Computing Dual Simulation for Graph Database Queries
Technical Report

Stephan Mennicke           Jan-Christoph Kalo
Denis Nagel                Hermann Kroll
Wolf-Tilo Balke
Institut für Informationssysteme, TU Braunschweig, Braunschweig, Germany
\{mennicke,kalo,kroll,balke\}@ifis.cs.tu-bs.de
denis.nagel@tu-bs.de

1 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 [29], 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 [9, 25, 5].

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

\[
\begin{align*}
\text{SELECT } \ast & \text{ WHERE } \{ \\
& \text{ ?director directed ?movie . } \quad (\mathcal{Z}_1) \\
& \text{ ?director worked_with ?coworker . } \}
\end{align*}
\]

($\mathcal{Z}_1$) consists of two triple patterns, $t_1$ and $t_2$. The first requires a directed link between assignments to variables ?director and ?movie while $t_2$ asks for ?director to be in a worked_with relationship with an object matching ?coworker. An evaluation of ($\mathcal{Z}_1$) 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, simple graph pattern matching for diverse querying tasks raised a growing interest in the database community [8, 13, 12, 10, 18, 20, 11, 24, 31]. Many of these applications employ a form of simulation graph pattern matching, showing computational advantages over isomorphic matching. Yet, an in-depth analysis of the approaches incorporating simulation [8, 20, 24, 31] 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 state-of-the-art 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 \cite{21}). 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 \cite{21}.

(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 like in 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 \cite{20}. 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 2). The duality in dual simulation allows for an active computation that may find many violations of the definition in a single iteration. Based on a novel characterization of dual simulation in Sect. 3, 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 graph query settings. We provide formal proof of the correctness of our algorithm as well as experimental justification for the performance improvements brought by our solution. And what is more, our algorithm is also applicable to highly compressed database formats, as e.g., the BitMat storage structure \cite{4}, and to massive parallelization techniques of bit-matrix operations.

Regarding (2), we also contribute a conservative extension of dual simulation to work with typical graph query operators, exemplarily taken from SPARQL (cf. Sect. 4). 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 in the SOI solution and do not influence the overall polynomial-time complexity. We do not only deal with well-designed patterns \cite{27, 4}. Although, well-designed patterns have been of special interest, recent studies show that non-well-designed patterns cannot be neglected, since they form a sizeable portion of practical query loads \cite{16}. Therefore, we may expect usage of non-well-designed patterns in the above-mentioned applications. The advantage of our extended dual simulation process is that is does not distinguish non-well-designed patterns from well-designed ones.

In Sect. 5 we perform extensive experiments on two real-world 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. By dual simulation processing, more than 95% irrelevant triples are disqualified 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 [25]. Moreover, we observe that our dual simulation may directly be incorporated as a pruning preprocessing step in RDFox. In Sect. [6] we elaborate on related work while we draw a conclusion in Sect. [7].

2 Graphs, Data and Matching

By graphs we refer to edge-labeled directed graphs with a finite set of nodes $V$, a finite (label) alphabet $\Sigma$, 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 will be 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 $E_i$ ($i \in \mathbb{N}$). For simplicity, we assume all graphs to be labeled over a fixed alphabet $\Sigma$. For every label $a \in \Sigma$, we associate with $G$ two adjacency maps, a forward map $\mathcal{F}^a_G$ and a backward map of $G$, $\mathcal{B}^a_G$. 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.,

$$\mathcal{F}^a_G(v) := \{w \mid (v,a,w) \in E\}$$

and

$$\mathcal{B}^a_G(v) := \{u \mid (u,a,v) \in E\}.$$  

In the Resource Description Framework (RDF), the basic ingredients are triples $(s,p,o)$, describing a relationship $(p)$ between two database resources $s$ and $o$. By analogy, $s$, $p$ and $o$ are thought of as subject, predicate and object. Database resources $(s$ or $o$) stem from two universes, $\mathcal{O}$, the set of all objects, each usually referenced by an IRI (Internationalized Resource Identifier) and $\mathcal{L}$, the set of literals. A literal is an element from an arbitrary data domain, such as the integers, usually to describe attribute values of objects. Predicates are also implemented by IRIs, stemming from the universe $\mathcal{P}$. Simplifying the presentation, we assume all three universes to be disjoint. Furthermore, we abstract from the implementation as IRIs and use intuitive names to identify database objects and predicates (cf. example database in Fig. [1(a)]). RDF allows for generalized triples of type $\mathcal{O} \times \mathcal{P} \times (\mathcal{O} \cup \mathcal{L})$, sufficient to formulate interrelations and attributes of objects. Attributes connect an object with a literal, e.g., in Fig. [1(a)], the information that Saint John has 70,063 inhabitants is reflected by the triple (Saint John, population, 70,063). Further note that literals may only occur in the third component of a triple.

A graph database is a finite instance of all possible triples. We formalize it as a graph with all objects and literals occurring in triples as the set of nodes, and all predicates as the alphabet. Handling literals properly leads to a divergence from our initial model.

**Definition 1 (Graph Database)** A graph database is a graph $DB = (O_{DB}, \Sigma, E_{DB})$ with a finite set of database objects and literals $O_{DB} \subseteq \mathcal{O} \cup \mathcal{L}$, a finite set of properties $\Sigma \subseteq \mathcal{P}$, and a labeled edge relation $E_{DB} \subseteq (O_{DB} \cap \mathcal{O}) \times \Sigma \times O_{DB}$. All notions for graphs carry over to graph databases.

A dual simulation [20] 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 $v_2$ belong to $S$. For a dual simulation $S$, $(v_1, v_2) \in S$ means that $v_2$ dual simulates $v_1$. As
and the nodes with the same label, e. g., \( G \) is a dual simulation between \( G \) and \( G' \). 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 \( [20] \). Reconsider the introductory example query (\( \mathcal{Q}_1 \)). The graph in Fig. 2(b) dual simulates the graph representation of (\( \mathcal{Q}_1 \)) in Fig. 1(b). A dual simulation is realized by ignoring node place. Hence, not every node of 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 \( G_2 \), as in

\[
\{ \text{place, director1, director}, \text{director2, director}, \text{movie, movie}, \text{coworker, coworker} \} \tag{1}
\]

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 2 (Dual Simulation [20])**. 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 \( G_2 \). \( \blacksquare \)

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 \( [20] \). Reconsider the introductory example query (\( \mathcal{Q}_1 \)). The graph in Fig. 2(b) dual simulates the graph representation of (\( \mathcal{Q}_1 \)) in Fig. 1(b). A dual simulation is realized by ignoring node place. Hence, not every node of

<table>
<thead>
<tr>
<th>Table 1: Summary of Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G = (V, \Sigma, E) )</td>
</tr>
<tr>
<td>( \delta^a_G, \beta^a_G )</td>
</tr>
<tr>
<td>( DB = (O_{DB}, \Sigma, E_{DB}) )</td>
</tr>
<tr>
<td>( \chi_S : V_1 \to 2^{V_2} )</td>
</tr>
<tr>
<td>( \mathcal{Q}, \mathcal{Q}_1, \mathcal{Q}_2 )</td>
</tr>
<tr>
<td>( [\mathcal{Q}]_{DB} )</td>
</tr>
<tr>
<td>( \mu : \vars(\mathcal{Q}) \to O_{DB} )</td>
</tr>
<tr>
<td>( \mu_1 = \mu_2 )</td>
</tr>
<tr>
<td>( \delta = (\Var, \Eq) )</td>
</tr>
</tbody>
</table>

![Figure 2: Two Graph Patterns](image-url)
the graph database has to participate in the 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 $(X_1)$ in Fig. 1(b), dual simulation (2) turn out to be particularly useful in the upcoming sections.

{(director: B. De Palma), (director: G. Hamilton), (coworker: D. Koepp), (coworker: H. Saltzman), (movie: Mission: Impossible), (movie: Goldfinger)}

It comprises exactly the nodes of the two subgraphs from the result set of $(X_1)$. 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 [20], as opposed to SPARQL query evaluation being PSPACE-complete [27]. How to perform this computation fast is subject to the next section. We apply dual simulation principles to SPARQL query processing in Sect. 4.

3 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 $(X_1)$) and a graph database (Fig. 1(a)), covering all nodes relevant for computing the result set of $(X_1)$. In Sect. 4, we show that the existence of such a dual simulation is not coincidental, since every match for SPARQL queries like $(X_1)$ 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 [20])** For any two graphs $G_1$ and $G_2$, there is a unique largest dual simulation $S_{\text{max}}$ between $G_1$ and $G_2$, i.e., for any dual simulation $S$ between $G_1$ and $G_2$, $S \subseteq S_{\text{max}}$.

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 [20, 17] 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. 2. The procedures are guaranteed to terminate when no pair of nodes can be disqualified anymore. Although the standard algorithms share an $O(|V_2|^3)$ 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 [17] and [20] to edge-labeled graphs (cf. Sect. 3.3 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. 5).

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. 3.3), 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. 3.3). Since queries are usually assumed to be much smaller than the database, the organizational overhead in memory is negligible. As we show in Sect. 5 the new procedure shows extremely low computation times, a solid basis for query processing, e.g., for SPARQL. Our solution is engineered in three steps. First, we define a set of inequalities equivalent to the coinductive definition of dual simulation in Def. 2. In Sect. 3.2, we 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.

3.1 Groundwork

This subsection lays out the foundation of our new procedure. Any binary relation \( R \subseteq A \times B \), over sets \( A \) and \( B \), has a characteristic function \( \chi_R : A \rightarrow 2^B \) with \( \chi_R(a) := \{ b \in B \mid (a, b) \in R \} \). For a dual simulation \( S \) between two graphs \( G_1 \) and \( G_2 \), \( \chi_S \) 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 \( \chi_S \) 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 the graph database \( G_2 \), we have a forward adjacency map \( \overline{\delta}^a \) for each label \( a \in \Sigma \) (cf. Sect. 2). Exploiting these maps, we prove existence of a \( w' \) in (3) simply by intersecting the row of \( v' \) in \( \overline{\delta}^a \) and the nodes simulating \( w \), i.e.,
\[
\overline{\delta}^a_{G_2}(v') \cap \chi_S(w) \neq \emptyset. \tag{4}
\]
(4) still only checks for one pair of nodes. Combining this equation for all \( v' \in \chi_S(v) \) yields
\[
\bigwedge_{v' \in \chi_S(v)} \overline{\delta}^a_{G_2}(v') \cap \chi_S(w) \neq \emptyset. \tag{5}
\]
The same encoding applies to Def. 2(ii), this time using the backward map,
\[
\bigwedge_{w' \in \chi_S(w)} \overline{\delta}^a_{G_2}(w') \cap \chi_S(v) \neq \emptyset. \tag{6}
\]
Combining 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.

\[
\begin{align*}
(i) & \quad \chi_S(w) \subseteq \bigcup_{v' \in \chi_S(v)} \overline{\delta}^a_{G_2}(v') \\
(ii) & \quad \chi_S(v) \subseteq \bigcup_{w' \in \chi_S(w)} \overline{\delta}^a_{G_2}(w')
\end{align*}
\tag{7}
\]

**Proof:** W.l.o.g., we show inequality (i) only. Inequality (ii) is completely analogous. Towards a contradiction, assume \( \chi_S(w) \nsubseteq \bigcup_{v' \in \chi_S(v)} \overline{\delta}^a_{G_2}(v') \). Hence, there is a \( w' \in \chi_S(w) \) such that for each \( v' \in \chi_S(v) \), \( w' \notin \overline{\delta}^a_{G_2}(v') \), i.e., \((v', a, w') \notin E_2\). As a consequence, \( \chi_S(v) \) and \( \overline{\delta}^a_{G_2}(w') \) 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 \( \chi_S(w) \subseteq \bigcup_{v' \in \chi_S(v)} \overline{\delta}^a_{G_2}(v') \).

\[1\] available at GitHub https://github.com/ifis-tu-bs/sparqlSim
Phrased differently, dual simulations \( S \) satisfy (7) for every edge \((v, a, w)\) of \( G_1\). Please note 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 \( S \).

**Proof:** The implication, i.e., a dual simulation \( S \) satisfies (7), is analogous to the proof of Lemma 1. Therefore, assume that one of the inequalities is not satisfied and conclude the assumption that \( S \) is a dual simulation is violated.

Conversely, assume we have \( S \subseteq V_1 \times V_2 \) such that (7) holds for every \((v, a, w)\) \( \in E_1 \). We need to show that \( S \) is a dual simulation. Let \((v, v')\) \( \in S \), i.e., \( v' \in \chi_S(v) \), and \((v, a, w)\) \( \in E_1 \). We need to show that there is a \( w' \) such that \((v', a, w')\) \( \in E_2 \) and \((w, w')\) \( \in S \). From (7)(ii) we get that for some \( w' \in \chi_S(w) \) we have that \( v' \in \mathcal{B}^a_{G_2}(w') \). This \( w' \) completes the proof, since (1) from \( v' \in \mathcal{B}^a_{G_2}(w') \) follows \((v', a, w')\) \( \in E_2 \) and (2) from \( w' \in \chi_S(w) \), we get that \((w, w')\) \( \in S \). Case \((u, a, v)\) \( \in E_1 \) is completely analogous. \( \Box \)

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 \( \chi_{S_1}(w) := \chi_{S_0}(w) \cap \bigcup_{v' \in \chi_{S_0}(v)} \mathcal{B}^a_{G_2}(v') \). 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 \( \text{PTIME} \) nature of other algorithms (cf. Sect. 3.3), we still miss a way to quickly compute \( \bigcup_{v' \in \chi_S(v)} \mathcal{B}^a_{G_2}(v') \) and access \( \chi_S(v) \). Therefore, the forthcoming implementation works with bit-representations of \( \chi_S(v) \) and \( \mathcal{B}^a_{G_2}, \mathcal{B}^a_{G_2} \), paving the way for optimization in time- and space-consumption (e.g., \( \mathcal{O} \)). In that setting, we derive a *system of inequalities* (SOI) from Prop. 2 for which dual simulations \( S \) serve as valid assignments.

### 3.2 Engineering

The goal of this subsection is to describe the facilities achieving a fast implementation of dual simulation pruning. 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

\[
\bigcup_{v' \in \chi_S(v)} \mathcal{B}^a_{G_2}(v') \quad \text{and} \quad \bigcup_{w' \in \chi_S(w)} \mathcal{B}^a_{G_2}(w').
\]

Combining vectors and matrices, especially 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,

\[
\mathcal{B}^{\text{born_in}}_{G_2} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 
\end{bmatrix} \quad \text{and} \quad \mathcal{B}^{\text{born_in}}_{G_2} = \begin{bmatrix}
0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 
\end{bmatrix}.
\]

Here, we assume the set of nodes of graphs to be ordered by some pre-defined index, i.e., \( \{v_1, v_2, v_3, v_4, v_5\} \) with \( v_1 = \text{place}, v_2 = \text{director1}, v_3 = \text{director2}, v_4 = \text{coworker}, \) and \( v_5 = \text{movie} \). Also, \( \chi_S \) 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 $\chi_S$. The desired unions \footnote{For vector $v$ and matrix $A$, $v \times_b A = w$ where $w(j) = 1$ iff there is an $i$ such that $v(i) = 1$ and $A(i, j) = 1$.} are now achieved by bit-matrix multiplications\footnote{For vector $v$ and matrix $A$, $v \times_b A = w$ where $w(j) = 1$ iff there is an $i$ such that $v(i) = 1$ and $A(i, j) = 1$.} (symbol $\times_b$),

$$\chi_S(v) \times_b \mathcal{D}_{G_2}^a \quad \text{and} \quad \chi_S(w) \times_b \mathcal{D}_{G_2}^a.$$  \hspace{2cm} (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 $\chi_S(\text{director}) = \chi_S(\text{place}) = (1, 1, 1, 1, 1)$. Then, for edge $(\text{director, born_in, place})$,

$$\chi_S(\text{director}) \times_b \mathcal{D}_{G_1}^{\text{born_in}} = (1, 0, 0, 0, 0) = r_1$$
$$\chi_S(\text{place}) \times_b \mathcal{D}_{G_1}^{\text{born_in}} = (0, 1, 1, 0, 0) = r_2.$$

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 \footnote{For vector $v$ and matrix $A$, $v \times_b A = w$ where $w(j) = 1$ iff there is an $i$ such that $v(i) = 1$ and $A(i, j) = 1$.}. In the example above, $r_2$ shows that $\chi_S(\text{director}) \neq (1, 1, 1, 1, 1)$, since the only reachable nodes are director1 and director2. Thus, $\chi_S(\text{director}) = (1, 1, 1, 1, 1) \leq (0, 1, 1, 0, 0) = \chi_S(\text{place}) \times_b \mathcal{D}_{G_1}^{\text{born_in}}$, but according to Prop. \footnote{For vector $v$ and matrix $A$, $v \times_b A = w$ where $w(j) = 1$ iff there is an $i$ such that $v(i) = 1$ and $A(i, j) = 1$.} a dual simulation $S$ satisfies \footnote{For vector $v$ and matrix $A$, $v \times_b A = w$ where $w(j) = 1$ iff there is an $i$ such that $v(i) = 1$ and $A(i, j) = 1$.}, now possible to formulate by bit-matrix operations for edges $(v, a, w) \in E_{G_1},$

$$\chi_S(w) \leq \chi_S(v) \times_b \mathcal{D}_{G_2}^a \quad \text{and} \quad \chi_S(v) \leq \chi_S(w) \times_b \mathcal{D}_{G_2}^a.$$  \hspace{2cm} (10)

After observing the wrong value of $\chi_S(\text{director})$, we update relation $S$ to $S'$ by $\chi_S(\text{director}) := \chi_S(\text{director}) \land r_2$ (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 $\mathcal{E} = (\text{Var}, \text{Eq})$, where every node $v$ of the graph pattern is a variable, i.e., $\text{Var} := V_1$, and $\text{Eq}$ contains for each pattern edge $(v, a, w) \in E_{G_1}$, the following equations:

$$w \leq v \times_b \mathcal{D}_{G_2}^a \quad \text{and} \quad v \leq w \times_b \mathcal{D}_{G_2}^a.$$  \hspace{2cm} (11)

Fig. \footnote{For vector $v$ and matrix $A$, $v \times_b A = w$ where $w(j) = 1$ iff there is an $i$ such that $v(i) = 1$ and $A(i, j) = 1$.} shows the SOI for computing dual simulations for the graphs in Fig. \footnote{For vector $v$ and matrix $A$, $v \times_b A = w$ where $w(j) = 1$ iff there is an $i$ such that $v(i) = 1$ and $A(i, j) = 1$.} (a) and (b). Assignments to the variables, $v, w \in \text{Var}$, 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. Let $S_i$ be the current candidate relation. Pick any unstable inequality $\mathcal{E} \in \text{Eq}$

(a) If $S_i$ is valid for $\mathcal{E}$, set $\mathcal{E}$ stable and continue with (2).

(b) If $S_i$ is not valid for $\mathcal{E} = w \leq v \times_b \mathcal{A}$ (for $\mathcal{A} \in \{ \mathcal{D}_{G_2}^a, \mathcal{D}_{G_2}^a \mid a \in \Sigma \}$), then $\chi_S(w) \times_b \mathcal{A} = r$ and $\chi_S(v) \neq r$. Update $S_i$ to $S_{i+1}$ such that

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

Furthermore, every inequality $y \leq v \times \mathcal{A} \in \text{Eq}$ are reset to unstable. Mark $\mathcal{E}$ stable and continue with (2).
place ≤ director1 \times_b \delta_{\text{\texttt{born_in}}_{\text{\texttt{Fig. 2(a)}}}}^h
place ≤ director2 \times_b \delta_{\text{\texttt{born_in}}_{\text{\texttt{Fig. 2(b)}}}}^h
director1 ≤ place \times_b \delta_{\text{\texttt{born_in}}_{\text{\texttt{Fig. 2(b)}}}}^h
director2 ≤ place \times_b \delta_{\text{\texttt{born_in}}_{\text{\texttt{Fig. 2(b)}}}}^h
coworker ≤ director1 \times_b \delta_{\text{\texttt{worked_with}}_{\text{\texttt{Fig. 2(b)}}}}^h
director1 ≤ coworker \times_b \delta_{\text{\texttt{worked_with}}_{\text{\texttt{Fig. 2(b)}}}}^h
movie ≤ director2 \times_b \delta_{\text{\texttt{directed}}_{\text{\texttt{Fig. 2(b)}}}}^h
director2 ≤ movie \times_b \delta_{\text{\texttt{directed}}_{\text{\texttt{Fig. 2(b)}}}}^h

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

Please note that the initialization step of $S_0$ can also be expressed in terms of inequalities, in that for every pattern node $v$, we add \text{\texttt{1}} to the set of inequalities Eq. \text{\texttt{12}}

\begin{equation}
\nu \leq \text{\texttt{1}}
\end{equation}

\text{\texttt{1}} is the vector containing a 1 in every component. The dual simulation given by \text{\texttt{1}} is the largest solution to the SOI in Fig. 3 thus constitutes the largest dual simulation.

3.3 Complexity and Optimization

Initializing $S_0$ takes time $\Theta(|V_1| \cdot |V_2|)$ 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 $\varepsilon = v \leq w \times_b \mathcal{A}$ be in Eq and $S_i$ the current candidate relation. Computing $r = \chi_{S_i}(w) \times_b \mathcal{A}$ is in $O(|V_2|^2)$ time. By further regarding the intersection $\chi_{S_i}(v) \land r$, we obtain an overall time complexity of $O(|V_2|^2 + |V_2|) = O(|V_2|^2)$ for updating $S_i$ to $S_{i+1}$ which validates $\varepsilon$. For every edge in $G_1$ we have two equations in Eq. i.e., $|\text{\texttt{Eq}}| = \Theta(|E_1|)$. Thus, assuming pattern $G_1$ and data graph $G_2$ as input, our algorithm has a combined complexity of $O((|V_1| \cdot |V_2|) \cdot |E_1| \cdot |V_2|^2)$. In terms of data complexity we have a worst-case runtime of $O(|V_2|^3)$, virtually the same complexity as of any other dual simulation algorithm.

The acquired combined complexity of our solution is higher than that of an algorithm leveraging by Henzinger et al.’s so-called \text{\texttt{HHK algorithm}}, being in $O(mn)$ time. Note that HHK assumes a single node-labeled graph (i.e., no labels on the edges) $G = (V, E)$ and $|V| = n < m = |E|$ ($m \leq n^2$). Dual simulation requires the execution of HHK two times, leaving the overall complexity invariant. However, considering a separation into pattern and data graph as well as adding edge labels does have an effect on the resulting HHK adaptation. Separating pattern $G_1 = (V_1, E_1)$ from data graph $G_2 = (V_2, E_2)$ yields an overall combined complexity of $O(|E_2| \cdot |V_2|)$, i.e., the runtime complexity solely depends on the data graph $G_2$. The crux of HHK is an additional data structure which tracks for each pattern node $v$, the set of adjacent data nodes from which the simulating nodes of $v$ are unreachable. These are the definite nodes that cannot simulate the respective adjacent nodes. The maintenance of these removal sets is the key component in the complexity analysis \text{\texttt{17}}. Remarkably, combined and data complexity are equal for HHK for unlabeled graphs. If, additionally, edge labels are considered, the runtime estimation alters at least to $O(|\Sigma(G_1)| \cdot |V_2|^3)$, where $\Sigma(G_1)$ denotes the set of actually used labels in $G_1$. This is because every update of a removal set requires only a single adjacency matrix of size $O(|V_2|^2)$. However, for every label on the incident edge of a node, there is a different removal set to maintain.

Due to the graph query setting, there is no difference in worst-case data complexity between HHK and our solution. In fact, the algorithm of Ma et al. [20], adjusted to labeled graphs, enjoys the same data...
Computing Dual Simulation for Graph Database Queries

complexity, i.e., $\mathcal{O}(|V_2|^3)$. As a consequence, we formulate the follow specific data complexity hypothesis for dual simulation graph query processing: The real computation times of naïve implementations of HHK and the algorithm of Ma et al. should show no significant differences in the (labeled) graph query setting, i.e., for two given edge-labeled graphs. We provide experimental evidence for this hypothesis in Sect. 5. Although the existence of suitable algorithmic tweaks for any of the abovementioned algorithms is not deniable, we advertise our algorithmic framework for its separation into algorithmic representation as a system of inequalities and evaluation algorithm, externally adaptable by static and dynamic heuristics.

An immediate optimization is given by altering the initial relation $S_0$, syntactically exploiting that for a variable/node $v$ in $G_1$, candidate nodes are only those supporting incident edges of $v$. Therefore, let us denote by $f^a_{G_2}$ the bit-vector that summarizes the rows of $F^a_{G_2}$ in that $f^a_{G_2}(i) = 1$ if there is a $j$ with $S^a_{G_2}(i,j) = 1$, and $f^a_{G_2}(i) = 0$ otherwise. In the same lines, $b^a_{G_2}$ is defined as the summary of $B^a_{G_2}$. Then for each variable/node $v$ in $G_1$, we replace inequality (12) by

$$v \leq \wedge_{(v,a,w) \in E_1} f^b_{G_2} \wedge \wedge_{(a,a,v) \in E_1} b^d_{G_1}. \tag{13}$$

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, indicating 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 $v \leq w \times b A$ 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. 5.3), there is not a single heuristic that fits all input patterns and databases.

In our proof-of-concept implementation, we keep $G_2$ by its adjacency matrices in memory. $G_1$ is stored by its system of inequalities, including $|V_1|$ bit-vectors representing $\chi_S$. For every graph pattern $G_1$, we only need to load those adjacency matrices in memory that are needed according to $G_1$. 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 for the label with the largest number of stored bits. Combined with the memory-economical implementation by Atre et al. [5] [4], we are quite optimistic that our implementation may directly be used within the preprocessing step of the BitMat tool set. In Sect. 5, our dual simulation processing applied to SPARQL queries yields decent pruning factors significantly improving upon those reported by Atre [4].

4 Dual Simulation for SPARQL

Having clarified the foundational and algorithmic aspects of dual simulation we now approach an actual query language, namely SPARQL. We choose SPARQL for its high-quality standardization by the W3C [29] and its extensive formal treatment, e.g., [27] [2] [3]. Although SPARQL 1.1 has been around for some time, the fundamental properties of the query language remain the same as for SPARQL 1.0. We are aware of the recent report on the semantic foundation of the Neo4J query language Cypher [14], and confident about the wider applicability of the forthcoming techniques to this language. 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.

### 4.1 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 \( v, v_1, v_2, \ldots \). A variable \( v_1 \) is usually introduced by a leading question mark as \(?v_1\) (cf. \((\mathcal{X}_1)\)).

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 \( t = (v_1, \text{population}, v_2) \) gives rise to a match identifying \( v_1 \) with node \texttt{Saint Joan} and \( v_2 \) with the literal \( 70.063 \) (cf. Fig. 1(a)). By \( \text{vars}(t) \) we denote the set of variables occurring in triple \( t \), i.e., \( \text{vars}(t) = \{v_1, v_2\} \) for the \( t \) mentioned above. A candidate w.r.t. \( DB \) is a partial function \( \mu : \mathcal{V} \rightarrow O_{DB} \). The set of variables for which candidate \( \mu \) is denoted by \( \text{dom}(\mu) \). A candidate \( \mu \) is a match for triple \( t \) in \( DB \) iff \( \text{dom}(\mu) = \text{vars}(t) \) and, assuming \( t = (v_1, a, v_2) \), \( (\mu(v_1), a, \mu(v_2)) \in E_{DB} \), abbreviated by \( \mu(t) \in DB \).

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

For dual simulation processing of a BGP \( G \) w.r.t. \( DB \), we compute the largest dual simulation between \( G(\mathcal{G}) \) and \( DB \). This procedure is sound in that every match \( \mu \) for \( G \) 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 \( G \) be a BGP. Each \( \mu \in [G]_{DB} \) is a dual simulation between \( G(\mathcal{G}) \) and \( DB \).

**Proof:** We show that \( \mu \) is a dual simulation between \( G(\mathcal{G}) \) and \( DB \). Let \( (v, o) \in \mu \), i.e., \( \mu(v) = o \), and let \( t \in G \) such that \( v \in \text{vars}(t) \). There are two cases to distinguish, for some \( a \in \Sigma \) and \( w, u \in \text{vars}(G) \), (a) \( t = (v, a, w) \) and (b) \( t = (u, a, v) \). Since case (b) is completely analogous, we consider only (a). As \( \mu \) is a a match for \( G \), it is a match for \( t \), i.e., there is exactly one \( o' = \mu(w) \) and \( (o, a, o') \in E_{DB} \). Hence, \( o' \) meets the requirements of Def. 2(i).

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, \( G \) a BGP and \( S \) the largest dual simulation between \( G(\mathcal{G}) \) and \( DB \). For each database node \( o \in O_{DB} \) such that there are \( v \in \text{vars}(G) \) and \( \mu \in [G]_{DB} \) with \( \mu(v) = o \), it holds that \( (v, o) \in S \).

**Proof:** Towards a contradiction, assume there is a database node \( o \) relevant to match variable \( v \) by \( \mu \in [G]_{DB} \) with \( (v, o) \notin S \). But then \( S \cup \mu \) is a dual simulation larger than \( S \), contradicting the assumption that \( S \) is the largest one.

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. 1(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 \( v \) is that nodes \( p_1 \) and \( p_3 \) distribute the obligations for simulating variable/node \( u \). 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. Non-transitive relationships sometimes appear transitive under dual simulation. As long as acyclic queries are concerned, our process is complete. Since this class of queries is rather small, we are not formally justifying this statement.

We compute the largest dual simulation by the largest solution to the SOI constructed from \( G(G) \) (cf. Sect. 3). 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 3** Let \( DB \) be a graph database, \( \mathcal{Q} \) a SPARQL query and \( \mathcal{E} \) any SOI representation of \( \mathcal{Q} \) with solutions \( S \subseteq \text{vars}(\mathcal{Q}) \times O_{DB} \). \( \mathcal{E} \) 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 \( v \in \text{vars}(\mathcal{Q}) \) and \( \mu \in [\mathcal{Q}]_{DB} \), then \((v,o) \in S\).

### 4.2 Advanced Graph Patterns

BGP, and SPARQL queries in general, may be combined by operators, further restricting and combining 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 with the UNION-operators.

The UNION-operator is the least invasive operator. It combines any two queries \( \mathcal{Q}_1 \) and \( \mathcal{Q}_2 \) to query \( \mathcal{Q}_1 \cup \mathcal{Q}_2 \). The result set is the union of the result sets of the constituent queries, i.e., \([\mathcal{Q}_1 \cup \mathcal{Q}_2]_{DB} := [\mathcal{Q}_1]_{DB} \cup [\mathcal{Q}_2]_{DB}\). It is well-known that any SPARQL query may be rewritten as the union of finitely many union-free queries. A SPARQL query \( \mathcal{Q} \) is union-free if the UNION-operator does not occur in \( \mathcal{Q} \).

**Proposition 3 (Proposition 3.8 [27])** Let \( \mathcal{Q} \) be a SPARQL query. Then there exist union-free SPARQL queries \( \mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_k (k \in \mathbb{N}) \) such that \( \mathcal{Q} \) is equivalent to \( \mathcal{Q}' = \mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \ldots \cup \mathcal{Q}_k \), i.e., \([\mathcal{Q}]_{DB} = [\mathcal{Q}']_{DB}\).

The construction of \( \mathcal{Q}' \) follows similar principles as constructing the DNF (disjunctive normal form) in propositional logic. In consequence, the result set of \( \mathcal{Q} \) is the union of the result sets of all the \( \mathcal{Q}_i \) \((1 \leq i \leq k)\). This means that instead of \( \mathcal{Q} \), we may process each union-free part of \( \mathcal{Q} \) individually and later combine their results. Hence, we assume every query to be union-free in the following.

While SPARQL’s disjunction unifies the result sets of the constituents, conjunction unifies compatible results, i.e., those results agreeing upon shared variables. Intuitively, two graph matches are joined to one result. Matches \( \mu_1 \) and \( \mu_2 \) are compatible, denoted \( \mu_1 \equiv \mu_2 \), if for all \( v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2) \) (\( v \) shared by \( \mu_1 \) and \( \mu_2 \)), \( \mu_1(v) = \mu_2(v) \). The conjunction of two queries \( \mathcal{Q}_1 \) and \( \mathcal{Q}_2 \) is the query \( \mathcal{Q}_1 \land \mathcal{Q}_2 \). As an
example, the SPARQL representation of the graph pattern in Fig. 4(a) may be described as the conjunction of two BGP's, \( G_1 = \{(v, \text{knows}, w)\} \) and \( G_2 = \{(u, \text{knows}, v)\} \). The semantics of conjunctions is defined by

\[
[\mathcal{D}_1 \text{ AND } \mathcal{D}_2]_{DB} := \{ \mu_1 \cup \mu_2 \mid \mu_i \in [\mathcal{D}_i]_{DB} \land \mu_1 = \mu_2 \}.
\]

For example, in the database in Fig. 4(b), queries \( G_i \) from above enjoy matches \( \mu_i (i = 1, 2) \) with \( \mu_1 (v) = \mu_2 (v) = p_1 \) and \( \mu_1 (w) = \mu_2 (w) = p_2 \). These matches are compatible, thus \( (\mu_1 \cup \mu_2) \in [G_1 \text{ AND } G_2]_{DB} \).

In contrast, \( \mu_1 \) from before and \( \mu_3 \) with \( \mu_3 (w) = p_2 \) and \( \mu_3 (v) = p_3 \) constitute incompatible matches, thus \( (\mu_1 \cup \mu_3) \notin [G_1 \text{ AND } G_2]_{DB} \). In a relational setting, conjunction simply is the explicit expression of inner joins.

Regarding our dual simulation process, for conjunctions \( \mathcal{D}_1 \text{ AND } \mathcal{D}_2 \), we create the systems of inequalities for \( \mathcal{D}_1 \) and \( \mathcal{D}_2 \) separately, denoted by \( \mathcal{E}(\mathcal{D}_1) \) and \( \mathcal{E}(\mathcal{D}_2) \). Recall that the variables of both queries directly refer to variables occurring in \( \mathcal{D}_1 \) and \( \mathcal{D}_2 \), respectively. The semantics of conjunctions requires matches to queries \( \mathcal{D}_1 \) and \( \mathcal{D}_2 \) 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 \( \mathcal{D}_1, \mathcal{D}_2 \) two SPARQL queries with sound systems of inequalities \( \mathcal{E}(\mathcal{D}_1) = (\text{Var}_1, \text{Eq}_1) \) and \( \mathcal{E}(\mathcal{D}_2) = (\text{Var}_2, \text{Eq}_2) \). Then \( \mathcal{E} = (\text{Var}_1 \cup \text{Var}_2, \text{Eq}_1 \cup \text{Eq}_2) \) is sound for \( \mathcal{D}_1 \text{ AND } \mathcal{D}_2 \).

**Proof:** Let \( \mu \in [\mathcal{D}_1 \text{ AND } \mathcal{D}_2]_{DB} \). It holds that \( \mu = \mu_1 \cup \mu_2 \) for compatible \( \mu_i \in [\mathcal{D}_i]_{DB} (i = 1, 2) \). Let \( v \in \text{vars}(\mathcal{D}_1) \text{ AND } \mathcal{D}_2 \) with \( \mu(v) = o \). We need to show that the largest solution \( S \) of \( \mathcal{E} \) contains \( (v, o) \).

In case \( v \in \text{vars}(\mathcal{D}_1) \setminus \text{vars}(\mathcal{D}_2) \) (i, j = 1, 2 and \( i \neq j \)), \( (v, o) \in S \) follows from soundness of \( \mathcal{E}(\mathcal{D}_j) \), since variable \( v \) cannot be influenced by any triple pattern of \( \mathcal{E}(\mathcal{D}_j) \). Otherwise, \( v \in \text{vars}(\mathcal{D}_1) \cap \text{vars}(\mathcal{D}_2) \) and it holds that \( \mu_1 (v) = \mu_2 (v) = o \). Hence, the largest solutions \( S_i \) of \( \mathcal{E}(\mathcal{D}_i) \) (i = 1, 2) contain \( (v, o) \), i.e., \( (v, o) \in S_1 \cap S_2 \). It remains to be shown that \( S_1 \cap S_2 \subseteq S \). Let \( (v, o) \in S_1 \cap S_2 \). By construction, any \( \varepsilon \in \text{Eq} \) either comes from \( \text{Eq}_1 \) or \( \text{Eq}_2 \), and since \( (v, o) \in S_1 \cap S_2 \), \( (v, o) \) cannot contradict \( \varepsilon \). Thus, \( (v, o) \) belongs to the largest solution \( S \).

Please note that the lemma is independent of the shape of \( \mathcal{D}_1 \) and \( \mathcal{D}_2 \). We only require that the respective systems of inequalities are sound w.r.t. the queries (cf. Def. 3).

### 4.3 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 \( \text{OPTIONAL} \), 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 .
  }
}
```

Optional patterns are left-outer joins in the relational model, i.e., matches to \( \mathcal{X}_2 \) assign nodes from the database to variable \( ?\text{director} \) and \( ?\text{movie} \), but also to variable \( ?\text{coworker} \) 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 \( \mathcal{D}_1 \) and
The result set of \( \mathcal{Q}_1 \) AND \( \mathcal{Q}_2 \) is contained in the result set of the optional pattern. Additionally, all matches to \( \mathcal{Q}_1 \) that have no compatible matches to \( \mathcal{Q}_2 \) are matches, 
\[
[\mathcal{Q}_1 \text{ OPTIONAL} \mathcal{Q}_2]_{\mathcal{DB}} := [\mathcal{Q}_1 \text{ AND } \mathcal{Q}_2]_{\mathcal{DB}} \cup \{ \mu \in [\mathcal{Q}_1]_{\mathcal{DB}} | \exists \mu' \in [\mathcal{Q}_2]_{\mathcal{DB}} : \mu = \mu' \}.
\]

In \( \mathcal{Q}'_2 \), variable \(?director\) occurs in two different roles. First, the optional pattern mandates variable \(?director\) to feature triples with label \text{directed}. Second, triples labeled \text{worked_with} are only optional. These two roles must be reflected by our SOI representation of \( \mathcal{Q}'_2 \) by including two copies of that variable, \(?director_m\) (mandatory) and \(?director_o\) (optional) with the property that a solution \( S \) in variable \(?director_o\) must not exceed \( S \) in variable \(?director_m\). In other words, there is no database node matching \(?director_o\) that does not match \(?director_m\). This is expressed by the following inequality,
\[
?director_o \leq ?director_m. \tag{14}
\]

Towards the general case, let us discuss another example, 
\[
\{(v_1,a,v_2)\} \text{ OPTIONAL} \{(v_3,b,v_2)\} \text{ AND } \{(v_3,c,v_4)\}. \tag{\mathcal{Q}'_3}
\]

The query consists of three triple patterns, the first two constitute an optional pattern and the results are joined with the third triple pattern. Fig. 5(b) and (c) show graph representations of the matches of \( \mathcal{Q}'_3 \) w.r.t. the graph database in Fig. 5(a), in which the variable assignments are indicated as labels next to the nodes. As before, we have variables occurring in two different roles, here \( v_2 \) and \( v_3 \). Analogous to \( \mathcal{Q}'_2 \), we simply derive \( v_{2m} \) and \( v_{2o} \) with \( v_{2o} \leq v_{2m} \) from the optional pattern. However, the first occurrence of \( v_3 \) is optional w.r.t. the second occurrence (in the third triple pattern), since \( v_3 \) must match a \( c \)-labeled edge and may feature the \( b \)-labeled edge from the optional pattern. We observe that occurrences of the same variables, e.g., \( v_3 \), may have interdependencies that we need to consider, even beyond optional patterns. Therefore, we first introduce the full syntax covered in this paper to derive a system of inequalities for each query following that syntax. Second, we derive sound SOIs for optional patterns.

Our query language \( \mathcal{I} \) comprises union-free SPARQL queries with AND and OPTIONAL operators. Queries in \( \mathcal{I} \) are derived by the following grammar,
\[
\mathcal{D} ::= \mathcal{G} \mid \mathcal{D} \text{ AND } \mathcal{D} \mid \mathcal{D} \text{ OPTIONAL } \mathcal{D}
\]
where \( \mathcal{G} \) ranges over by BGPs. Queries in \( \mathcal{I} \) range over by \( \mathcal{D}, \mathcal{D}_1, \mathcal{D}_2, \ldots \). As observed above, we need to consider mandatory and optional variable occurrences. Function \text{mand} maps queries \( \mathcal{Q} \) from \( \mathcal{I} \) to the set of variables that occur as mandatory in \( \mathcal{Q} \), defined by
1. \( \text{mand}(\mathcal{G}) := \text{vars}(\mathcal{G}) \),
2. \( \text{mand}(\mathcal{Q}_1 \text{ AND } \mathcal{Q}_2) := \text{mand}(\mathcal{Q}_1) \cup \text{mand}(\mathcal{Q}_2) \), and
3. \( \text{mand}(\mathcal{Q}_1 \text{ OPTIONAL } \mathcal{Q}_2) := \text{mand}(\mathcal{Q}_1) \).

For handling optional pattern \( \mathcal{Q}_1 \text{ OPTIONAL } \mathcal{Q}_2 \) correctly, we need to decide, in which cases an occurrence of variable \( v \) in \( \mathcal{Q}_2 \) has an optional dependency to another occurrence of the same variable. The case \( v \in \text{vars}(\mathcal{Q}_1) \) is reflected by query \( \mathcal{Q}_2 \). 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., \( (14) \). More precisely, for the special case of query \( \mathcal{Q} = \mathcal{Q}_1 \text{ OPTIONAL } \mathcal{Q}_2 \), 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 \( v \in \text{vars}(\mathcal{Q}_2) \cap \text{mand}(\mathcal{Q}_1) \). For \( v \in \text{vars}(\mathcal{Q}_2) \cap \text{mand}(\mathcal{Q}_1) \), we reserve a unique name \( v_{\mathcal{Q}_2} \), which we use to replace \( v \) in every inequality of \( \mathcal{Q}_2 \), achieved by a renaming \( \rho := \{(v, v_{\mathcal{Q}_2}) \mid v \in \text{vars}(\mathcal{Q}_2) \cap \text{mand}(\mathcal{Q}_1)\} \). Upon renaming, we add inequality 

\[
\forall v_{\mathcal{Q}_2} \leq v \tag{15}
\]

for \( v \in \text{vars}(\mathcal{Q}_2) \cap \text{mand}(\mathcal{Q}_1) \) to the overall SOI. The largest solution to the resulting SOI consists of all assignments to the new variables \( v_{\mathcal{Q}_2} \), i.e., to variables not occurring in the original formulation of the query. Since these variables are only needed to handle optionality correctly, and since the largest solution for these variables is subsumed by the respective mandatory variables (cf. \( (15) \)), 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 \( \mathcal{E}(\mathcal{Q}_1) = (\text{Var}_1, \text{Eq}_1) \) and \( \mathcal{E}(\mathcal{Q}_2) = (\text{Var}_2, \text{Eq}_2) \). Furthermore, define renaming as \( \rho := \{(v, v_{\mathcal{Q}_2}) \mid v \in \text{vars}(\mathcal{Q}_2) \cap \text{mand}(\mathcal{Q}_1)\} \).

Then

\[
\mathcal{E} := (\text{Var}_1 \cup \rho(\text{Var}_2), \text{Eq}_1 \cup \rho(\text{Eq}_2) \cup \text{Eq}_o)
\]

with \( \text{Eq}_o := \{v_{\mathcal{Q}_2} \leq v \mid v \in \text{vars}(\mathcal{Q}_2) \cap \text{mand}(\mathcal{Q}_1)\} \) is sound for \( \mathcal{Q}_1 \text{ OPTIONAL } \mathcal{Q}_2 \).

**Proof:** Let \( \mu \in \left[ \mathcal{Q}_1 \text{ OPTIONAL } \mathcal{Q}_2 \right]_{DB} \) with \( \mu(v) = o \) for \( v \in \text{vars}(\mathcal{Q}_1 \text{ OPTIONAL } \mathcal{Q}_2) \). We need to show that \( (v, o) \in S \) where \( S \) is the largest solution of \( \mathcal{E} \). There are two cases to distinguish, (a) \( \mu = \mu_1 \cup \mu_2 \) where \( \mu_i \in \left[ \mathcal{Q}_i \right]_{DB} \) (\( i = 1, 2 \)) with \( \mu_1 \Rightarrow \mu_2 \) and (b) \( \mu = \mu_1 \) where \( \mu_1 \in \left[ \mathcal{Q}_1 \right]_{DB} \) and there is no \( \mu_2 \in \left[ \mathcal{Q}_2 \right]_{DB} \) compatible to \( \mu_1 \). Case (a) becomes analogous to the proof of Lemma 3 considering that for any occurrence of \( v_{\mathcal{Q}_2} \) in \( \rho(\text{Eq}_2) \), inequality \( (15) \) makes the requirements upon \( v_{\mathcal{Q}_2} \) only weaker. Hence, \( \mu_2(v) \) is preserved. In case (b), we distinguish two further cases for variable \( v \), (i) \( v \in \text{vars}(\mathcal{Q}_1) \setminus \text{vars}(\mathcal{Q}_2) \) and (ii) \( v \in \text{vars}(\mathcal{Q}_1) \cap \text{vars}(\mathcal{Q}_2) \). The claim for case (i) directly follows from the sound SOI \( \mathcal{E}(\mathcal{Q}_1) \). In case (ii), it might be that in the largest solution \( S_2 \) of \( \mathcal{E}(\mathcal{Q}_2) \), \( (v, o) \notin S_2 \). However, \( v \) is subject to renaming, since it is a variable of both sub-queries. Therefore \( (v_{\mathcal{Q}_2}, o) \notin \rho S_2 \) but as we added inequality \( (15) \) to \( \text{Eq}_o \), we get that \( (v, o) \in S \) by soundness of \( \mathcal{E}(\mathcal{Q}_1) \). \( \square \)

### 4.3.1 The General Case

The general case, outlined by example \( \mathcal{Q}_3 \), needs to take the contexts of optional patterns into account. Since \( v_3 \) has a mandatory occurrence in \( \mathcal{Q}_3 \) but an optional in the sub-query \{\( \{v_1, a, v_2\} \)\} \text{ OPTIONAL } \{\( \{v_3, b, v_2\}\)\}, SPARQL’s evaluation semantics defines the second occurrence of \( v_3 \) to be mandatory w.r.t. the first. For any optional pattern \( \mathcal{Q}_1 \text{ OPTIONAL } \mathcal{Q}_2 \) occurring as a sub-query of a query \( \mathcal{Q} \in \mathcal{I} \), if a variable \( v \in \text{vars}(\mathcal{Q}_2) \) occurs as mandatory in \( \mathcal{Q} \), then we perform the same renaming as in Lemma 4.
for $\mathcal{D}_2$. For a variable $v \in \text{vars}(\mathcal{D}_2)$, there may be several candidates. From all the choices we pick the \textit{syntactically closest}. As an example, consider the optional patterns

$$P = (P_1 \text{ OPTIONAL } P_2) \text{ OPTIONAL } P_3$$
$$R = R_1 \text{ OPTIONAL } (R_2 \text{ OPTIONAL } R_3).$$

Assume that $y \in \text{vars}(P_i) \ (i = 1, 2, 3)$ and $z \in \text{vars}(R_i) \ (i = 1, 2, 3)$. The occurrences of $y$ in $P_2$ and $P_3$ are syntactically closest to the mandatory occurrence of $y$ in $P_1$, giving rise to inequalities

$$y_{P_2} \leq y \leq y_{P_3}.$$

It may also be that $x \in \text{vars}(P_i) \ (i = 2, 3)$ and $x \notin \text{vars}(P_1)$. In these situations, we rename $x$ to $x_{P_2}$ and $x_{P_3}$, respectively, but would not add any interdependencies between these variables. In extreme cases, the original variable $x$ may not occur in the resulting SOI at all. In these cases, the soundness proof requires that every solution to $x_{P_2}$ or $x_{P_3}$ also is a solution to variable $x$.

The occurrence of $z$ in $R_3$ is closest to the occurrence in $R_2$, and the occurrence in $R_2$ is closest to $R_1$, raising the following inequalities,

$$z_{R_3} \leq z_{R_2} \leq z.$$

Handling the general case formally, needs to conduct a notion of $\mathcal{S}$-contexts, being queries with holes. Since the proof of the resulting soundness lemma is completely analogous, thus gives no more insights than the proof of Lemma 4, our considerations about optional patterns are complete.

At this point, we have given sound SOI representations for SPARQL queries. The following subsection gives the cumulating theorem in this respect as well as some final remarks on the limits of our pruning process. Sect. 5 provides evidence of the effectiveness and efficiency of the derived procedure.

4.4 Discussion

Before discussing an important query type, we conclude this section by the soundness theorem.

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

The proof consists of an induction over the structure of $\mathcal{D}$ and uses all results obtained so far.

**Proof:** We proceed by induction over the structure of $\mathcal{D}$. For the base case, $\mathcal{D} = G$, Theorem 1 provides us with the necessary argument. Since the largest dual simulation is the largest solution of the respective SOI, soundness of $\mathcal{E}(G)$ immediately follows. Assume for queries $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{S}$, soundness of the respective SOIs $\mathcal{E}(\mathcal{D}_1)$ and $\mathcal{E}(\mathcal{D}_2)$ is already provided, which may already conduct some renaming due to our discussion in Sect. 4.3. For the recursive step, we distinguish two cases. First, if $\mathcal{D} = \mathcal{D}_1 \text{ AND } \mathcal{D}_2$, then $\mathcal{E}(\mathcal{D})$ is sound due to Lemma 3. Lemma 4 proves soundness of $\mathcal{E}(\mathcal{D})$ with $\mathcal{D} = \mathcal{D}_1 \text{ OPTIONAL } \mathcal{D}_2$. □

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

---

3 Adjustments to the soundness notion has no influence on the lemma’s correctness.
Our dual simulation process is not restricted to well-designed patterns, being SPARQL queries \( \mathcal{Q} \) with the property that for every sub-query \( \mathcal{Q}_1 \) OPTIONAL \( \mathcal{Q}_2 \) and every \( v \in \text{vars}(\mathcal{Q}_2) \) that also occur outside the optional pattern, \( v \in \text{vars}(\mathcal{Q}_1) \) \cite{27}. Query \( \mathcal{Q}_3 \) is not well-designed, since \( v_3 \) 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 \( v_3 \) 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 \( \mathcal{S} \), \( \text{PSPACE} \)-completeness of the evaluation problem \cite{30} may be evaded, since checking whether a given relation \( S \) constitutes a valid assignment to \( \mathcal{E}(Q) \) may be performed in \( \text{PTIME} \). However, more expressive fragments of SPARQL add combinatorial complexity not solvable by pure dual simulation pattern matching.

There are two reasons making well-designed patterns interesting. First, the fragment containing only well-designed patterns has a \( \text{CO} \)-NP-complete evaluation problem \cite{27,2}, as opposed to \( \text{PSPACE} \)-completeness of SPARQL’s evaluation problem. Second, every well-designed pattern is \textit{weakly monotone} \cite{2}, an important property when discussing \textit{NULL} semantics. To this end, we cannot tell whether or not we handle all weakly monotone queries effectively. However, the next section provides evidence that our pruning is quite effective.

5 Evaluation

In this section, we evaluate our prototype called \textsc{sparqlsim}. First, we compare our algorithm to the state-of-the-art dual simulation algorithm as introduced by Ma et al. \cite{20} and used in implementations of \cite{24,31,20} for evaluation purposes. 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 pruning effectiveness, 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 \cite{4}. Details concerning the evaluation results, a list of queries, and our implementation can be found on our project’s Github page.

5.1 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 \cite{9} and the high-performance in-memory database RDFox \cite{25}. 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. To achieve stable query results, we deactivated caching for Virtuoso. RDFox is not using any caching techniques at all. 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 processed the queries on the pruned result sets 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.
Computing Dual Simulation for Graph Database Queries

Table 2: Runtimes of our SPARQLSIM for BGPs from queries $B_0$-$B_{21}$ compared to Ma et al. [20].

<table>
<thead>
<tr>
<th>Query</th>
<th>$t_{SPARQLSIM}$</th>
<th>$t_{MA \ ET \ AL.}$</th>
<th>Query</th>
<th>$t_{SPARQLSIM}$</th>
<th>$t_{MA \ ET \ AL.}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>0.10385</td>
<td>6.72121</td>
<td>$B_{10}$</td>
<td>0.02397</td>
<td>0.27126</td>
</tr>
<tr>
<td>$B_1$</td>
<td>0.03876</td>
<td>3.33471</td>
<td>$B_{11}$</td>
<td>0.01392</td>
<td>0.02099</td>
</tr>
<tr>
<td>$B_2$</td>
<td>0.79097</td>
<td>3.84781</td>
<td>$B_{12}$</td>
<td>0.01477</td>
<td>0.02287</td>
</tr>
<tr>
<td>$B_3$</td>
<td>0.69797</td>
<td>5.62662</td>
<td>$B_{13}$</td>
<td>0.35515</td>
<td>11.30355</td>
</tr>
<tr>
<td>$B_4$</td>
<td>0.00003</td>
<td>0.00004</td>
<td>$B_{14}$</td>
<td>5.46599</td>
<td>16.63957</td>
</tr>
<tr>
<td>$B_5$</td>
<td>0.04091</td>
<td>0.31700</td>
<td>$B_{15}$</td>
<td>13.43710</td>
<td>24.99660</td>
</tr>
<tr>
<td>$B_6$</td>
<td>0.41105</td>
<td>0.54291</td>
<td>$B_{16}$</td>
<td>0.00002</td>
<td>0.00003</td>
</tr>
<tr>
<td>$B_7$</td>
<td>0.26991</td>
<td>0.51206</td>
<td>$B_{17}$</td>
<td>1.12649</td>
<td>2.30390</td>
</tr>
<tr>
<td>$B_8$</td>
<td>0.13562</td>
<td>5.51084</td>
<td>$B_{18}$</td>
<td>0.32056</td>
<td>0.54057</td>
</tr>
<tr>
<td>$B_9$</td>
<td>0.02551</td>
<td>0.08707</td>
<td>$B_{19}$</td>
<td>0.69515</td>
<td>5.15070</td>
</tr>
</tbody>
</table>

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 [15] (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 $D_0$-$D_5$ stem from [4], benchmark queries $B_0$-$B_{19}$ appeared in the DBpedia benchmark dataset in [23]. The LUBM benchmark dataset was generated for 10,000 universities, comprising 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. $L_0$-$L_5$).

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., \texttt{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.

5.2 Evaluation Analysis

Comparing Dual Simulation Algorithms Due to the fact that Ma et al.’s algorithm [20] considers BGPs as input, we have removed the \texttt{OPTIONAL} keyword from benchmark queries $B_0$-$B_{21}$. Evaluation times are shown in Table 2. We observe that the optimizations allowed by SPARQLSIM (cf. Sect. 3.3) 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.

Dual Simulation as Pruning Mechanism for SPARQL First, we analyze SPARQLSIM’s pruning effectiveness (cf. Table 3) 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. For queries with 0 triples left, there is no need for any further query evaluation. 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 3). In comparison, the effectiveness of our pruning is smaller for LUBM queries, being least effective for the query $L_1$. Here, only 0.9% of the triples after
Table 3: Result set sizes, numbers of required triples, runtimes of SPARQLSIM in seconds and numbers of triples after pruning.

<table>
<thead>
<tr>
<th>Query</th>
<th>Result No.</th>
<th>Req. Triples</th>
<th>$t_{\text{SPARQLSIM}}$</th>
<th>Tripl. aft. Pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_0$</td>
<td>10,448,905</td>
<td>3,276,841</td>
<td>106.451</td>
<td>10,181,730</td>
</tr>
<tr>
<td>$L_1$</td>
<td>226,641</td>
<td>114,989</td>
<td>8.464</td>
<td>25,429,750</td>
</tr>
<tr>
<td>$L_2$</td>
<td>32,828,280</td>
<td>15,416,012</td>
<td>147.335</td>
<td>48,674,046</td>
</tr>
<tr>
<td>$L_3$</td>
<td>11</td>
<td>35</td>
<td>0.138</td>
<td>126</td>
</tr>
<tr>
<td>$L_4$</td>
<td>10</td>
<td>33</td>
<td>0.125</td>
<td>101</td>
</tr>
<tr>
<td>$L_5$</td>
<td>7</td>
<td>35</td>
<td>1.220</td>
<td>35</td>
</tr>
</tbody>
</table>

| $R_0$ | 523,066    | 3,139,273    | 4.396                  | 3,141,102          |
| $R_1$ | 12         | 60           | 0.088                  | 60                 |
| $R_2$ | 5794       | 28,704       | 0.143                  | 28,704             |
| $R_3$ | 25,102,459 | 22,630,477   | 6.230                  | 22,691,521         |
| $R_5$ | 365,693    | 79,943       | 0.574                  | 79,944             |
| $B_0$ | 12         | 60           | 0.088                  | 60                 |
| $B_1$ | 859,751    | 726,749      | 0.022                  | 726,812            |
| $B_2$ | 913,786    | 1,587,731    | 0.532                  | 1,588,127          |
| $B_3$ | 438,542    | 386,000      | 0.606                  | 386,020            |
| $B_4$ | 0          | 0            | 0.000                  | 0                  |
| $B_5$ | 0          | 0            | 0.033                  | 0                  |
| $B_6$ | 815,522    | 886,826      | 0.503                  | 886,939            |
| $B_7$ | 34,991     | 37,965       | 0.443                  | 37,965             |
| $B_8$ | 8416       | 30,258       | 0.113                  | 30,258             |
| $B_9$ | 8247       | 13,116       | 0.022                  | 13,116             |
| $B_{10}$ | 8061 | 12,642 | 0.027 | 12,642 |
| $B_{11}$ | 9849 | 8955 | 0.018 | 8955 |
| $B_{12}$ | 9554 | 8660 | 0.018 | 8660 |
| $B_{13}$ | 123,467 | 365,131 | 0.273 | 365,154 |
| $B_{14}$ | 22,673,220 | 27,652,055 | 4.322 | 27,747,192 |
| $B_{15}$ | 0 | 0 | 0.000 | 0 |
| $B_{16}$ | 2 | 4 | 0.009 | 4 |
| $B_{17}$ | 7,898,331 | 8,285,964 | 0.917 | 8,294,385 |
| $B_{18}$ | 66,903 | 41,808 | 0.472 | 41,808 |
| $B_{19}$ | 879,460 | 292,531 | 0.602 | 292,541 |
pruning are actually part of some result. Later on we provide evidence that, e.g., for \( L_1 \), our pruning still 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 more than 30 iterations to compute the largest dual simulation, leading to high running times of our algorithm, e.g., for \( L_0 \) or \( L_2 \). 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 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.

**Comparison to State-Of-The-Art RDF Databases**  
The next experiments compare the query evaluation time of the in-memory database RDFox to SPARQLSIM in combination with RDFox as a query processor. In Table 4 we observe an improvement of the query time in 15 out of 32 queries. Particularly interesting is our improvement on query \( L_1 \) with a query processing time of 25,900 seconds on RDFox. Here, we could run our dual simulation algorithm in only 8 seconds (cf. Table 3), decreasing the query time of RDFox by more than 20 times. For \( L_0 \), however, \( t_{\text{SPARQLSIM}} \) alone is around 5 times slower than RDFox \( (t_{\text{DB}}) \). Also, in queries \( D_5, B_0, B_7-B_9, B_17, B_21 \) we show good improvements of the in-memory databases’ query times. For most of the remaining queries we show comparable results to RDFox varying by some milliseconds.

Table 5 shows an improvement of the runtimes of only 3 queries for Virtuoso. For most other queries, evaluation times are on par with \( t_{\text{DB}} \). We notice that 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., \( D_4 \) with doubled evaluation time \( t_{\text{DB \ pruned}} \) on the 3% portion of DBpedia. We believe that Virtuoso could benefit from an integration of SPARQLSIM as a pruning technique. In turn, our tool may advance by employing Virtuoso’s built-in heuristics for query planning. On the downside, our algorithm is often slightly slower than the professionally implemented and highly optimized RDF triple store. Particularly, some of the more complex queries took longer to produce the pruning than for Virtuoso to produce the answers. These queries took several iterations in SPARQLSIM. We believe that we can benefit from more sophisticated join order optimization techniques as used for example in Virtuoso which could boost our query time tremendously. The very fast pruning time for the cyclic query \( L_1 \) requires only two iterations, and thereby shows the potential of our technique.

**5.3 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 \( L_1 \) is most profitable, since huge intermediate tables can be avoided. In this case we observe a decrease by more than one order of magnitude while the pruning time is vastly fast in only two iterations. In contrast, because intermediate results in the evaluation of \( L_0 \) are rather small, the benefits of dual simulation pruning are not as significant as for \( L_1 \). Furthermore, the low selectivity predicates of \( L_0 \) result in a rather big number of iterations that increases the pruning time compared to e.g., \( L_1 \). 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.
Both queries, discussed so far, are outstanding in their own roles. While $\mathcal{L}_0$’s evaluation is always faster than the computation of the dual simulation pruning, both database systems we considered benefit from the pruning for $\mathcal{L}_1$. The mandatory cores of both queries are depicted in Fig. 6. First observe that both queries are cyclic. Although $\mathcal{L}_0$ is quite small, our dual simulation algorithm takes more than 30 iterations until it reaches the fixpoint. From a brute force analysis we learn that the number of iterations may be reduced by 16, but only resulting in half the time of the computation reported in Table 3. After having stabilized the equations for any two nodes of $\mathcal{L}_0$, regarding the third node may invalidate equations for the other two nodes again. Hence, the evaluation performance of Virtuoso and RDFox cannot be beaten by our current implementation, no matter which specific heuristic we choose. Remarkably, the predicates of $\mathcal{L}_0$ share quite a low selectivity rate. In contrast, dual simulation between query $\mathcal{L}_1$ and the LUBM dataset takes only two iterations, allowing for an overall improvement of Virtuoso as well as RDFox.

Regarding the effectiveness of the pruning, LUBM query $\mathcal{L}_1$ represents one the worst examples with over 200 times more leftover triples than necessary. The reason for such a huge difference can be found in the counterexample to Theorem 1 described at the end of Sect. 4.1. Let us transfer the known example by considering a subexpression of query $\mathcal{L}_1$ which is depicted in Fig. 6(b). At its core, $\mathcal{L}_1$ asks for all publications together with two of their authors, both affiliated with a department (one is a student member, the other is an employee) that is part of the university from which the student got their degree. Suppose we have two disjoint matches isomorphic to the graph representation of $\mathcal{L}_1$, i.e., two different papers with authors from two different departments. It is important that the departments belong to different universities. Now assume the second paper has a third author who got his degree from the second university but is a student member of the first department. Furthermore, this student has no other incident edges. Then this student node is not part of any match due to SPARQL. However, dual simulation does not discriminate this node, since it reflects a similar situation and all adjacent nodes dual simulate their respective counterparts in $\mathcal{L}_1$.

The LUBM dataset is especially prone to queries like $\mathcal{L}_1$, since it is a very large dataset with only little diversity in the generated subgraphs (recall that 18 predicates are distributed over 1.33 billion edges). As a consequence of the low diversity, potential matches are often adjacent and dual simulation combines them frequently by edges not belonging to any match. Custom-tailored notions of query matches under dual simulations may give the additional nodes/subgraphs semantics.

### 6 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
Ma et al. [20] 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, the performance improvements by dual simulation comes with a loss of topology [20].

Mottin et al. build on simulation as part of their query paradigm called Exemplar Queries [24]. 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 [31]. 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 efficient 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 [30,26,19,4]. Our proposal appreciates the graph data model and performs light-weight graph algorithms to support traditional database optimization.

A large portion of research has been made in simulation-based indexing techniques, which have already been used for join-ahead pruning in XML databases [22]. The index is created by computing bisimulation equivalence classes of nodes on the original database. Each equivalence class groups structurally bisimilar nodes [28,32]. 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, promising to obtain a much smaller database fingerprint than possible with bisimulations, since simulation equivalence is coarser than bisimulation.

7 Conclusion

In this paper, we proposed efficient processing of SPARQL queries based on graph pattern matching. Our algorithms build 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 BGP, whereas composite queries, including AND and OPTIONAL operators, are handled by conservative extensions of dual simulation.

Our evaluation has shown that we outperform standard dual simulation algorithms on a variety of real-world SPARQL BGP. 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 our technique by directly integrating it 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 and conducting 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


Table 4: Query processing times on the full and pruned dataset, and query times including pruning times for RDFox. All times are measured in seconds.

<table>
<thead>
<tr>
<th>Query</th>
<th>$t_{DB}$</th>
<th>$t_{DB\ \text{pruned}}$</th>
<th>$t_{DB\ \text{pruned}} + t_{\text{SPARQLSIM}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_0$</td>
<td>19.100</td>
<td>1.401</td>
<td>107.852</td>
</tr>
<tr>
<td>$L_1$</td>
<td>25,900.00</td>
<td>888.000</td>
<td>896.464</td>
</tr>
<tr>
<td>$L_2$</td>
<td>161.000</td>
<td>15.690</td>
<td>163.025</td>
</tr>
<tr>
<td>$L_3$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.138</td>
</tr>
<tr>
<td>$L_4$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.125</td>
</tr>
<tr>
<td>$L_5$</td>
<td>0.000</td>
<td>0.000</td>
<td>1.223</td>
</tr>
</tbody>
</table>

| $D_0$ | 1.400   | 1.115         | 5.511                          |
| $D_1$ | 0.000   | 0.000         | 0.002                          |
| $D_2$ | 1.100   | 0.003         | 0.091                          |
| $D_3$ | 0.620   | 0.002         | 0.145                          |
| $D_4$ | 5.960   | 3.493         | 9.722                          |
| $D_5$ | 3.230   | 0.016         | 0.590                          |

| $B_0$ | 1.468   | 0.000         | 0.088                          |
| $B_1$ | 0.099   | 0.030         | 0.052                          |
| $B_2$ | 0.348   | 0.110         | 0.642                          |
| $B_3$ | 0.104   | 0.012         | 0.618                          |
| $B_4$ | 0.033   | 0.000         | 0.000                          |
| $B_5$ | 0.000   | 0.000         | 0.033                          |
| $B_6$ | 12.830  | 0.042         | 0.545                          |
| $B_7$ | 14.410  | 0.002         | 0.445                          |
| $B_8$ | 0.793   | 0.001         | 0.114                          |
| $B_9$ | 0.117   | 0.001         | 0.023                          |
| $B_{10}$ | 0.004  | 0.001         | 0.028                          |
| $B_{11}$ | 0.001  | 0.000         | 0.018                          |
| $B_{12}$ | 0.001  | 0.001         | 0.019                          |
| $B_{13}$ | 0.643  | 0.022         | 0.295                          |
| $B_{14}$ | 3.282  | 1.998         | 6.320                          |
| $B_{15}$ | 0.941  | 0.000         | 0.000                          |
| $B_{16}$ | 0.000  | 0.000         | 0.009                          |
| $B_{17}$ | 0.758  | 0.310         | 1.227                          |
| $B_{18}$ | 0.119  | 0.001         | 0.473                          |
| $B_{19}$ | 18.750 | 0.048         | 0.650                          |
Table 5: Query processing times on the full and pruned dataset, and query times including pruning times for Virtuoso. All times are measured in seconds.

<table>
<thead>
<tr>
<th>Query</th>
<th>$t_{DB}$</th>
<th>$t_{DB}$ pruned</th>
<th>$t_{DB}$ pruned + $t_{SPARQLSIM}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_0$</td>
<td>5.126</td>
<td>2.261</td>
<td>108.712</td>
</tr>
<tr>
<td>$L_1$</td>
<td>50.853</td>
<td>0.971</td>
<td>9.435</td>
</tr>
<tr>
<td>$L_2$</td>
<td>56.676</td>
<td>26.767</td>
<td>174.102</td>
</tr>
<tr>
<td>$L_3$</td>
<td>0.001</td>
<td>0.000</td>
<td>0.138</td>
</tr>
<tr>
<td>$L_4$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.125</td>
</tr>
<tr>
<td>$L_5$</td>
<td>0.000</td>
<td>0.000</td>
<td>1.223</td>
</tr>
<tr>
<td>$D_0$</td>
<td>0.395</td>
<td>0.359</td>
<td>4.755</td>
</tr>
<tr>
<td>$D_1$</td>
<td>0.001</td>
<td>0.000</td>
<td>0.002</td>
</tr>
<tr>
<td>$D_2$</td>
<td>0.002</td>
<td>0.000</td>
<td>0.089</td>
</tr>
<tr>
<td>$D_3$</td>
<td>0.010</td>
<td>0.003</td>
<td>0.147</td>
</tr>
<tr>
<td>$D_4$</td>
<td>2.148</td>
<td>4.008</td>
<td>10.238</td>
</tr>
<tr>
<td>$D_5$</td>
<td>0.039</td>
<td>0.021</td>
<td>0.595</td>
</tr>
<tr>
<td>$B_0$</td>
<td>0.002</td>
<td>0.000</td>
<td>0.088</td>
</tr>
<tr>
<td>$B_1$</td>
<td>0.003</td>
<td>0.001</td>
<td>0.023</td>
</tr>
<tr>
<td>$B_2$</td>
<td>0.003</td>
<td>0.003</td>
<td>0.030</td>
</tr>
<tr>
<td>$B_3$</td>
<td>0.001</td>
<td>0.002</td>
<td>0.020</td>
</tr>
<tr>
<td>$B_4$</td>
<td>0.001</td>
<td>0.002</td>
<td>0.020</td>
</tr>
<tr>
<td>$B_5$</td>
<td>0.054</td>
<td>0.031</td>
<td>0.303</td>
</tr>
<tr>
<td>$B_6$</td>
<td>1.082</td>
<td>0.441</td>
<td>4.762</td>
</tr>
<tr>
<td>$B_7$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>$B_8$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.009</td>
</tr>
<tr>
<td>$B_9$</td>
<td>0.121</td>
<td>0.099</td>
<td>1.016</td>
</tr>
<tr>
<td>$B_{10}$</td>
<td>0.043</td>
<td>0.009</td>
<td>0.031</td>
</tr>
<tr>
<td>$B_{11}$</td>
<td>0.012</td>
<td>0.003</td>
<td>0.476</td>
</tr>
<tr>
<td>$B_{12}$</td>
<td>0.102</td>
<td>0.056</td>
<td>0.658</td>
</tr>
<tr>
<td>$B_{13}$</td>
<td>0.069</td>
<td>0.064</td>
<td>0.596</td>
</tr>
<tr>
<td>$B_{14}$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>$B_{15}$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.034</td>
</tr>
<tr>
<td>$B_{16}$</td>
<td>0.042</td>
<td>0.026</td>
<td>0.594</td>
</tr>
<tr>
<td>$B_{17}$</td>
<td>0.022</td>
<td>0.013</td>
<td>0.516</td>
</tr>
<tr>
<td>$B_{18}$</td>
<td>0.003</td>
<td>0.001</td>
<td>0.444</td>
</tr>
<tr>
<td>$B_{19}$</td>
<td>0.021</td>
<td>0.005</td>
<td>0.118</td>
</tr>
</tbody>
</table>