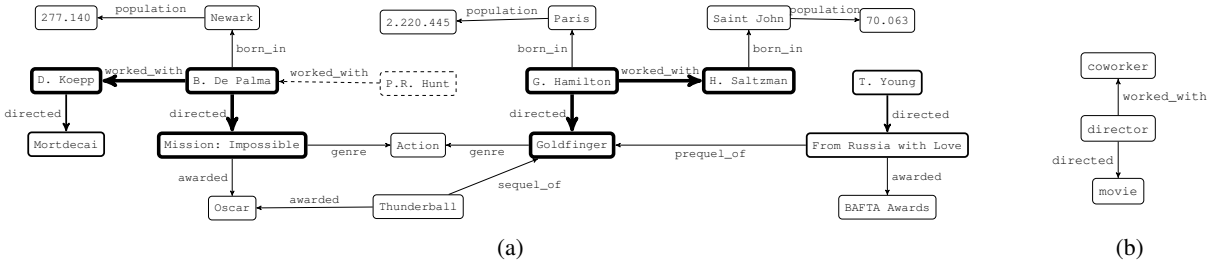

Fig. 1: Representation of (a) an Example Graph Database and (b) a Graph Pattern for ($\mathcal{X}_1$)

The algorithm presented by Ma et al. follows a single passive strategy that checks whether the definition of dual simulation is met resulting in a huge amount of iterations and influencing the overall runtime (cf. Table II). Based on a novel characterization of dual simulation in Sect. III, we develop a more flexible algorithmic solution to the dual simulation problem: the fixpoint of a system of inequalities (SOI) allows for *fast dual simulation processing* in the graph query setting. We provide formal proof of the correctness of our algorithm as well as experimental justification for the performance improvements brought by our solution.

Regarding (2), we also contribute a conservative extension of dual simulation to work with typical graph query operators, exemplarily taken from SPARQL (cf. Sect. IV). We obtain an overapproximation of the actual SPARQL query results for further inspection, filtering, or actual query processing, depending on the specific application. These extensions are *complete* in that none of the matches under the SPARQL semantics is neglected by dual simulation. In particular, this allows for *sound* pruning and in any case makes it safe to use the result for further query processing. Our algorithmic framework remains efficient, since all the features we need to add are directly implementable within the SOI solution and do not influence the overall polynomial-time complexity.

In Sect. V, we perform extensive experiments on two large-scale databases. First, we provide evidence of the runtime improvements over the algorithm by Ma et al. due to our solution. Second, we step into one possible application, namely *per-query database pruning*. More than 95% irrelevant triples are disqualified by dual simulation processing for all evaluated queries, which is the reason for improved query evaluation times compared to two state-of-the-art graph databases Virtuoso [9] and RDFox [24]. Moreover, we observe that our dual simulation process may directly be incorporated as a pruning preprocessing step in RDFox. In Sect. VI, we elaborate on related work while we draw a conclusion in Sect. VII. Most proofs of the formal results had to be omitted due to space limitations. However, they can be found in the supplementary report [19] accompanying this paper.

II. GRAPHS, DATA AND MATCHING

By *graphs* we refer to edge-labeled directed graphs with a finite set of nodes V, a finite label alphabet Σ, and a directed labeled edge relation $E \subseteq V \times \Sigma \times V$. A *graph* is a triple $G = (V, \Sigma, E)$ of the aforementioned components. As exemplified in Fig. 1, nodes are depicted as rounded-corner rectangles (with its identifier/name as centered label) while edges are represented by directed arrows (with associated labels next to the arrow) between nodes. We often identify the components of of graphs G_i by V_i and E_i ($i \in \mathbb{N}$). As a matter of simplicity we assume all graphs to be labeled over a fixed alphabet Σ. For every label $a \in \Sigma$, we associate with graphs G two adjacency maps, a forward map $\mathfrak{F}^a_G$ and a backward map $\mathfrak{B}^a_G$ of G. Both mappings associate a subset of nodes with each node $v \in V$, in case of forward maps, the set of successor nodes, and in case of backward maps, the set of predecessor nodes of v, i. e., $\mathfrak{F}^a_G(v) := \{w \mid (v, a, w) \in E\}$ and $\mathfrak{B}^a_G(v) := \{u \mid (u, a, v) \in E\}$.

Graph databases are graphs which associate database objects, e. g., entities and literals, with each other via predicates. To distinguish graph databases from ordinary graphs we denote them by $DB = (O_{DB}, \Sigma, E_{DB})$, where O_{DB} is the set of database objects and elements of E_{DB} are sometimes called links. We refrain from making the data model more concrete, since all upcoming notions and techniques are independent of any further restrictions, as e. g., given by RDF's requirement of having literals only as edge targets.

A dual simulation [18] between two graphs G_1, G_2 is a binary relation $S \subseteq V_1 \times V_2$ such that for each pair of nodes $(v_1, v_2) \in S$, all incoming and outgoing edges of v_1 are also featured by v_2 and the adjacent nodes of v_1 and v_2, ordered in pairs, belong to S. For a dual simulation S, $(v_1, v_2) \in S$ means that v_2 dual simulates v_1. As an example consider the graphs depicted in Fig. 2(a) and (b) as G_1 and G_2. A dual simulation relates the nodes with the same label, e. g., `place` in G_2 dual simulates node `place` in G_1, and both nodes `director1` and `director2` in G_1 relate to `director` in

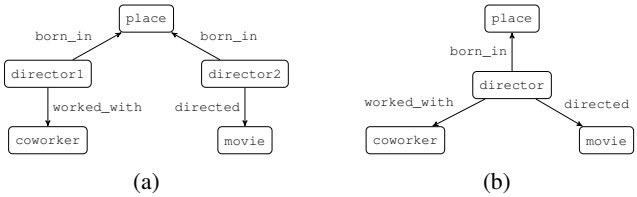

Fig. 2: Two Graph Patterns

TABLE I: Summary of Symbols

$G = (V, \Sigma, E)$	edge-labeled directed graph
$\mathfrak{F}^a_G$, $\mathfrak{B}^a_G$	forward/backward map for label a in G
$DB = (O_{DB}, \Sigma, E_{DB})$	graph database
$\chi_S : V_1 \to 2^{V_2}$	characteristic function for relation $S \subseteq V_1 \times V_2$
$\mathcal{Q}$, $\mathcal{Q}_1$, $\mathcal{Q}_2$	SPARQL queries
$[\![\mathcal{Q}]\!]_{DB}$	set of matches due to SPARQL semantics
$\mu : vars(\mathcal{Q}) \to O_{DB}$	match to query $\mathcal{Q}$ in DB
$\mu_1 \rightleftharpoons \mu_2$	compatibility predicate between μ_1 and μ_2
$\mathcal{E} = (\mathtt{Var}, \mathtt{Eq})$	system of inequalities

G_2, as in

$$\left\{ \begin{array}{l} (\mathtt{place},\mathtt{place}),(\mathtt{director1},\mathtt{director}), \\ (\mathtt{director2},\mathtt{director}),(\mathtt{movie},\mathtt{movie}), \\ (\mathtt{coworker},\mathtt{coworker}) \end{array} \right\} \quad (1)$$

Node `director2` features two outgoing edges, one labeled `born_in` to node `place`, the other labeled `directed` to `movie`. Node `director` in G_2 dual simulates `director2`, since it has an outgoing edge with label `born_in` to node `place`, and `place` in G_2 dual simulates `place` in G_1. The same argument holds for node `movie`. By following through the argumentation for every pair of nodes in (1), it can be shown that G_2 indeed dual simulates G_1 under the indicated dual simulation (1). Observe that a single node, e. g., `director`, may dual simulate more than one node.

Definition 1 (Dual Simulation [18]) Let $G_i = (V_i, \Sigma, E_i)$ $(i = 1, 2)$ be two graphs. A relation $S \subseteq V_1 \times V_2$ is a *dual simulation between G_1 and G_2* iff for each $(v_1, v_2) \in S$,

(i) $(v_1, a, w_1) \in E_1$ implies $\exists w_2 \in V_2 : (v_2, a, w_2) \in E_2$ and $(w_1, w_2) \in S$,

(ii) $(u_1, a, v_1) \in E_1$ implies $\exists u_2 \in V_2 : (u_2, a, v_2) \in E_2$ and $(u_1, u_2) \in S$.

We say that G_2 *dual simulates* G_1 iff there is a non-empty dual simulation between G_1 and G_2. ■

Note that the trivial dual simulation $S = \emptyset$ would certify that any two graphs are dual simulating each other. In a graph query setting we call G_1 *pattern graph* and G_2 is the *graph database*. Reconsider the introductory example query $(\mathcal{X}_1)$. The graph in Fig. 2(b) dual simulates the graph representation of $(\mathcal{X}_1)$ in Fig. 1(b). A dual simulation is realized by ignoring node `place`. Hence, not every node of the graph database has to participate in a dual simulation relation. Furthermore, the graph in Fig. 2(a) neither dual simulates nor is dual simulated by the graph in Fig. 1(b). Regarding the graph database depicted in Fig. 1(a) and the graph representation of $(\mathcal{X}_1)$ in Fig. 1(b), dual simulation (2) turns out to be particularly useful in the upcoming sections.

$$\left\{ \begin{array}{l} (\mathtt{director},\mathtt{B.\ De\ Palma}),(\mathtt{director},\mathtt{G.\ Hamilton}), \\ (\mathtt{coworker},\mathtt{D.\ Koepp}),(\mathtt{coworker},\mathtt{H.\ Saltzman}), \\ (\mathtt{movie},\mathtt{Mission:\ Impossible}),(\mathtt{movie},\mathtt{Goldfinger}) \end{array} \right\} \quad (2)$$

It comprises exactly the nodes of the two subgraphs from the result set of $(\mathcal{X}_1)$. Instead of considering the full graph database (i. e., Fig. 1(a)) we would ignore all graph database nodes but those mentioned by dual simulation (2). Computing this dual simulation is possible in PTIME [18], as opposed to SPARQL query evaluation being PSPACE-complete [26], [29]. How to perform this computation fast is subject to the next section. We apply dual simulation principles to SPARQL for query processing in Sect. IV.

III. A PERSPECTIVE ON DUAL SIMULATION

At the end of the last section we have seen a dual simulation between a graph representation of a SPARQL query (BGP $(\mathcal{X}_1)$) and a graph database (Fig. 1(a)), covering all nodes relevant for computing the result set of $(\mathcal{X}_1)$. In Sect. IV we show that the existence of such a dual simulation is not coincidental, since every match for SPARQL queries like $(\mathcal{X}_1)$ is contained in a maximal dual simulation (cf. Theorem 1). A dual simulation S is maximal iff there is no dual simulation S' such that $S \subset S'$. Fortunately, there is exactly one such maximal dual simulation between any two graphs, the *largest dual simulation*.

Proposition 1 (Proposition 2.1 [18]) *For any two graphs G_1 and G_2, there is a unique largest dual simulation $S_{\max}$ between G_1 and G_2, i. e., for any dual simulation S between G_1 and G_2, $S \subseteq S_{\max}$.*

The proof exploits the fact that, whenever we have two dual simulations S_1 and S_2 between the graphs, their union $S_1 \cup S_2$ is a dual simulation. Incorporating dual simulation in graph pattern matching or SPARQL query processing amounts to computing the largest dual simulation between an appropriate representation of the query and the graph database. All graph database nodes captured by the largest dual simulation are relevant for answering the query.

Computing the largest (dual) simulation is the algorithmic basis for solving the graph (dual) simulation problem, i. e., given two graphs G_1 and G_2, does G_2 (dual) simulate G_1. To the best of our knowledge, all published algorithms for this task [15], [18] work on the same principles. Starting with the largest possible relation between the two node sets, the algorithms incrementally disqualify pairs of nodes violating Def. 1. The procedures are guaranteed to terminate when no pair of nodes can be disqualified anymore. Although the standard algorithms share an $\mathcal{O}(|V_2|^3)$ data (runtime) complexity, we observed that these algorithms only allow for the naive evaluation strategy described above, which have originally been invented for comparing graphs of unknown sizes with each other. The aforementioned data complexity follows from generalizing the existing algorithms [15] and [18] to edge-labeled graphs (cf. our supplementary report [19] for a detailed derivation). This inflexibility generates high query running times that would easily be outperformed by state-of-the-art query evaluation, e. g., by Virtuoso (cf. Sect. V).

Subsequently, we develop a novel solution which computes the largest dual simulation and exploits run-time analytics to dynamically adapt evaluation strategies. Key to our solution is the reformulation of the algorithm as a system of inequalities which allows for two dynamically interchangeable evaluation strategies. Although the worst-case complexity of our solution remains unaltered (cf. Sect. III-C), compared to the existing

algorithms, we gain a degree of freedom allowing for a systematic reduction of iterations to eventually reach the largest dual simulation (cf. Sect. III-C). As we show in Sect. V the new procedure shows extremely low computation times, a solid basis for query processing. Our solution is engineered in three steps. First, we define a set of inequalities equivalent to the coinductive definition of dual simulation in Def. 1. We further show how to derive a fast implementation based on bit-vectors and bit-matrices. Last, we provide a discussion on optimizations realized in our software prototype[1].

A. Groundwork

Any binary relation $R \subseteq A \times B$, over sets A and B, has a characteristic function $\chi_R : A \to 2^B$ with $\chi_R(a) := \{b \in B \mid (a,b) \in R\}$. For a dual simulation S between graphs G_1 and G_2, χ_S associates with each node $v \in V_1$ a set of dual simulating nodes $\chi_S(v) \subseteq V_2$. Consider an edge (v,a,w) of G_1 and node $v' \in \chi_S(v)$. If S is a dual simulation, then for χ_S we derive

$$\exists w' : (v',a,w') \in E_2 \text{ and } w' \in \chi_S(w). \tag{3}$$

The problem with (3) is that there may be many w' qualifying for $(v',a,w') \in E_2$ but $w' \notin \chi_S(w)$. We pursue to have a single operation allowing us to quickly verify the existence of w'. Therefore, recall that for any graph, here graph database G_2, we have a forward adjacency map $\mathfrak{F}^a_{G_2}$ for each label $a \in \Sigma$ (cf. Sect. II). By exploiting these maps we prove existence of a w' in (3) simply by intersecting the row of v' in $\mathfrak{F}^a_{G_2}$ and the nodes simulating w, i. e.,

$$\mathfrak{F}^a_{G_2}(v') \cap \chi_S(w) \neq \emptyset. \tag{4}$$

(4) still only checks for one pair of nodes (v,v'). Combining this equation for all $v' \in \chi_S(v)$ yields

$$\textstyle\bigwedge_{v' \in \chi_S(v)} \quad \mathfrak{F}^a_{G_2}(v') \cap \chi_S(w) \neq \emptyset. \tag{5}$$

The same encoding applies to Def. 1(ii), this time using the backward map,

$$\textstyle\bigwedge_{w' \in \chi_S(w)} \quad \mathfrak{B}^a_{G_2}(w') \cap \chi_S(v) \neq \emptyset. \tag{6}$$

The combination of both equations (5) and (6) yields two inequalities equivalent to the definition of dual simulation and the key for our efficient implementation.

Lemma 1 *Let $G_1 = (V_1, \Sigma, E_1)$ and $G_2 = (V_2, \Sigma, E_2)$ be graphs with $(v,a,w) \in E_1$. For a binary relation $S \subseteq V_1 \times V_2$ satisfying* (5) *and* (6)*, it holds that* (7) *is satisfied.*

$$\begin{array}{lrcl} (i) & \chi_S(w) & \subseteq & \bigcup_{v' \in \chi_S(v)} \mathfrak{F}^a_{G_2}(v') \\ (ii) & \chi_S(v) & \subseteq & \bigcup_{w' \in \chi_S(w)} \mathfrak{B}^a_{G_2}(w') \end{array} \quad \text{and} \tag{7}$$

PROOF: W. l. o. g., we show inequality (i) only. Inequality (ii) is completely analogous. Towards a contradiction assume $\chi_S(w) \not\subseteq \bigcup_{v' \in \chi_S(v)} \mathfrak{F}^a_{G_2}(v')$. Hence, there is a $w' \in \chi_S(w)$ such that for each $v' \in \chi_S(v)$, $w' \notin \mathfrak{F}^a_{G_2}(v')$, i. e., $(v',a,w') \notin E_2$. As a consequence, $\chi_S(v)$ and $\mathfrak{B}^a_{G_2}(w')$ are disjoint for each $v' \in \chi_S(v)$, contradicting our assumption that (6) holds. Therefore, such a w' cannot exist, allowing to conclude that $\chi_S(w) \subseteq \bigcup_{v' \in \chi_S(v)} \mathfrak{F}^a_{G_2}(v')$. □

Phrased differently, dual simulations S satisfy (7) for every edge (v,a,w) of G_1. Lemma 1 reveals an important observation that, to the best of our knowledge, has not been published so far: The reason why (7) holds is that part (ii) prevents part (i) from getting ill-formed and vice versa. The fast algorithm we obtain here is a consequence of the duality in dual simulation. Conversely, every solution to (7) is a dual simulation.

Proposition 2 *Let G_1 and G_2 be graphs. $S \subseteq V_1 \times V_2$ is a dual simulation between G_1 and G_2 iff for every edge $(v,a,w) \in E_1$,* (7) *holds for S.*

The proof builds on the principles of Lemma 1 and can be found in our report [19]. Hence, (7) characterizes dual simulations, and we can use it to compute the largest dual simulation. The algorithm works as follows. We begin with $S_0 := V_1 \times V_2$. For each edge of G_1, check whether (7) is satisfied by S_0. Assume (7)(i) fails for an edge (v,a,w). Then, S_1 is computed by $\chi_{S_1}(u) := \chi_{S_0}(u)$ for $u \neq w$ and $\chi_{S_1}(w) := \chi_{S_0}(w) \cap \bigcup_{v' \in \chi_{S_0}(v)} \mathfrak{F}^a_{G_2}(v')$. We get rid of all non-simulating nodes of w relative to S_0 in a single iteration. This procedure is repeated for $S_1, S_2, \ldots$ until we reach an S_k satisfying (7) for every edge of G_1.

Even though we maintain the PTIME nature of other algorithms (cf. Sect. III-C), we still miss a way to quickly compute $\bigcup_{v' \in \chi_S(v)} \mathfrak{F}^a_{G_2}(v')$ and access $\chi_S(v)$. Therefore, the forthcoming implementation works with bit-representations of $\chi_S(v)$ and $\mathfrak{F}^a_{G_2}, \mathfrak{B}^a_{G_2}$, paving the way for optimization in time- and space-consumption (e. g., [5]). In that setting we derive a *system of inequalities* (SOI) from Prop. 2, for which dual simulations S serve as valid assignments.

B. Engineering

Our goal is to obtain the facilities for achieving a fast implementation of dual simulation processing. Recall that we need to compute the largest dual simulation and we do this by a *system of inequalities* according to (7). The challenge is to find a way to quickly compute the unions

$$\textstyle\bigcup_{v' \in \chi_S(v)} \mathfrak{F}^a_{G_2}(v') \quad \text{and} \quad \bigcup_{w' \in \chi_S(w)} \mathfrak{B}^a_{G_2}(w'). \tag{8}$$

Combinations of vectors and matrices, especially when encoding information only bit-wise, promise fast computations. Hence, we interpret the adjacency maps of G_2 as adjacency bit matrices. Reconsider the graph in Fig. 2(a). For label `born_in`, this graph provides two adjacency matrices,

$$\mathfrak{F}^{\texttt{born\_in}}_{Fig.\ 2(a)} = \begin{pmatrix} 0&0&0&0&0 \\ 1&0&0&0&0 \\ 1&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix} \quad \text{and} \quad \mathfrak{B}^{\texttt{born\_in}}_{Fig.\ 2(a)} = \begin{pmatrix} 0&1&1&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{pmatrix}.$$

Here, we assume the set of nodes of graphs to be ordered by some pre-defined index, e. g., v_1 = `place`, v_2 = `director1`, v_3 = `director2`, v_4 = `coworker`, and

[1] available at GitHub https://github.com/ifis-tu-bs/sparqlSim

$v_5 = \texttt{movie}$. Also, χ_S can be seen as a matrix with $k = |V_1|$ rows, one for each node of pattern graph G_1, and $n = |V_2|$ columns. Specifically, for a dual simulation S a 1 in position (i, j) means that the i^{th} node of the pattern graph is simulated by the j^{th} node of the graph database. For ease of presentation, the pattern graph G_1 does not have an indexed node set. Consequently, for node v of the pattern graph and $j \leq n$, we access the j^{th} component of v's row by $\chi_S(v, j)$. By $\chi_S(v)$ we get v's row vector sliced from matrix χ_S. The desired unions (8) are now achieved by bit-matrix multiplications[2] (symbol $\times_b$),

$$\chi_S(v) \times_b \mathfrak{F}^a_{G_2} \quad \text{and} \quad \chi_S(w) \times_b \mathfrak{B}^a_{G_2}. \tag{9}$$

The result of the multiplication is the *reachable nodes via a-labeled (forward) edges* from any simulating node of v. For instance, assume that $\chi_S(\texttt{director}) = \chi_S(\texttt{place}) = (1, 1, 1, 1, 1)$. Then, for edge $(\texttt{director}, \texttt{born\_in}, \texttt{place})$:

$$\begin{array}{lcll} \chi_S(\texttt{director}) & \times_b & \mathfrak{F}^{\texttt{born\_in}}_{Fig.\ 2(a)} & = (1,0,0,0,0) = r_1 \\ \chi_S(\texttt{place}) & \times_b & \mathfrak{B}^{\texttt{born\_in}}_{Fig.\ 2(a)} & = (0,1,1,0,0) = r_2. \end{array}$$

Hence, r_1 reveals that only node `place` is reachable via forward edges labeled `born_in`. Conversely, by `born_in`-labeled backward edges we reach `director1` as well as `director2`. The results are used to update a given relation S, according to (7). In the example above, r_2 shows that $\chi_S(\texttt{director}) \neq (1, 1, 1, 1, 1)$, since the only reachable nodes are `director1` and `director2`. Thus, $\chi_S(\texttt{director}) = (1, 1, 1, 1, 1) \not\leq (0, 1, 1, 0, 0) = r_2$, but according to Prop. 2, a dual simulation S satisfies (7), now possible to formulate by bit-matrix operations for edges $(v, a, w) \in E_{G_1}$,

$$\begin{array}{rcl} \chi_S(w) & \leq & \chi_S(v) \times_b \mathfrak{F}^a_{G_2} \quad \text{and} \\ \chi_S(v) & \leq & \chi_S(w) \times_b \mathfrak{B}^a_{G_2}. \end{array} \tag{10}$$

After observing the wrong value of $\chi_S(\texttt{director})$ we update relation S to S' by $\chi_{S'}(\texttt{director}) := \chi_S(\texttt{director}) \wedge r_2$ (component-wise conjunction of the two vectors). This enables us to give an algorithm for the dual simulation problem between two graphs G_1 and G_2 as a solution of the system of inequalities $\mathcal{E} = (\mathsf{Var}, \mathsf{Eq})$, where every node v of the graph pattern is a variable, i. e., $\mathsf{Var} := V_1$, and Eq contains for each pattern edge $(v, a, w) \in E_1$, the following equations:

$$w \leq v \times_b \mathfrak{F}^a_{G_2} \quad \text{and} \quad v \leq w \times_b \mathfrak{B}^a_{G_2}. \tag{11}$$

Fig. 3 shows the SOI for computing dual simulations for the graphs in Fig. 2(a) and (b). Assignments to the variables $v, w \in \mathsf{Var}$ are relations $S \subseteq V_1 \times V_2$. The algorithm computing the largest dual simulation between G_1 and G_2 proceeds as follows.

1) Set $S_0 := V_1 \times V_2$ and all inequalities in Eq *unstable*.

[2]For vector $\mathfrak{v}$ and matrix $\mathfrak{A}$, $\mathfrak{v} \times_b \mathfrak{A} = \mathfrak{w}$ where $\mathfrak{w}(j) = 1$ iff there is an i such that $\mathfrak{v}(i) = 1$ and $\mathfrak{A}(i, j) = 1$.

$$\begin{array}{rcl} \texttt{place} & \leq & \texttt{director1} \times_b \mathfrak{F}^{\texttt{born\_in}}_{Fig.\ 2(b)} \\ \texttt{place} & \leq & \texttt{director2} \times_b \mathfrak{F}^{\texttt{born\_in}}_{Fig.\ 2(b)} \\ \texttt{director1} & \leq & \texttt{place} \times_b \mathfrak{B}^{\texttt{born\_in}}_{Fig.\ 2(b)} \\ \texttt{director2} & \leq & \texttt{place} \times_b \mathfrak{B}^{\texttt{born\_in}}_{Fig.\ 2(b)} \\ \texttt{coworker} & \leq & \texttt{director1} \times_b \mathfrak{F}^{\texttt{worked\_with}}_{Fig.\ 2(b)} \\ \texttt{director1} & \leq & \texttt{coworker} \times_b \mathfrak{B}^{\texttt{worked\_with}}_{Fig.\ 2(b)} \\ \texttt{movie} & \leq & \texttt{director2} \times_b \mathfrak{F}^{\texttt{directed}}_{Fig.\ 2(b)} \\ \texttt{director2} & \leq & \texttt{movie} \times_b \mathfrak{B}^{\texttt{directed}}_{Fig.\ 2(b)} \end{array}$$

Fig. 3: System of Inequalities Characterizing Largest Dual Simulation between Fig. 2(a) and (b)

2) Let S_i be the current candidate relation. Pick any unstable inequality $\epsilon \in \mathsf{Eq}$
   a) If S_i is valid for ϵ, set ϵ stable and continue with (2).
   b) If S_i is invalid for inequality $\epsilon = v \leq w \times_b \mathfrak{A}$ (for $\mathfrak{A} \in \{\mathfrak{F}^a_{G_2}, \mathfrak{B}^a_{G_2} \mid a \in \Sigma\}$), then $\chi_{S_i}(w) \times_b \mathfrak{A} = r$ and $\chi_S(v) \not\leq r$. Update S_i to S_{i+1} such that

$$\chi_{S_{i+1}}(x) := \begin{cases} \chi_{S_i}(x) \wedge r & \text{if } x = v \text{ and} \\ \chi_{S_i}(x) & \text{otherwise.} \end{cases}$$

   Furthermore, every inequality $y \leq v \times \mathfrak{A} \in \mathsf{Eq}$ are reset to unstable. Mark ϵ stable and continue with (2).

The initialization step of S_0 can also be expressed in terms of inequalities, in that for every pattern node v, we add (12) to the set of inequalities Eq.

$$v \leq \underline{\mathbf{1}} \tag{12}$$

$\underline{\mathbf{1}}$ is the vector containing a 1 in every component. The dual simulation given by (1) is the largest solution to the SOI in Fig. 3, thus it constitutes the largest dual simulation.

C. Complexity and Optimization

Initializing S_0 takes time $\mathcal{O}(|V_1| \cdot |V_2|)$ in a naive implementation. We execute step 2) at most $|V_1| \cdot |V_2|$ times, since there are $|V_1|$ pattern nodes for which at most $|V_2|$ data nodes can be disqualified. Let $\epsilon = v \leq w \times_b \mathfrak{A}$ be in Eq and S_i the current candidate relation. Computing $r = \chi_{S_i}(w) \times_b \mathfrak{A}$ is in $\mathcal{O}(|V_2|^2)$ time. By further regarding the intersection $\chi_{S_i}(v) \wedge r$, we obtain an overall time complexity of $\mathcal{O}(|V_2|^2 + |V_2|) = \mathcal{O}(|V_2|^2)$ for updating S_i to S_{i+1} which validates ϵ. For every edge in G_1 we have two equations in Eq, i. e., $|\mathsf{Eq}| = \mathcal{O}(|E_1|)$. Thus, assuming pattern G_1 and data graph G_2 as input, our algorithm has a combined complexity of $\mathcal{O}((|V_1| \cdot |V_2|) \cdot |E_1| \cdot |V_2|^2)$. In terms of data complexity we have a worst-case runtime of $\mathcal{O}(|V_2|^3)$, virtually the same complexity as of any other dual simulation algorithm (cf. [19]).

Our characterization of dual simulation and its implementation open up dynamic evaluation strategies for the constructed SOI. First, the order in which the equations are evaluated has an impact on the overall runtime. For our experiments, we have chosen an order that aims at shrinking the simulation as early as possible, e. g., by preferring inequalities with matrix components having more empty columns, which indicates sparsity of the respective matrices. Second, the computation

of r (step 2b of the algorithm) may be performed row-wise or column-wise. Again, we follow the strategy of fewer iterations, i. e., in $v \leq w \times_b \mathfrak{A}$ we choose a row-wise evaluation if and only if $\chi_S(w)$ has fewer bits set than $\chi_S(v)$. As it turns out (cf. Sect. V-C) there is not a single heuristic that fits all input patterns and databases.

Our proof-of-concept implementation keeps G_2 in memory by its adjacency matrices. G_1 is stored by its system of inequalities, including $\mathcal{O}(|V_1|)$ bit-vectors representing χ_S. For every graph pattern G_1, it suffices to load those adjacency matrices that are needed the pattern. Hence, the worst-case memory consumption is determined by the graph pattern and by the adjacency matrix requiring the most memory. Note that due to bit-vector storage techniques, such as gap-length encoding, the worst memory consumption might not occur with the label storing the most bits. Combined with the memory-economical implementation by Atre et al. [4], [5] we are quite optimistic that our implementation may directly be used within the preprocessing step of the *BitMat* tool set. Our dual simulation processing applied to SPARQL queries yields decent pruning factors (cf. Sect. V), significantly improving upon those reported by Atre [4].

IV. Dual Simulation for SPARQL

Having clarified the foundational and algorithmic aspects of dual simulations we now approach an actual query language, namely SPARQL. We choose SPARQL for its high-quality standardization by the W3C [28] and its extensive formal treatment, e. g., [2], [3], [26], [29]. Subsequently, for SPARQL's least complex construct we canonically obtain dual simulation processing respecting all matches any SPARQL query processor would find. We further discuss SPARQL's join operators. For each query language feature we obtain a soundness result guaranteeing that the original SPARQL matches are preserved for further processing.

A. Basic Graph Patterns

As for RDF, *triple patterns* are first-class citizens of SPARQL. For the presentation of the upcoming material, we assume subject and object of a triple $t = (s, p, o)$ to be variables from an infinite domain of variables $\mathcal{V}$, ranging over by $\mathtt{v}, \mathtt{v}_1, \mathtt{v}_2, \ldots$. A variable $\mathtt{v}_1$ is usually introduced by a leading question mark, i. e., $?\mathtt{v}_1$ (cf. ($\mathcal{X}_1$)). In formal notation, however, we drop this syntactic convention and write $\mathtt{v}_1$.

Querying a graph database $DB = (O_{DB}, \Sigma, E_{DB})$ yields a set of partial mappings from the set of variables to actual database objects. For instance, the single triple pattern $t = (\mathtt{v}_1, \mathtt{population}, \mathtt{v}_2)$ gives rise to a match identifying $\mathtt{v}_1$ with node `Saint Join` and $\mathtt{v}_2$ with `70.063` (cf. Fig. 1(a)). By $vars(t)$ we denote the set of variables occurring in triple t, i. e., $vars(t) = \{\mathtt{v}_1, \mathtt{v}_2\}$ for the abovementioned t. A *candidate in DB* is a partial function $\mu : \mathcal{V} \to O_{DB}$. $dom(\mu)$ denotes the set of variables for which candidate μ is defined. A candidate μ is a *match for triple t in DB* iff $dom(\mu) = vars(t)$ and, assuming $t = (\mathtt{v}_1, a, \mathtt{v}_2)$, $(\mu(\mathtt{v}_1), a, \mu(\mathtt{v}_2)) \in E_{DB}$, abbreviated by $\mu(t) \in DB$.

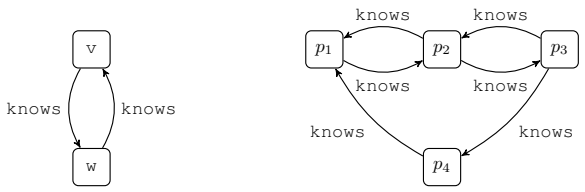

Fig. 4: (a) Graph Pattern P and (b) Graph Database K, an example adapted from Ma et al. [18]

We call sets of triple patterns $\mathbb{G}$ *basic graph patterns* (BGPs). Function *vars* and thereupon the notion of matches extend to BGPs by $vars(\mathbb{G}) = \bigcup_{t \in \mathbb{G}} vars(t)$, and μ is a match for $\mathbb{G}$ iff μ is a match for all triples $t \in \mathbb{G}$. The result set $[\![\mathbb{G}]\!]_{DB}$ for $\mathbb{G}$ w. r. t. *DB* contains all matches for $\mathbb{G}$ in *DB*. Every BGP $\mathbb{G}$ can be seen as a graph $G(\mathbb{G}) = (V_{\mathbb{G}}, \Sigma, \mathbb{G})$ by taking the set of variables occurring in $\mathbb{G}$ as set of nodes, i. e., $V_{\mathbb{G}} := \{\mathtt{v}, \mathtt{w} \mid (\mathtt{v}, a, \mathtt{w}) \in \mathbb{G}\}$. The graph in Fig. 1(b) represents such a conversion of of query ($\mathcal{X}_1$).

For dual simulation processing of a BGP $\mathbb{G}$ w. r. t. *DB*, we compute the largest dual simulation between $G(\mathbb{G})$ and *DB*. This procedure is sound in that every match μ for $\mathbb{G}$ in *DB* is a dual simulation and therefore must be contained in the largest dual simulation.

Lemma 2 *Let DB be a graph database and $\mathbb{G}$ be a BGP. Each $\mu \in [\![\mathbb{G}]\!]_{DB}$ is a dual simulation between $G(\mathbb{G})$ and DB.*

The reason why Lemma 2 holds is that every match essentially constitutes a graph homomorphism. We provide the detailed proof in our technical report [19]. The nodes disqualified by the largest dual simulation are irrelevant for any further query processing, obeying the original SPARQL semantics.

Theorem 1 *Let DB be a graph database, $\mathbb{G}$ a BGP and S the largest dual simulation between $G(\mathbb{G})$ and DB. For each database node $o \in O_{DB}$ such that there are $\mathtt{v} \in vars(\mathbb{G})$ and $\mu \in [\![\mathbb{G}]\!]_{DB}$ with $\mu(\mathtt{v}) = o$, it holds that $(\mathtt{v}, o) \in S$.*

Unfortunately, the converse, i. e., irrelevant nodes for BGP result sets are ruled out by the largest dual simulation, does not hold in general. Consider the example graphs P and K depicted in Fig. 4(a) and (b). The largest dual simulation between P and K includes node p_4 which is, however, not belonging to any match for the respective BGP. The reason why p_4 must not be disqualified for variable/node $\mathtt{v}$ is that nodes p_1 and p_3 distribute the obligations for simulating variable/node $\mathtt{w}$. Informally, p_1 knows p_4 via p_2 and p_3, although p_1 and p_4 do not have a direct link to one another.

We compute the largest dual simulation by the largest solution of the SOI constructed from $G(\mathbb{G})$ (cf. Sect. III). From Theorem 1 we learn the desirable property for systems of inequalities $\mathcal{E}$ of any query $\mathcal{Q}$, that we must not remove nodes from the database important for any further processing of matches. We call this property *soundness of $\mathcal{E}$ w. r. t. $\mathcal{Q}$*.

Definition 2 Let *DB* be a graph database, $\mathcal{Q}$ a SPARQL query and $\mathcal{E}$ any SOI representation of $\mathcal{Q}$ with solutions $S \subseteq vars(\mathcal{Q}) \times O_{DB}$. $\mathcal{E}$ is *sound w. r. t.* $\mathcal{Q}$ iff for the largest solution S of $\mathcal{E}$, it holds that if $\mu(v) = o$ for some $v \in vars(\mathcal{Q})$ and $\mu \in [\![\mathcal{Q}]\!]_{DB}$, then $(v, o) \in S$. ■

B. Advanced Graph Patterns

BGPs, and SPARQL queries in general, may be combined by operators, further restricting and linking the sets of matches. This subsection is devoted to applying dual simulation principles to queries with UNION- and AND-operators. The AND-operator is best characterized by relational inner-joins of the results of two queries.

The UNION-operator is the least invasive operator. It combines any two queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ to query $\mathcal{Q}_1$ UNION $\mathcal{Q}_2$. The result set is the union of the result sets of the constituent queries, i. e., $[\![\mathcal{Q}_1 \text{ UNION } \mathcal{Q}_2]\!]_{DB} := [\![\mathcal{Q}_1]\!]_{DB} \cup [\![\mathcal{Q}_2]\!]_{DB}$. It is well-known that any SPARQL query may be rewritten as the union of finitely many *union-free* queries (cf. Proposition 3.8 [26]). A SPARQL query $\mathcal{Q}$ is *union-free* if the UNION-operator does not occur in $\mathcal{Q}$. Instead of $\mathcal{Q}$ we may process each union-free part of $\mathcal{Q}$ individually and later combine their results. Henceforth, we assume every query to be union-free.

While SPARQL's disjunction unifies the result sets of the constituents, conjunction unifies compatible results, i. e., those results agreeing upon shared variables. Matches μ_1 and μ_2 are *compatible*, denoted $\mu_1 \rightleftharpoons \mu_2$, if for all $v \in dom(\mu_1) \cap dom(\mu_2)$ (v shared by μ_1 and μ_2), $\mu_1(v) = \mu_2(v)$. The conjunction of two queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ is the query $\mathcal{Q}_1$ AND $\mathcal{Q}_2$. As an example, the SPARQL representation of the graph pattern in Fig. 4(a) may be described as the conjunction of two BGPs, $\mathbb{G}_1 = \{(\mathtt{v}, \mathtt{knows}, \mathtt{w})\}$ and $\mathbb{G}_2 = \{(\mathtt{w}, \mathtt{knows}, \mathtt{v})\}$. The semantics of conjunctions is defined by

$$[\![\mathcal{Q}_1 \text{ AND } \mathcal{Q}_2]\!]_{DB} := \{\mu_1 \cup \mu_2 \mid \mu_i \in [\![\mathcal{Q}_i]\!]_{DB} \wedge \mu_1 \rightleftharpoons \mu_2\}.$$

For example, in the database in Fig. 4(b), queries $\mathbb{G}_i$ from above enjoy matches μ_i ($i = 1, 2$) with $\mu_1(\mathtt{v}) = \mu_2(\mathtt{v}) = p_1$ and $\mu_1(\mathtt{w}) = \mu_2(\mathtt{w}) = p_2$. These matches are compatible, thus $(\mu_1 \cup \mu_2) \in [\![\mathbb{G}_1 \text{ AND } \mathbb{G}_2]\!]_{DB}$. In contrast, μ_1 from before and μ_3 with $\mu_3(\mathtt{w}) = p_2$ and $\mu_3(\mathtt{v}) = p_3$ constitute incompatible matches, thus $(\mu_1 \cup \mu_3) \notin [\![\mathbb{G}_1 \text{ AND } \mathbb{G}_2]\!]_{DB}$.

Regarding our dual simulation process, for conjunctions $\mathcal{Q}_1$ AND $\mathcal{Q}_2$, we create the systems of inequalities for $\mathcal{Q}_1$ and $\mathcal{Q}_2$ separately, denoted by $\mathcal{E}(\mathcal{Q}_1)$ and $\mathcal{E}(\mathcal{Q}_2)$. Recall that the variables of both queries directly refer to variables occurring in $\mathcal{Q}_1$ and $\mathcal{Q}_2$, respectively. The semantics of conjunctions requires matches to queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ to be compatible. In consequence, assignments to common variables must be identical. This may be achieved by simply unifying the systems of inequalities of both queries. The following lemma defines the sound system of inequalities.

Lemma 3 *Let DB be a graph database and $\mathcal{Q}_1, \mathcal{Q}_2$ queries with sound systems of inequalities $\mathcal{E}(\mathcal{Q}_1) = (\mathtt{Var}_1, \mathtt{Eq}_1)$ and $\mathcal{E}(\mathcal{Q}_2) = (\mathtt{Var}_2, \mathtt{Eq}_2)$. Then $\mathcal{E} = (\mathtt{Var}_1 \cup \mathtt{Var}_2, \mathtt{Eq}_1 \cup \mathtt{Eq}_2)$ is sound for $\mathcal{Q}_1$ AND $\mathcal{Q}_2$.*

A proof can again be found in our technical report [19]. The result itself is independent of the shape of $\mathcal{Q}_1$ and $\mathcal{Q}_2$, allowing them to also contain optional patterns, whose proper handling we describe subsequently.

C. Optional Patterns

The last syntactic construct of SPARQL for which we provide a sound dual simulation procedure is that of optional patterns. While, in terms of complexity, it is the most involved SPARQL operator [29], our procedure needs rather small adjustments. Reconsider our introductory query ($\mathcal{X}_1$), where we asked for directors and their coworkers. If we are not sure whether every director has a person listed they worked with, then we may put this information in an optional pattern, yielding query ($\mathcal{X}_2$).

```
SELECT * WHERE {
  ?director directed ?movie .
  OPTIONAL {
    ?director worked_with ?coworker . } }
```
($\mathcal{X}_2$)

Optional patterns are left-outer joins in the relational model, i. e., matches to ($\mathcal{X}_2$) definitely assign nodes from the database to variable `?director` and `?movie`, but to variable `?coworker` only if there is one. Regarding the graph database in Fig. 1(a), we obtain all bold subgraphs, as before, and additionally the semi-thick subgraphs (with `D. Koepp` and `T. Young` as `?director`). In general, for queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$, the result set of $\mathcal{Q}_1$ AND $\mathcal{Q}_2$ is contained in the result set of the optional pattern $\mathcal{Q}_1$ OPTIONAL $\mathcal{Q}_2$. Additionally, all matches to $\mathcal{Q}_1$ that have no compatible matches to $\mathcal{Q}_2$ are matches, i. e.,

$$[\![\mathcal{Q}_1 \text{ OPTIONAL } \mathcal{Q}_2]\!]_{DB} := [\![\mathcal{Q}_1 \text{ AND } \mathcal{Q}_2]\!]_{DB} \cup \{\mu \in [\![\mathcal{Q}_1]\!]_{DB} \mid \nexists \mu' \in [\![\mathcal{Q}_2]\!]_{DB} : \mu \rightleftharpoons \mu'\}.$$

In ($\mathcal{X}_2$), variable `?director` occurs in two different roles. First, the optional pattern mandates variable `?director` to feature triples with label `directed`. Second, triples labeled `worked_with` are only optional. These two roles must be reflected by our SOI representation of ($\mathcal{X}_2$) by including two copies of that variable, $\texttt{?director}_m$ (mandatory) and $\texttt{?director}_o$ (optional) with the property that a solution S in variable $\texttt{?director}_o$ must not exceed S in variable $\texttt{?director}_m$. In other words, there is no database node matching $\texttt{director}_o$ that does not match $\texttt{?director}_m$. This is expressed by inequality

$$\texttt{?director}_o \leq \texttt{?director}_m. \tag{13}$$

To faithfully describe such dependencies, we need to distinguish optional variable occurrences from mandatory ones, based the formal query syntax.

The query language $\mathcal{S}$ comprises union-free SPARQL queries with AND and OPTIONAL operators, as the following grammar describes:

$$\mathcal{Q} \quad ::= \quad \mathbb{G} \mid \mathcal{Q} \text{ AND } \mathcal{Q} \mid \mathcal{Q} \text{ OPTIONAL } \mathcal{Q}$$

where $\mathbb{G}$ ranges over by BGPs. Queries in $\mathcal{S}$ range over by $\mathcal{Q}, \mathcal{Q}_1, \mathcal{Q}_2, \ldots$. As observed above, we need to consider mandatory and optional variable occurrences. Function $mand$

maps queries $\mathcal{Q}$ from $\mathcal{S}$ to the set of variables that occur as mandatory in $\mathcal{Q}$, defined by

1) $mand(\mathbb{G}) := vars(\mathbb{G})$,
2) $mand(\mathcal{Q}_1 \text{ AND } \mathcal{Q}_2) := mand(\mathcal{Q}_1) \cup mand(\mathcal{Q}_2)$, and
3) $mand(\mathcal{Q}_1 \text{ OPTIONAL } \mathcal{Q}_2) := mand(\mathcal{Q}_1)$.

For handling optional pattern $\mathcal{Q}_1 \text{OPTIONAL}\, \mathcal{Q}_2$ correctly, we need to decide, in which cases an occurrence of variable $\mathtt{v}$ in $\mathcal{Q}_2$ has an optional dependency to another occurrence of the same variable. The case $\mathtt{v} \in vars(\mathcal{Q}_1)$ is reflected by query $(\mathcal{X}_2)$. Upon identification of such mandatory/optional pairs, we rename the optional occurrences of variables in our SOI and add an inequality as before, e. g., (13). More precisely, for the special case of query $\mathcal{Q} = \mathcal{Q}_1$ OPTIONAL $\mathcal{Q}_2$, we create the SOI representation for $\mathcal{Q}$ by first identifying mandatory/optional dependencies between $\mathcal{Q}_1$ and $\mathcal{Q}_2$, that are occurrences of variables $\mathtt{v} \in vars(\mathcal{Q}_2) \cap mand(\mathcal{Q}_1)$. For $\mathtt{v} \in vars(\mathcal{Q}_2) \cap mand(\mathcal{Q}_1)$, we reserve a unique name $\mathtt{v}_{\mathcal{Q}_2}$, which we use to replace $\mathtt{v}$ in every inequality of $\mathcal{Q}_2$, achieved by a renaming $\rho := \{(\mathtt{v}, \mathtt{v}_{\mathcal{Q}_2}) \mid \mathtt{v} \in vars(\mathcal{Q}_2) \cap mand(\mathcal{Q}_1)\}$. Upon renaming, we add inequality

$$\mathtt{v}_{\mathcal{Q}_2} \leq \mathtt{v} \tag{14}$$

for $\mathtt{v} \in vars(\mathcal{Q}_2) \cap mand(\mathcal{Q}_1)$ to the overall SOI. The largest solution to the resulting SOI consists of all assignments to the new variables $\mathtt{v}_{\mathcal{Q}_2}$, i. e., to variables not occurring in the original formulation of the query. Since these variables are only surrogates necessary for handling optionality correctly, and the largest solution for these variables is subsumed by the respective mandatory variables (cf. (14)), we may ignore them in the final result of the pruning step.

Lemma 4 *Let DB be a graph database and $\mathcal{Q}_1, \mathcal{Q}_2$ two* SPARQL *queries with sound systems of inequalities $\mathcal{E}(\mathcal{Q}_1) = (\mathtt{Var}_1, \mathtt{Eq}_1)$ and $\mathcal{E}(\mathcal{Q}_2) = (\mathtt{Var}_2, \mathtt{Eq}_2)$. Furthermore, define renaming as ρ by $\rho(\mathtt{v}) := \mathtt{v}_{\mathcal{Q}_2}$ for all $\mathtt{v} \in vars(\mathcal{Q}_2) \cap mand(\mathcal{Q}_1)$. Then*

$$\mathcal{E} \quad = \quad (\mathtt{Var}_1 \cup \mathtt{Var}_2 \cup \rho(\mathtt{Var}_2), \mathtt{Eq}_1 \cup \rho(\mathtt{Eq}_2) \cup \mathtt{Eq}_0)$$

with $\mathtt{Eq}_0 := \{\mathtt{v}_{\mathcal{Q}_2} \leq \mathtt{v} \mid \mathtt{v} \in vars(\mathcal{Q}_2) \cap mand(\mathcal{Q}_1)\}$ is sound for $\mathcal{Q}_1$ OPTIONAL $\mathcal{Q}_2$.

A more detailed description of variable dependencies in optional patterns is included in the technical report [19]. What if optional patterns occur within the clauses of a conjunction? Let us consider another example:

$$(\{(\mathtt{v}_1, a, \mathtt{v}_2)\} \text{ OPTIONAL } \{(\mathtt{v}_3, b, \mathtt{v}_2)\}) \text{ AND } \{(\mathtt{v}_3, c, \mathtt{v}_4)\}. \quad (\mathcal{X}_3)$$

The query consists of three triple patterns, the first two constitute an optional pattern and their results are joined with the third triple pattern. Fig. 5(b) and (c) show possible matches of $(\mathcal{X}_3)$ w. r. t. the graph database in Fig. 5(a). Analogous to $(\mathcal{X}_2)$, we derive $\mathtt{v}_{2m}$ and $\mathtt{v}_{2o}$ with $\mathtt{v}_{2o} \leq \mathtt{v}_{2m}$ from the optional pattern. The first occurrence of $\mathtt{v}_3$ is optional whilst the second occurrence is a mandatory one. Matches to $\mathtt{v}_3$ have an outgoing c-labeled edge and may feature the b-labeled edge from the optional pattern. Although both occurrences are not directly linked in an optional pattern, the second occurrence restricts the possible assignments for the first one. We make use of renamings based on a unique identification of subqueries. If we abstract $(\mathcal{X}_3)$ to $\mathcal{Q}_1$ AND $\mathcal{Q}_2$, then $\mathcal{Q}_1$ is R_1 OPTIONAL R_2. In the course of renaming, we replace $\mathtt{v}_3$ in R_2 by a fresh variable, e. g., ${\mathtt{v}_3}^{R_2}$, and add inequality ${\mathtt{v}_3}^{R_2} \leq \mathtt{v}_3$ to the system of inequalities for $(\mathcal{X}_3)$. Renaming functions ρ_i $(i = 1, 2)$ are defined accordingly to rename variables that occur only optional in $\mathcal{Q}_i$ but mandatory in the other subquery $\mathcal{Q}_j$. The solution of the resulting system of inequalities is interpreted as if all renamed variables are unified with their originals. This interpretation allows us to maintain correctness of lemmas 3 and 4 without cluttering the formal notation of the results.

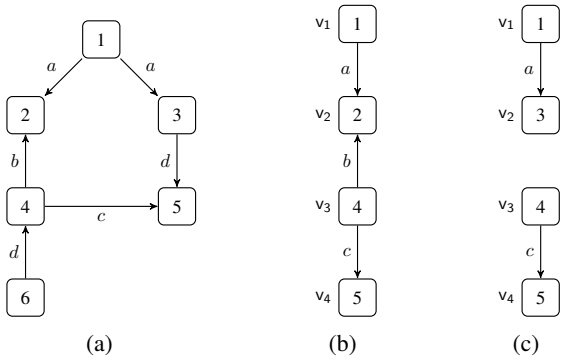

Fig. 5: (a) Graph Database, (b) and (c) Matches of $(\mathcal{X}_3)$

D. Discussion

Before we discuss an important query type, we conclude this section by showing that the constructed systems of inequalities are sound for any query $\mathcal{Q} \in \mathcal{S}$, using all the results we obtained so far in the proof.

Theorem 2 (Soundness) *Let DB be a graph database and $\mathcal{Q} \in \mathcal{S}$. Then $\mathcal{E}(\mathcal{Q})$ is a sound SOI.*

Our theoretical considerations are limited to SPARQL queries in which every node of a triple pattern is a variable. SPARQL also allows mentioning constants, i. e., objects and literals from the database, often drastically reducing the number of possible results. The key to integrating constant nodes into our pruning technique is to alter inequality (12).

Our dual simulation process is not restricted to well-designed patterns. Well-designed patterns are SPARQL queries $\mathcal{Q}$ with the property that for every sub-query $\mathcal{Q}_1$ OPTIONAL $\mathcal{Q}_2$ and every $\mathtt{v} \in vars(\mathcal{Q}_2)$ that also occurs outside the optional pattern also occurs in $\mathcal{Q}_1$, i. e., $\mathtt{v} \in vars(\mathcal{Q}_1)$ [26]. Query $(\mathcal{X}_3)$ is not well-designed, since $\mathtt{v}_3$ occurs as an optional variable but also outside the optional sub-pattern. Non-well-designed patterns give rise to cross-product results, as indicated by the match in Fig. 5(c). Assume that we have several c-labeled edges, then each of these edges together with the a-labeled edge forms an answer to the query. In these situations, our procedure remains effective, since it handles both occurrences of variable $\mathtt{v}_3$ separately. In fact,

the addition of AND and OPTIONAL operators does not influence the complexity of our procedure. Considering dual simulation as a query processor for $\mathcal{S}$, PSPACE-completeness of the evaluation problem [29] may be evaded, since checking whether a given relation S constitutes a valid assignment to $\mathcal{E}(Q)$ and extensions of it may be performed in PTIME. More expressive fragments of SPARQL add combinatorial complexity not solvable by pure dual simulation pattern matching.

V. EVALUATION

First, we compare our algorithm to the state-of-the-art dual simulation algorithm as introduced by Ma et al. [18] and used in implementations of [18], [23], [30] for evaluation purposes. Both are implemented within our prototype called SPARQLSIM. Second, we analyze how our SPARQL extension of dual simulation may be used to effectively and efficiently prune graph databases to improve query processing on an in-memory RDF database and a triple store based on relational database technology. After analyzing the effectiveness of the pruning, we compare query evaluation times with two graph database systems on two very large graph datasets comprising 750 million and 1.3 billion triples. We focus on time-consuming optional queries which were also used by Atre [4]. Details concerning the evaluation results, a list of queries, and our implementation can be found on our project's Github page.

A. Experimental Setup

For the first experiment, we have implemented the dual simulation algorithm of Ma et al. as an option in our tool. To evaluate our prototypes' performance as a pruning mechanism, we employed one of the fastest RDF databases Virtuoso [9] and the high-performance in-memory database RDFox [24]. All experiments have been performed on a server running Ubuntu 16.04 with four XEON E7-8837, 2.67 GHz, having 8 Cores each, 384 GB RAM and a Kingston DCP1000 NVMe PCI-E SSD. We deactivated caching for Virtuoso to achieve stable query evaluation times. RDFox is not using query caches. For the evaluation, we have run all queries 10 times on each database and averaged the times.

Since we provide a dual simulation algorithm that can be used as an external pruning mechanism, we imported the result sets from our tool into the two databases manually and then re-evaluated the queries on the pruning in comparison to queries on the full databases. Here, we did not consider the export time from our tool and the import time into the database, because our tool could easily be integrated into a standard database system, using our computations internally.

Our evaluation data comprises two popular RDF datasets: (1) The DBpedia dump 2016-10 in the English language version [6] and (2) the synthetic Lehigh University Benchmark [14] (LUBM) dataset generated for 10,000 universities. DBpedia comprises 751,603,507 triples with 216,132,665 nodes and 65,430 predicates. While the DBpedia queries $\mathcal{D}_0$-$\mathcal{D}_5$ stem from [4], benchmark queries $\mathcal{B}_0$-$\mathcal{B}_{19}$ appeared in the DBpedia benchmark dataset in [22]. The LUBM benchmark dataset comprises 1,381,692,508 triples with 18 predicates and 328,620,750 nodes. Since official query sets hardly cover optional patterns, we rely on queries that have been used by Atre [4] (cf. $\mathcal{L}_0$-$\mathcal{L}_5$).

TABLE II: Runtimes of our SPARQLSIM for BGPs from queries $\mathcal{B}_0$-$\mathcal{B}_{21}$ compared to Ma et al. [18].

Query	$t_{\text{SPARQLSIM}}$	$t_{\text{MA ET AL.}}$	Query	$t_{\text{SPARQLSIM}}$	$t_{\text{MA ET AL.}}$
$\mathcal{B}_0$	0.10385	6.72121	$\mathcal{B}_{10}$	0.02397	0.27126
$\mathcal{B}_1$	0.03876	3.33471	$\mathcal{B}_{11}$	0.01392	0.02099
$\mathcal{B}_2$	0.79097	3.84781	$\mathcal{B}_{12}$	0.01477	0.02287
$\mathcal{B}_3$	0.69797	5.62662	$\mathcal{B}_{13}$	0.35515	11.30355
$\mathcal{B}_4$	0.00003	0.00004	$\mathcal{B}_{14}$	5.46599	16.63957
$\mathcal{B}_5$	0.04091	0.31700	$\mathcal{B}_{15}$	13.43710	24.99660
$\mathcal{B}_6$	0.41105	0.54291	$\mathcal{B}_{16}$	0.00002	0.00003
$\mathcal{B}_7$	0.26991	0.51206	$\mathcal{B}_{17}$	1.12649	2.30390
$\mathcal{B}_8$	0.13562	5.51084	$\mathcal{B}_{18}$	0.32056	0.54057
$\mathcal{B}_9$	0.02551	0.08707	$\mathcal{B}_{19}$	0.69515	5.15070

The space our tool allocates for storing the adjacency matrices sums up to 35 GB for LUBM and 23 GB for DBpedia. The biggest matrices of LUBM consume between 1 GB and 4 GB of main memory (11 out of 36, e. g.,`rdf:type`). 99% of the DBpedia predicates allocate less than 1 MB. Constructing the adjacency matrices and producing the result triples requires additional space for storing maps and string objects.

B. Evaluation Analysis

a) Comparison of Dual Simulation Algorithms: Due to the fact that Ma et al.'s algorithm [18] considers BGPs as input we have removed the SPARQL keyword OPTIONAL from benchmark queries $\mathcal{B}_0$-$\mathcal{B}_{19}$. Evaluation times are shown in Table II. We observe that the optimizations allowed by SPARQLSIM (cf. Sect. III-C) pay off, since we outperform Ma et al.'s algorithm in every case, often even by an order of magnitude. When running in graph database query scenarios, it is this order of magnitude the naive algorithm lacks.

b) Dual Simulation as Pruning Mechanism: First, we analyze SPARQLSIM's pruning effectiveness (cf. Table III) of dual simulation for all LUBM and DBpedia queries. Observe that the number of triples is drastically decreased from the original databases for all queries. Over all tested queries we prune at least 95% of the original database. Hence, for most DBpedia queries we prune all triples not required for any result (compare req. triples and tripl. aft. pruning in Table III). In comparison, the effectiveness of our pruning is smaller for LUBM queries, being least effective for query $\mathcal{L}_1$. Later on we provide evidence that, e. g., for $\mathcal{L}_1$, our pruning allows the two database systems to enormously improve upon their evaluation times.

Regarding efficiency, SPARQLSIM's evaluation time heavily depends on the query and the dataset. With LUBM, having only 18 distinct predicates, we have an extreme case that often needs over 30 iterations to compute the largest dual simulation, which leads to high running times of our algorithm, e. g., for $\mathcal{L}_0$ or $\mathcal{L}_2$. As an outstanding characteristic, these two queries have a huge number of results. It is further a combination of the cyclic shape of the queries and the low selectivity of the predicates within the queries that explains the long runtime

TABLE III: Result set sizes, numbers of required triples, runtimes of SPARQLSIM in seconds and numbers of triples after pruning.

Query	Result No.	Req. Triples	$t_{\text{SPARQLSIM}}$	Tripl. aft. Pruning
$\mathcal{L}_0$	10,448,905	3,276,841	106.451	10,181,730
$\mathcal{L}_1$	226,641	114,989	8.464	25,429,750
$\mathcal{L}_2$	32,828,280	15,416,012	147.335	48,674,046
$\mathcal{L}_3$	11	35	0.138	126
$\mathcal{L}_4$	10	33	0.125	101
$\mathcal{L}_5$	7	35	1.220	35
$\mathcal{D}_0$	523,066	3,139,273	4.396	3,141,102
$\mathcal{D}_1$	0	0	0.002	0
$\mathcal{D}_2$	12	60	0.088	60
$\mathcal{D}_3$	5794	28,704	0.143	28,704
$\mathcal{D}_4$	25,102,459	22,630,477	6.230	22,691,521
$\mathcal{D}_5$	365,693	79,943	0.574	79,944
$\mathcal{B}_0$	12	60	0.088	60
$\mathcal{B}_1$	859,751	726,749	0.022	726,812
$\mathcal{B}_2$	913,786	1,587,731	0.532	1,588,127
$\mathcal{B}_3$	438,542	386,000	0.606	386,020
$\mathcal{B}_4$	0	0	0.000	0
$\mathcal{B}_5$	0	0	0.033	0
$\mathcal{B}_6$	815,522	886,826	0.503	886,939
$\mathcal{B}_7$	34,991	37,965	0.443	37,965
$\mathcal{B}_8$	8416	30,258	0.113	30,258
$\mathcal{B}_9$	8247	13,116	0.022	13,116
$\mathcal{B}_{10}$	8061	12,642	0.027	12,642
$\mathcal{B}_{11}$	9849	8955	0.018	8955
$\mathcal{B}_{12}$	9554	8660	0.018	8660
$\mathcal{B}_{13}$	123,467	365,131	0.273	365,154
$\mathcal{B}_{14}$	22,673,220	27,652,055	4.322	27,747,192
$\mathcal{B}_{15}$	0	0	0.000	0
$\mathcal{B}_{16}$	2	4	0.009	4
$\mathcal{B}_{17}$	7,898,331	8,285,964	0.917	8,294,385
$\mathcal{B}_{18}$	66,903	41,808	0.472	41,808
$\mathcal{B}_{19}$	879,460	292,531	0.602	292,541

TABLE IV: Query processing times on the full and pruned dataset, and query times including pruning times for RDFox. All times are measured in seconds.

Query	t_{DB}	$t_{\text{DB pruned}}$	$t_{\text{DB pruned}} + t_{\text{SPARQLSIM}}$
$\mathcal{L}_0$	19.100	1.401	107.852
$\mathcal{L}_1$	25,900.000	888.000	896.464
$\mathcal{L}_2$	161.000	15.690	163.025
$\mathcal{L}_3$	0.000	0.000	0.138
$\mathcal{L}_4$	0.000	0.000	0.125
$\mathcal{L}_5$	0.000	0.000	1.223
$\mathcal{D}_0$	1.400	1.115	5.511
$\mathcal{D}_1$	0.000	0.000	0.002
$\mathcal{D}_2$	1.100	0.003	0.091
$\mathcal{D}_3$	0.620	0.002	0.145
$\mathcal{D}_4$	5.960	3.493	9.722
$\mathcal{D}_5$	3.230	0.016	0.590
$\mathcal{B}_0$	1.468	0.000	0.088
$\mathcal{B}_1$	0.099	0.030	0.052
$\mathcal{B}_2$	0.348	0.110	0.642
$\mathcal{B}_3$	0.104	0.012	0.618
$\mathcal{B}_4$	0.033	0.000	0.000
$\mathcal{B}_5$	0.000	0.000	0.033
$\mathcal{B}_6$	12.830	0.042	0.545
$\mathcal{B}_7$	14.410	0.002	0.445
$\mathcal{B}_8$	0.793	0.001	0.114
$\mathcal{B}_9$	0.117	0.001	0.023
$\mathcal{B}_{10}$	0.004	0.001	0.028
$\mathcal{B}_{11}$	0.001	0.000	0.018
$\mathcal{B}_{12}$	0.001	0.001	0.019
$\mathcal{B}_{13}$	0.643	0.022	0.295
$\mathcal{B}_{14}$	3.282	1.998	6.320
$\mathcal{B}_{15}$	0.941	0.000	0.000
$\mathcal{B}_{16}$	0.000	0.000	0.009
$\mathcal{B}_{17}$	0.758	0.310	1.227
$\mathcal{B}_{18}$	0.119	0.001	0.473
$\mathcal{B}_{19}$	18.750	0.048	0.650

of our algorithm. In DBpedia, predicates usually have a much higher selectivity. Hence, we usually perform the computation for these queries in only a split-second.

c) Runtime of RDF Databases: By the next experiments we compare the query evaluation time of the in-memory database RDFox to SPARQLSIM in combination with RDFox as a query processor. In Table IV, we observe an improvement of the query time in 15 out of 32 queries. Especially the improvement on query $\mathcal{L}_1$ with a query processing time of 25,900 seconds on RDFox is worth mentioning. We could run our dual simulation algorithm in only 8 seconds (cf. Table III), decreasing the query time of RDFox by more than 20 times. For $\mathcal{L}_0$, however, $t_{\text{SPARQLSIM}}$ alone is around 5 times slower than RDFox (t_{DB}). Also, in queries $\mathcal{D}_5$, $\mathcal{B}_0$, $\mathcal{B}_7$-$\mathcal{B}_9$, $\mathcal{B}_{17}$, $\mathcal{B}_{21}$ we show good improvements of the in-memory databases' evaluation times. For most of the remaining queries we show comparable results to RDFox, varying by some milliseconds.

Table V shows an improvement of the running times of only 3 queries for Virtuoso. For most other queries, evaluation times are on par with t_{DB}. For some queries, our pruning could not increase Virtuoso's evaluation time as much as for RDFox. A detailed analysis of Virtuoso's query plans revealed that this was due to changes in the join order that sometimes seems to turn against optimal evaluation times by drastically increasing the number of intermediate results, e. g., $\mathcal{D}_4$ with doubled evaluation time $t_{\text{DB pruned}}$ on the 3% portion of DBpedia. Nevertheless, we believe that Virtuoso could benefit from a direct integration of SPARQLSIM as a pruning technique. In turn, our tool may advance by employing Virtuoso's built-in heuristics for query evaluation plans. On the downside, our algorithm is often slightly slower than the professionally implemented and highly optimized RDF triple store. Some of the more complex queries took longer to produce the pruning than for Virtuoso to produce the actual answers. These queries took several iterations in SPARQLSIM to compute. We believe that we can benefit from more sophisticated join order optimization techniques as used for example in Virtuoso which could boost our computation times tremendously. The very fast pruning time for the cyclic query $\mathcal{L}_1$ requires only two iterations, and thereby points to the potential of our solution.

C. Discussion

The evaluation results suggest dual simulation pruning as an effective technique allowing two state-of-the-art graph database systems to improve upon their query evaluation times, sometimes enormously. Preprocessing $\mathcal{L}_1$ is most profitable, since huge intermediate tables can be avoided. In this case we observe a decrease by more than one order of magnitude

TABLE V: Query processing times on the full and pruned dataset, and query times including pruning times for Virtuoso. All times are measured in seconds.

Query	t_{DB}	$t_{\text{DB pruned}}$	$t_{\text{DB pruned}} + t_{\text{SPARQLSIM}}$
$\mathcal{L}_0$	5.126	2.261	108.712
$\mathcal{L}_1$	50.853	0.971	9.435
$\mathcal{L}_2$	56.676	26.767	174.102
$\mathcal{L}_3$	0.001	0.000	0.138
$\mathcal{L}_4$	0.000	0.000	0.125
$\mathcal{L}_5$	0.000	0.000	1.223
$\mathcal{D}_0$	0.395	0.359	4.755
$\mathcal{D}_1$	0.001	0.000	0.002
$\mathcal{D}_2$	0.002	0.000	0.089
$\mathcal{D}_3$	0.010	0.003	0.147
$\mathcal{D}_4$	2.148	4.008	10.238
$\mathcal{D}_5$	0.039	0.021	0.595
$\mathcal{B}_0$	0.002	0.000	0.088
$\mathcal{B}_1$	0.003	0.001	0.023
$\mathcal{B}_2$	0.003	0.003	0.030
$\mathcal{B}_3$	0.001	0.002	0.020
$\mathcal{B}_4$	0.001	0.002	0.020
$\mathcal{B}_5$	0.054	0.031	0.303
$\mathcal{B}_6$	1.082	0.441	4.762
$\mathcal{B}_7$	0.000	0.000	0.000
$\mathcal{B}_8$	0.000	0.000	0.009
$\mathcal{B}_9$	0.121	0.099	1.016
$\mathcal{B}_{10}$	0.043	0.009	0.031
$\mathcal{B}_{11}$	0.012	0.003	0.476
$\mathcal{B}_{12}$	0.102	0.056	0.658
$\mathcal{B}_{13}$	0.069	0.064	0.596
$\mathcal{B}_{14}$	0.000	0.000	0.000
$\mathcal{B}_{15}$	0.000	0.000	0.034
$\mathcal{B}_{16}$	0.042	0.026	0.594
$\mathcal{B}_{17}$	0.022	0.013	0.516
$\mathcal{B}_{18}$	0.003	0.001	0.444
$\mathcal{B}_{19}$	0.021	0.005	0.118

while the pruning time is vastly fast in only two iterations. In contrast, because intermediate results in the evaluation of $\mathcal{L}_0$ are rather small, the benefits of dual simulation pruning are not as significant as for $\mathcal{L}_1$. Furthermore, the low selectivity predicates of $\mathcal{L}_0$ result in a rather big number of iterations that increases the pruning time compared to e. g., $\mathcal{L}_1$. As a general rule we recommend using dual simulation for pruning in cases where queries produce large intermediate results. Such cases can usually be detected employing database statistics for join result size estimation, also used for join order optimization. Our technical report [19] contains more details of $\mathcal{L}_0$ and $\mathcal{L}_1$. Remarkably, the complexity of both queries may not be found in the optional patterns but in their cyclic shape.

VI. Related Work

Recently, graph pattern matching has become a trending topic for graph databases, different from the canonical though costly prime candidate of graph isomorphism, with the goal of reducing structural requirements of the answer graphs. Especially, simulations have been implemented for different graph database tasks [8], [10], [12], [23]. Ma et al. [18] introduce the notion of *dual simulation*. Having a simulation preorder in a database context considering forward and backward edges is mentioned as early as in the year 2000 [1]. On the downside, performance improvements by dual simulation come with a loss of topology [18].

Mottin et al. [23] build on simulation as one solution to their query paradigm called *Exemplar Queries*. For a given exemplar graph pattern, the user obtains subgraphs from the database similar to the exemplar. We foresee that exemplar queries as well as other applications of graph pattern matching may exhibit the portion of SPARQL integrated in our framework, making their proposals even more attractive to users.

Using simulation for graph database pruning has been proposed as a component in Panda [30]. In Panda, subgraph simulation is used to filter unnecessary tuples before answering isomorphism queries. Their large-scale evaluation shows improvements in query time compared to several other isomorphism-based query processors. In contrast, we rely on dual simulation being more effective in pruning unnecessary triples, and we implement a fast dual simulation algorithm operating on bit-matrices which are particularly useful for large graph databases. Furthermore, we use a more expressive query model that could also be integrated into their pruning technique to support more complex queries. Other existing approaches for optimizing graph database querying rely on adapting traditional database optimization techniques, usually leading to major improvements with regard to the query performance [7], [9]. However, graph database queries usually consist of numerous joins with oftentimes huge intermediate results, requiring specialized optimization techniques. Therefore, join order estimation for graph databases, especially RDF triple stores, is still an active field [4], [17], [25], [29]. Our proposal appreciates the graph data model and performs lightweight algorithms to support traditional database optimization.

Simulation-based indexing techniques have already been used for join-ahead pruning in databases on XML data [21]. The index is created by computing bisimulation equivalence classes of nodes on the original database. Each equivalence class groups structurally bisimilar nodes [27], [31]. Bisimulation is more restrictive than dual simulation which we use throughout this paper. However, our algorithm could benefit from similar ideas. It would be sufficient to produce dual simulation equivalence classes, which promises to obtain a much smaller database fingerprint than possible with bisimulations, since (dual) simulation equivalence is coarser than bisimulation.

VII. Conclusion

We proposed efficient processing of SPARQL queries based on graph pattern matching. Our algorithm builds upon dual simulation and for all extensions, due to SPARQL, we provided soundness proofs. To derive an algorithm competing with state-of-the-art graph databases we contribute an alternative characterization of dual simulation in terms of a system of inequalities. Dual simulation is directly applicable to SPARQL's BGPs, whereas composite queries including AND and OPTIONAL operators, are handled by conservative extensions of dual simulation.

We could show that our algorithm outperforms standard dual simulation algorithms on a variety of real-world SPARQL BGPs. Furthermore, our dual simulation algorithm can be used to aggressively prune triples, speeding up graph database query processing for state-of-the-art graph databases. In comparison to these graph databases, we could improve the query evaluation time for several queries drastically and showed comparable results for the others. We believe that most database systems would benefit from a direct integration of our proposal into their query processor. Further applications already using dual simulation may benefit from our SPARQL extension to offer more expressive query capabilities.

We plan to extend our prototype by applying more heuristics with which we conduct extensive experiments to find better guidelines for the applicability of dual simulation pruning. Our experiments with two state-of-the-art graph database systems showed that such guidelines make sense on a per-system and per-data basis. We are currently investigating the limits of our dual simulation procedure w. r. t. different SPARQL fragments. While this work suggests a tremendous enhancement of the complexity of optional pattern evaluation, other operators add combinatorial problems unavoidable for a dual simulation evaluation semantics for SPARQL.

REFERENCES

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[2] Marcelo Arenas, Claudio Gutierrez, Daniel P. Miranker, Jorge Pérez, and Juan F. Sequeda. Querying semantic data on the web? *SIGMOD Rec.*, 41(4):6–17, January 2013.

[3] Marcelo Arenas and Martin Ugarte. Designing a query language for rdf: Marrying open and closed worlds. *ACM Trans. Database Syst.*, 42(4):21:1–21:46, October 2017.

[4] Medha Atre. Left bit right: For sparql join queries with optional patterns (left-outer-joins). In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1793–1808, New York, NY, USA, 2015. ACM.

[5] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.

[6] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The Semantic Web*, pages 722–735, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 121–132, New York, NY, USA, 2013. ACM.

[8] Joel Brynielsson, Johanna Hogberg, Lisa Kaati, Christian Martenson, and Pontus Svenson. Detecting social positions using simulation. In *Proceedings of the 2010 International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '10, pages 48–55, Washington, DC, USA, 2010. IEEE Computer Society.

[9] Orri Erling and Ivan Mikhailov. *RDF Support in the Virtuoso DBMS*, pages 7–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[10] Wenfei Fan. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 8–21, New York, NY, USA, 2012. ACM.

[11] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. Keys for graphs. *Proc. VLDB Endow.*, 8(12):1590–1601, August 2015.

[12] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *Proc. VLDB Endow.*, 3(1-2):264–275, September 2010.

[13] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. *Proc. VLDB Endow.*, 3(1-2):1161–1172, September 2010.

[14] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158 – 182, 2005. Selcted Papers from the International Semantic Web Conference, 2004.

[15] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 453–, Washington, DC, USA, 1995. IEEE Computer Society.

[16] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the 39th international conference on Very Large Data Bases*, pages 133–144. VLDB Endowment, 2013.

[17] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. Static analysis and optimization of semantic web queries. *ACM Trans. Database Syst.*, 38(4):25:1–25:45, December 2013.

[18] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.*, 39(1):4:1–4:46, January 2014.

[19] Stephan Mennicke, Jan-Christoph Kalo, Denis Nagel, Hermann Kroll, and Wolf-Tilo Balke. Fast dual simulation processing of graph database queries (supplement). *CoRR*, abs/1810.09355, 2018.

[20] Stephan Mennicke, Denis Nagel, Jan-Christoph Kalo, Niklas Aumann, and Wolf-Tilo Balke. Reconstructing graph pattern matches using SPARQL. In *(LWDA) Conference Proceedings, Rostock, Germany, September 11-13, 2017.*, page 152, 2017.

[21] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 277–295, London, UK, UK, 1999. Springer-Verlag.

[22] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. Dbpedia sparql benchmark – performance assessment with real queries on real data. In *The Semantic Web – ISWC 2011*, pages 454–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[23] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: A new way of searching. *The VLDB Journal*, 25(6):741–765, December 2016.

[24] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. *RDFox: A Highly-Scalable RDF Store*, pages 3–20. Springer International Publishing, 2015.

[25] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM.

[26] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[27] François Picalausa, Yongming Luo, George H. L. Fletcher, Jan Hidders, and Stijn Vansummeren. A structural approach to indexing triples. In *Proceedings of the 9th International Conference on The Semantic Web: Research and Applications*, ESWC'12, pages 406–421, Berlin, Heidelberg, 2012. Springer-Verlag.

[28] Eric Prud'hommeaux and Seaborne, Andy. SPARQL Query Language for RDF. Technical report, W3C, 2008.

[29] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[30] Miao Xie, Sourav S. Bhowmick, Gao Cong, and Qing Wang. Panda: toward partial topology-based search on large networks in a single machine. *The VLDB Journal*, 26(2):203–228, Apr 2017.

[31] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gstore: Answering sparql queries via subgraph matching. *Proc. VLDB Endow.*, 4(8):482–493, May 2011.